

CS 330 Homework

Regexercise

1 Overview

Your responsibility in this homework is to fall in love with regular expressions. You will do so in the context of writing a handful of regexes for ten independent pattern-matching and substitution tasks.

Regular expressions are available in many languages. However, in most of these languages, regular expressions are not defined in the syntax of the core language. Rather, they are relegated to a library. Ruby is a notable exception, making regular expressions a first-class type in the language (instead of awkwardly piggybacking strings).

2 Requirements

To receive credit for this homework, you must satisfy these requirements:

1. Place all files in directory `<YOUR-REPOSITORY>/regexercise`.
2. Use a Ruby interpreter 1.9 or greater. Run `ruby -v` to check the version. See the course blog for instructions on installing your own version of Ruby. Older versions of Ruby do not support lookahead assertions.
3. All code must run on `thing-0[456]`.
4. Since Ruby supports a functional style of programming, you should use loops minimally—in particular, `for` and `while` are completely unnecessary. `String.scan` and `gsub` should be your primary tools for iteratively processing text.
5. Write script `wrap` to wrap a file's text at roughly a specified number of characters. The script accepts two command-line parameters: the path to a file containing the text to be wrapped and the number of characters at which to wrap. For example, suppose there's a file in the working directory named `numbers.txt` that contains the following text:

```
12345 789
123 56
12 4 6
```

Running `./wrap numbers.txt 4` produces

```
12345
789
123
56
```

All lines with more than 4 characters are broken at the nearest space after the 4th character. This space and any immediate successors are replaced by a linebreak. The wrapped text is written to standard output.

Implementation hint: find all spans of characters with length in $[1, \text{ncolumns}]$ and followed by a space, linebreak, or end-of-file. These characters represent places where it's legal break a line. Replace the matching text after these spans with a linebreak.

6. Write script `classify` that determines if the name of a Java source file matches the public class that is defined in it. The script accepts one command-line parameter: the path to a Java source file. If the class defined in the file matches the root of the basename of the path, exit with status 0. Otherwise, exit with status 1. Suppose file `../Monster.java` contains the following text:

```
public class Monster {
    ...
}
```

Running `./classify ../Monster.java && echo $?` yields a 0. You may assume that the main class defined in the file is opened all on one line, starting with an unindented `public`. The line may contain modifiers like `abstract` but the class name will be preceded by `class`. Ignore interfaces and `enums`.

7. Write script `import` to sort contiguous sequences of `import` statements in a Java source code file. The script accepts one command-line parameter: the path to a Java source file. It sorts each series of `import` statements independently, and writes the entire reorganized source code to standard output. For example, suppose file `Microwaveable.java` contains the following text:

```
import java.util.Scanner;
import java.io.FileNotFoundException;

import java.awt.Robot;
import edu.uwec.rescom.IoT;

public interface Microwaveable {
    ...
}
```

Running `./import Microwaveable.java` produces the following output:

```
import java.io.FileNotFoundException;
import java.util.Scanner;

import edu.uwec.rescom.IoT;
import java.awt.Robot;
```

```
public interface Microwaveable {
    ...
}
```

Implementation hint: give `gsub` a pattern that matches an entire sequence at a time, then pass the resulting lines to a block that manipulates the text. Look at the Ruby string and array documentation to see what these classes afford you.

8. Write script `mesozoicize` to annotate utterances of the form “# million years ago” in a file with parenthesized references to the appropriate geologic period of the Mesozoic Era. If the number is in [65, 143], the period is Cretaceous. If the number is in [144, 205], the period is Jurassic. If the number is in [206, 248], the period is Triassic. The script accepts a single command-line parameter: the path to a file containing the text to be annotated. It annotates each reference to the Mesozoic Era and writes the results to standard output. For example, suppose `dinos` contains the following text:

```
Eoraptor appeared 231 million years ago.
Stegosaurus appeared 155 millions years ago.
```

Running `./mesozoicize dinos` produces:

```
Eoraptor appeared 231 million years ago (Triassic).
Stegosaurus appeared 155 millions years ago (Jurassic).
```

If statements are not allowed on this problem. Use the range-matching/alternation idea discussed in class. You may assume integral numbers.

9. Write script `preprocess` to insert `#included` text into a file much like the C preprocessor does. The script accepts one command-line parameter: the path to the file to preprocess. Any occurrences of form `#include "some/path"` in the text are replaced by the text contained in the file at `some/path`. (Do not interpret `some/path` literally. Whatever appears between the quotation marks is the path to the file.) The results are written to standard output. For example, suppose `color.h` contains the text `void red();` and `paint.c` contains the following:

```
#include "color.h"
int main() {
    ..
}
```

Running `./preprocess paint.c` produces the following output:

```
void red();
int main() {
    ..
}
```

If the included file contains `#include` directives of its own, also process them.

Implementation hint: recursion requires naming your code so that you can call it, so write a function that takes in a file, reads it, and substitutes all its `#include` directives. Give `gsub` a block that recursively invokes your function on the matched paths.

10. Write script `grepp` to match lines in a file to a specified pattern, similar to `grep`. The script accepts two command-line parameters: a Ruby regular expression and the path to the file whose lines are to be matched. The script writes to standard output any line that matches the pattern. For example, suppose `strings` contains the following text:

```
cow54
dog
cat9
```

Running `./grepp '\d{2}' strings` produces the following output:

```
cow54
```

11. Write script `metricate` to replace imperial measurements in a file with metric measurements. The script accepts one parameter: the path to a file containing the text to be converted. Any measurements conforming to the patterns in the first column of the table below are converted to the forms in the second column using the specified conversion formula:

Pattern	Replacement	Formula
# inch or # inches or # in	# cm	×2.54
# mile or # miles or # mph	# km or # kph	×1.60934
# pound or # pounds or # lbs	# kg	×0.453592
# yard or # yards or # yd	# m	×0.9144

The converted text is written to standard output.

The numbers may contain decimal digits (e.g., 1.75, 90.3, 0.000009). The converted measurements must contain the exact same number of decimal digits (e.g., 50.7 miles → 81.6 km, 5 inches → 13 cm, 145.00001 lbs → 65.77084 kg).

Implementation hint: give `gsub` a pattern that captures both the overall number and the decimal digits. Pass it a block that uses the captured results (available in `$1`, `$2`, etc.) and a format string to produce numbers with the correct number of decimals (e.g., `("%.4f" % 0.12348) → 0.0125`).

12. Write script `idread` that examines all the alphabetic identifiers in a source file and determines if they are *readable*. We define *readable* as being comprised of only of words that are present in `/usr/share/dict/words`. The scripts accepts one command-line parameter: the path to a file containing the identifiers to examine. It iterates through

the list of alphabetic identifiers and writes them to standard output, with each followed by a smiley if the identifier is readable and a frowney otherwise. Identifiers may consist of multiple words, depending on the case of internal letters. For example, **flowerPower** is two words, each of which must be considered separately. For example, suppose **ids** contains the following text:

```
jolly
swick
fwerp
goober
not
Street
holiday
isGreat
isFwungy
hasNothing
isChangedALot
greetPeopleWithASmile
sometimes5
```

Running `./idread ids` produces the following output:

```
jolly :)
swick :)
fwerp :(
goober :)
not :)
Street :)
holiday :)
isGreat :)
isFwungy :(
hasNothing :(
isChangedALot :(
greetPeopleWithASmile :(
```

Ignore case when determining if a word appears in the dictionary. Note that some seemingly-legal identifiers are rejected because they contain verb conjugations not found in the dictionary file.

Implementation hint: use `scan` and a block to process each identifier. In the block, decompose the identifier into words. Holding the contents of `/usr/share/dict/words` in memory is probably not a good idea. Instead, use the `system` command and the `grep` utility to determine if a word is in the dictionary.

This problem is motivated by Landon and Objective-C, who both avoid abbreviations in identifiers.

13. Write script `geocode` to replace all `geo` tags in a file with links to Google Maps. The script accepts one parameter: the path to a file containing `geo` tags. Any instances of `geo:PLACENAME:` are replaced with:

```
<a href="http://maps.google.com/maps?address=LATITUDE, LONGITUDE">PLACENAME</a>
```

Assume `PLACENAME` contains no colons. Write the results of the substitution to standard output. For example, suppose `anim` contains the following text:

```
Pixar's in geo:Emeryville, CA:.
Disney Animation's in geo:Burbank, CA:.
```

Running `./geocode anim` produces the following output:

```
Pixar's in <a href="http://maps.google.com/maps?q=37.831,-122.285">Emeryville, CA</a>.
Disney Animation's in <a href="http://maps.google.com/maps?q=34.181,-118.309">Burbank, CA</a>.
```

Determine the latitude and longitude using Google's geocoding web service¹. A query is issued and its results are retrieved in the following manner:

```
require 'open-uri'
require 'json'

# insert code to detect all tags
# for each place
  place = ...
  params = {:address => place}
  url = URI.parse("https://maps.googleapis.com/maps/api/geocode/json")
  url.query = URI.encode_www_form(params)
  body = JSON.parse(url.read)
  sleep 1
```

Variable `body` contains a hash/dictionary built from the JSON-formatted response. Use the `lat` and `lng` fields of the `location` structure of the geometry `geometry` structure of the 0th entry of the `results` array. Round the results to three-decimal places.

The `sleep` command is very important. If you issue queries too quickly, Google will block your requests.

14. Write script `deadlinks` to determine whether each link in a file is *dead*. We define *dead* as not responding to an HTTP HEAD request. The script accepts one parameter: the path to a file whose links are to be examined. It iterates through all links in the file and writes them to standard output, with each followed by a smiley if the link is not dead and a frowney otherwise. For example, suppose `bookmarks` contains the following text:

```
Go to http://www.asdkfjasdklfjaasdfjasdfjk.com/ (Please don't register this domain.)
and http://www.sometimesredsometimesblue.com/
```

Running `./deadlinks bookmarks` produces the following output:

```
http://www.asdkfjasdklfjaasdfjasdfjk.com/ :(
http://www.sometimesredsometimesblue.com/ :)
```

¹<https://developers.google.com/maps/documentation/geocoding/>

You can assume all links start with `http` and end right before a quotation mark, whitespace, or end-of-line. (This definition is less than rigorous.)

Implementation hint: you can issue a HEAD request to a web page in the following manner:

```
require 'net/http'
# ...
url = URI.parse(url)
request = Net::HTTP.new(url.host, url.port)
response = request.request_head(url.path)
```

Consider the request successful if `response.code` is less than 400. A 404 error, for example, means the link is dead.

To be eligible for later-week submission, you must successfully complete `wrap` and `classify`. Additionally, the grading script must report that you met these requirements.

3 Submission

To submit your work for grading:

1. Run the grading script from your homework directory using `../specs/grade`.
2. Commit and push your work to your repository.
3. Verify that your solution is on Bitbucket by viewing your repository in a web browser.

A passing grading script does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The grading script checks some of them, but not all.
- You must successfully submit your code to your repository. Expect to have issues with Git.
- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don't want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.

The grading script allows you to signal your instructor when requirements are met. You only need to send an email if you are resubmitting.