Regular Expressions

Background
definitions
regular expressions in formal language theory
regular expressions in Unix
RegExp objects in Ruby

Patterns

✦ Regular expressions give us a method for doing string searches based on patterns
✦ In a normal search we’re looking for instances of a specific string

```ruby
>> s = "supercalifragilisticexpialidocious"
=> "supercalifragilisticexpialidocious"
>> s.length
=> 34
>> s.index("list")
=> 14
>> s.index("array")
=> nil
```
✦ In CS terminology the index method implements an exact match search

History

✦ The term “regular expression” comes from formal language theory
✦ Items in a regular expression can be
 ✦ symbols from the alphabet (characters)
 ✦ special symbols used as place-holders (meta-characters)
    • matches any character
    • matches “one or more of the preceding symbol”
    • matches “zero or more of the preceding symbol”
✦ Examples (using the alphabet of lower case letters):
  "ab" matches only one string, "ab"
  "a.a" matches "aaa", "aba", ...
  "a+a" matches "aa", "aaa", "aaaa", ...
  "a.a+" matches "aa", "aaa", "aaaa", "aaaaa", ...
  "a.*a" matches "aa", "aaa", "aaaa", ..., "aba", "aba+a", "aba+a", ...
  (anything string of any length that starts and ends with a)
History

- Shell commands have pattern matching similar to regular expression matching
- but note that in a shell command *, means “0 or more letters”
- in a regular expression, “0 or more letters” is . (note the dot)
- Regular expressions became widely used when they were added to Perl
- Perl extended the basic regular expression syntax, adding symbols to mean “any letter”, “any digit”, “start of line”, and many others
- The term “Perl regular expression” means “the notation used to specify patterns in Perl”
- Ruby has Perl regular expressions (but with a much nicer way to use them)

To Learn More

- The Thomas book is the definitive reference understanding how to create and use regular expressions in Ruby
- If you’re really interested there are entire books on pattern matching algorithms
- regular expressions

Basic Pattern Matching

- To see if a string contains a pattern use the =~ operator:

  - the item on the left is a string, the item on the right is a regular expression
  - regular expressions are strings enclosed in slashes
  - the value of s =~ r is the index of the first substring of s that matches r
  - the value of the expression is nil if the string does not contain the pattern

  ```ruby
  line = "supercalifragilisticexpialidocious"
  line =~ /a.*a/  # => 6
  line =~ /a.*z/  # => nil
  ```

RegExP Objects

- Strings that start and end with a slash are instances of the RegExp class:

  ```ruby
  r = /a.*a/  # => /a.*a/
  r.class  # => RegExp
  line =~ r  # => 6
  ```
Another Look at [ ]

- Examples on previous slides showed how to use the [ ] operator with strings
  - `line = "supercalifragilisticexpialadocious"
  - `line[3]` => 101
  - `?e` => 101

- This operator also allows us to put a string between the brackets:
  - `line["frag"]` => "frag"
  - `line["frog"]` => nil
  - `puts "found it" if line["frag"]`
    found it

- `s.index(t)` and `s[t]` are different ways to see if `t` is a substring of `s`

Greedy Match

- The expression in this example finds a substring that starts and ends with `a`:
  - `line = "supercalifragilisticexpialadocious"
  - `line[/a.*a/]` => "alifragilisticexpiala"

- The output illustrates an important point:
  - **regular expression matches are greedy**
    - the algorithm that does the matching finds the longest possible substring to make the match
    - in this example, the search for `/a.*a/` finds the substring from the **first a** to the **last a**
    - there is a way (illustrated later) to ask Ruby to find the shortest match (aka lazy match)

Another Look at [ ]

- We can also put a regular expression between the brackets
  - The result is the part of the string that matches the regular expression
    - `line = "supercalifragilisticexpialadocious"
    - `line["frag"]` => "frag"
    - `line[/cal/]` => "cal"
    - `line[/f.*g/]` => "frag"
    - `line[/p.*p/]` => "percalifragilisticexp"

Regular Expressions and Scripts

- Extracting the part of a string that matches a pattern is a very common operation in scripts
  - scripts read data from files or lines generated by other applications
  - we often want to scan through these strings to find important nuggets

- Example: get the current time from a date string
  - `t = `date`` => "Mon Feb 18 20:42:25 PST 2008\n"
  - the time is the part of this string that has a set of two-digit numbers separated by colons
    - `t[/..:..:../]` => "20:42:25"
More Meta-Characters

- The meta-characters we've seen so far are . + and *
- Ruby regular expressions have many more meta-characters
  \d a single digit (any of the characters '0' to '9')
  \w a single “word” character ('0' to '9' or 'a' to 'z' or 'A' to 'Z')
  \s a single “whitespace” character (space, tab, newline, etc)

```
>> size = 'wc wordlist.txt'
=> "58022 58022 585858 wordlist.txt"
>> size[/\d+/]
=> "58022"
>> t = 'date'
=> "Mon Feb 18 21:04:41 PST 2008"
>> t[/\w+/]
=> "Mon"
```

Even though it requires two or more characters to type them, we call them "meta-characters"

- Characters placed between square brackets define a set; the pattern matches any one character in the set
  [aeiou] a vowel
  [a-z] a lower case letter
  [a-zA-Z] a lower or upper case letter
  [;:'".,!?] one of these punctuation marks
  ['aeiou'] any character except a vowel

```
>> s = "Programming Ruby (2e) has > 800 action-packed pages."
>> s[/[\w\d]/]
=> "Programming"
>> s[/[\d]+/]
=> "803"
```

```
Even though it requires two or more characters to type them, we call them "meta-characters"
```

```
A better way to write the last example:
>> r = /[;:'".,!?]/
>> s[r]
=> "."
```

```
Meta-characters . + * etc lose their special meaning when between square brackets
```

```
Meta-character ^ means "not" when it's the first character between square brackets
```

```
A special class of meta-characters are known as anchors
They match locations in a string, but not any characters in the string

^ the beginning of the string
$ the end of the string
\b a word boundary
```

```
>> s = "Programming Ruby (2e) has > 800 action-packed pages."
>> s[/\w{4,}/]
=> "Programming"
>> s[/[\d]+/]
=> "803"
```

```
>> s[/\b[\d]+\b/]
=> "803"
>> s[/[\b[a-z]+\b]/]
=> "has"
>> s[//>]/
=> "="
>> s[/>/]/
=> nil
```

```
^ and $ are often used to test the structure of a line, e.g.
if line[/>/] ... is one way to see if a line is a define
```

```
It can be very tricky to get just the right expression (e.g. "lower case word with at least four letters"). Plan on spending some time in irb testing different combinations....
```

More Meta-Characters

- Use a multiplier to specify the number of times an item must occur
  (n) exactly n copies of the preceding item
  (n,m) between n and m copies of the preceding item
  (n,) at least n copies

```
>> s = "Programming Ruby (2e) has > 800 action-packed pages."
>> s[/\w\{4,\}/]
=> "Programming"
>> s[/[\w]\{4,\}/]
=> "Programming"
>> s[/[\d]\{4,\}\b/]
=> "action"
>> s[/[\d][\w]\{4,\}\b/]
=> "io"
>> s[/[\w]\{3\}/]
=> nil
>> s = /[\w]\{2\}/
=> 35
```
Lazy Matches

- An earlier example showed that the pattern matcher is greedy:

```
line = "supercalifragilisticexpialadocious"
line[/a.*a/]
=> "alifragilisticexpiala"
```

- The match above is the longest substring starting and ending with a

- To find the shortest match, use \?

```
x? means "the fewest number of item x required to make a match"
line[/a.*?a/] => "alifra"
```

Lazy Match Example

- Transfer RNA (tRNA) molecules are short sequences of the same bases found in DNA
  - actually has U instead of T, but sequence databases often have T...
  - A project I worked on wanted to align tRNA sequences to a 2D "template"

```
All tRNAs have GG here

To fit this sequence to the template, insert a gap (a gray circle) before GG
```

Quoting Special Characters

- With all these meta-characters (., *, ?, [^ etc) there is bound to be a situation where you want to write a pattern that includes a meta-character
  - e.g. how would you see if a sentence ends in a question mark?

```
s = "Hello, how are you?"
s =~ /?$/(irb):68: invalid regular expression; there's no previous pattern, to which '?' would define cardinality at 1: /?$/
```

- To "quote" a meta-character put a backslash in front of it:

```
s = "Hello, how are you?"
s =~ /\?$/(irb):68: invalid regular expression; there's no previous pattern, to which '?' would define cardinality at 1: /\?$/
```

- Note: you can also put a slash in a pattern by quoting it

```
s = "Add 1/2 cup chopped hazelnuts"
s =~ /\d\d\d/ => 18
```
Example: Yeast ORF IDs

- Genes in yeast are often identified by an **ORF name**
  - ORF = "open reading frame"
- The standard nomenclature is
  - the letter Y
  - a chromosome ID (A through P)
  - the letter L or R (left or right "arm" of the chromosome)
  - a three-digit integer
  - the letter C (complementary strand) or W
- Examples:
  - YAR033W
  - YDL055C

Example: Yeast ORFs

- As an in-class exercise let's come up with regular expressions for ORF names
  - a pattern for any gene on a complementary strand:
    `/Y.....C/`
  - looks promising, but this pattern also matches other things besides ORF names
    ```
    $ egrep 'Y.....C' CHR_1/NC_001133.gbk
    /locus_tag="YAR033C"
    TAKNYP14AVTMAETAVIAEQIAAYLPNYDDSNCTFEPSTSTETVASTAVAAVF
    /locus_tag="YAL040C"
    
    how can we fix this so it's Y followed by A, B, ... or P?
    how about Y followed by A, B, ... P and then an L or R?
    `/Y[A-P][L]...C/`
    ```
  - still one “false positive” on chromosome I:
    ```
    KHTDQVVAIEVVYENDEELNDIABESLLKLNHNIVKSHGFIRKSYELVIYELC
    ```
- how about Y followed by A, B, ... P and then 3 digits?
  `/Y[A-P][L]...[0-9]{3}C/`
- final step: a pattern that matches any gene
  `/Y[A-P][L]...[0-9]{3}[CW]/`
  or alternatively
  `/Y[A-P][L]...[0-9]{3}[CW]/`
- Writing regular expressions is like programming in general
  - involves an initial attempt followed by lots of debugging as you discover some of your original assumptions were wrong
- Advice: start with a simple, general pattern and refine it step by step
- And of course: use irb
  - cut an paste a sample input string
  - try looking for patterns with s[r] or s =~ r

Example: Crossword Puzzle

- Here is a partially filled-in puzzle:
  - clue for 1A: “Bristly”
  - clue for 1D: “Relatively smart”
  - clue for 4D: “1937 Oscar role for Luise Rainer” (?!?)
- Let's scan the word list for potential answers
  ```
  $ egrep '.et.se' wordlist.txt
  acetose
  betise
  ketose
  metasequoia
  ...
  ```
  - It's a start, but we want only 6-letter words
Example: Crossword Puzzle

- What we need are anchors that say the pattern must begin at the start of the string and end at the end of the string:

  ```
  !egrep '^.*se$' wordlist.txt
  betise
  ketose
  setose
  ```

- Clue for 1A: “Bristly”

In-Class Projects

- What are regular expressions that match the following?
  - **time**
    - 5:22AM or 8:52 P.M.
  - **url**
    - `http://www.xxx.yyy`
  - **quoted strings**
    - “hello, world”