More Details about Regular Expressions

Basic Regular Expression Notation

- Summary of basic notations to match single characters and sequences of characters

1. `/[abc]/ = /a|b|c/` — Character class; disjunction matches one of a, b or c
2. `/[b-e]/ = /b|c|d|e/` — Range in a character class
3. `/[^b-e]/` — Complement of character class

4. `/./` — Wildcard matches any character

5. `/a*/ /[af]*/ /(abc)*/` — Kleene star: zero or more
6. `/a?/ /(ab|ca)?/` — Zero or one; optional
7. `/a+/ /([a-zA-Z]1|ca)+/` — Kleene plus: one or more

8. `/a{8}/ /b{1,2}/ /c{3,}/`
   Counters: exact number of repeats

Regular Expressions

- Anchors
  - Constrain the position(s) at which a pattern may match
  - Think of them as "extra" alphabet symbols, though they actually match ε (the zero-length string):
    - `/^a/` Pattern must match at beginning of string
    - `/a$/` Pattern must match at end of string
    - `/\bword23\b/` "Word" boundary: `/[a-zA-Z0-9_][^a-zA-Z0-9_]/` following `/[^a-zA-Z0-9_][a-zA-Z0-9_]/`
    - `/\B23\B/` "Word" non-boundary
- Parentheses
  - Can be used to *group* together parts of the regular expression, sometimes also called a *sub-match*

Regular Expressions

- Escapes
  - A backslash "\" placed before a character is said to "escape" (or "quote") the character. There are six classes of escapes:
    1. **Numeric character representation**: the octal or hexadecimal position in a character set: "\012" = "\xA"
    2. **Meta-characters**: The characters which are syntactically meaningful to regular expressions, and therefore must be escaped in order to represent themselves in the alphabet of the regular expression: "[](){}|^$.?+*\" (note the inclusion of the backslash).
    3. **"Special" escapes** (from the "C" language):

newline:	"\n" = "\xA"	carriage return:	"\r" = "\xD"
tab:	"\t" = "\x9"	formfeed:	"\f" = "\xC"

Regular Expressions

- Escapes (continued)
  - Classes of escapes (continued):
    4. **Aliases**: shortcuts for commonly used character classes. (Note that the capitalized version of these aliases refer to the complement of the alias's character class):
       - whitespace: "**\s**" = "`[ \t\r\n\f\v]`"
       - digit: "**\d**" = "`[0-9]`"
       - word: "**\w**" = "`[a-zA-Z0-9_]`"
       - non-whitespace: "**\S**" = "`[^ \t\r\n\f]`"
       - non-digit: "**\D**" = "`[^0-9]`"
       - non-word: "**\W**" = "`[^a-zA-Z0-9_]`"
    5. **Memory/registers/back-references**: "`\1`", "`\2`", etc.
    6. **Self-escapes**: any character other than those which have special meaning can be escaped, but the escaping has no effect: the character still represents the regular language of the character itself.

Regular Expressions

- Greediness
  - Regular expression counters/quantifiers which allow for a regular language to match a variable number of times (i.e., the Kleene star, the Kleene plus, "`?`", "`{min,max}`", and "`{min,}`") are inherently *greedy:*
    - That is, when they are applied, they will match as many times as possible, up to `max` times in the case of "`{min,max}`", at most once in the "`?`" case, and infinitely many times in the other cases.
    - Each of these quantifiers may be applied non-greedily, by placing a question mark after it. Non-greedy quantifiers will at first match the **minimum** number of times.
    - For example, against the string "From each according to his abilities":
      - `/\b\w+.*\b\w+/` matches the entire string, and
      - `/\b\w+.*?\b\w+/` matches just "From each"

How to use Regular Expressions

- In Perl, a regular expression can just be used directly for matching, the following is true if the string matches:

  string =~ m/ <regular expr> /

- But in many other languages, including Python (and Java), the regular expression is first defined with the compile function

  pattern = re.compile(“<regular expr>”)

- Then the pattern can be used to match strings

  m = pattern.search(string)

  where m will be true if the pattern matches anywhere in the string

- Another option is to use the function

  re.match(“<regular expr>”, string)

  which combines compile with the match function (next page)

More Regular Expression Functions

- Python includes other useful functions
  - pattern.match – true if matches the beginning of the string
  - pattern.search – scans through the string and is true if the match occurs in any position
    These functions return a "MatchObject" or None if no match found
  - pattern.findall – finds all occurrences that match and returns them in a list
- MatchObjects also have functions to find the matched text
  - match.group( ) – returns the string(s) matched by the RE
    - Includes all the subgroups indicated by internal parentheses
  - match.start( ) – returns the starting position of the match
  - match.end( ) – returns the ending position of the match
  - match.span( ) – returns a tuple containing the start, end
  - And note that using the MatchObject as a condition in, for example, an If statement will be true, while if the match failed, None will be false

Substitution with Regular Expressions

- Once a regular expression has matched in a string, the matching sequence may be replaced with another sequence of zero or more characters:
  - Convert "red" to "blue"
    - Perl: $string =~ s/red/blue/g;
    - **Python: p = re.compile("red") string = p.sub("blue", string)**
  - Convert leading and/or trailing whitespace to an '=' sign:
    - Python:
```
p = re.compile("^\s+|\s+$")
string = p.sub("=",string)
```
  - Remove all numbers from string: "These 16 cows produced 1,156 gallons of milk in the last 14 days."
    - Python:
```
p = re.compile(" \d{1,3}(,\d{3})*")
 string = p.sub("",string)
```
    - The result: "These cows produced gallons of milk in the last days."

Extensions to Regular Expressions

- Memory/Registers/Back-references
  - Many regular expression languages include a memory/register/back-reference feature, in which sub-matches may be referred to later in the regular expression, and/or when performing replacement, in the replacement string:
    A sub-match, or a match group, is defined as matching something in parentheses (as in the /(\w+)/), and the back-reference /\1/ says to match the same string that matched the sub-match:
    - Perl: `/(\w+)\s+\1\b/` matches a repeated word
    - Python:
      ```
      p = re.compile("(\w+)\s+\1\b)")
      p.search("Paris in the the spring").group()
      ```
      returns 'the the'
  - Note: finite automata cannot be used to implement this memory feature.
- If you want to use internal parentheses without triggering this features, follow the left paren with ?: to make a non-capturing subgroup: Python: p = recompile("\w+\s+(?:\w+\.)+")

Regular Expression Examples

Character classes and Kleene symbols

[A-Z] = one capital letter

[0-9] = one numerical digit

[st@!9] = s, t, @, ! or 9 (equivalent to using | on single characters)

[A-Z] matches G or W or E (a single capital letter)

does not match GW or FA or h or fun

[A-Z]+ = one or more consecutive capital letters

matches GW or FA or CRASH

[A-Z]? = zero or one capital letter

[A-Z]* = zero, one or more consecutive capital letters

matches on EAT or I

so, [A-Z]ate

matches Gate, Late, Pate, Fate, but not GATE or gate

and [A-Z]+ate

matches: Gate, GRate, HEate, but not Grate or grate or STATE

and [A-Z]*ate

matches: Gate, GRate, and ate, but not STATE, grate or Plate

Regular Expression Examples (cont'd)

Some longer examples:

([A-Z][a-z]+)\s([a-z0-9]+)

matches: Intel c09yt745 but not IBM series5000

[A-Z]\w+\s\w+\s\w+[!]

matches: The dog died!

It also matches that portion of " he said, " The dog died! "

[A-Z]\w+\s\w+\s\w+[!]$

matches: The dog died!

But does not match "he said, " The dog died! " because the $ indicates end of Line, and there is a quotation mark before the end of the line

(\w+ats?\s)+

parentheses define a pattern as a unit, so the above expression will match:

Fat cats eat Bats that Splat

Helpful Regular Expression Websites

1. Free interactive testing/learning/exploration tools:
   a. Regular Expression tester:
      https://www.regexpal.com/

2. Tutorial:
   a. The Python Regular Expression HOWTO:

      https://docs.python.org/3/howto/regex.html

      A good introduction to the topic, and assumes that you will be using Python.

3. Regular expression summary pages
   a. Dave Child's Regular Expression Cheat Sheet from addedbytes.com
      http://www.cheatography.com/davechild/cheat-sheets/regular-expressions/