7
Regular Expressions

What is a Regular Expression?

- Regular expressions are a useful tool for string processing
- The term “regular expression” is often abbreviated as “regex”
- The 3 participating components of regular expressions:
  - The search pattern - definition of what is to be “matched”
  - The input string - the data the search pattern will “match on”
  - The regex engine - software component that “matches” the search pattern to the input string
- A regex is used to define a search pattern
- A regex engine then uses this search pattern to match on text in some input string

What can Regular Expressions do?

- Searching
- Find and replace text
- String splitting
- Data clean up
- User input validation
- Renaming files
- Database queries

Some common regex applications

- Detecting the presence or absence of a given pattern in an input string.
- Extracting substrings matching all or portions of a pattern.
- Splitting input strings into substrings based on separators specified as patterns.
- Search and replace operations.

Where did Regular Expressions come from?

- Theoretical (mathematical) concepts began in the 1950s, led by Stephen Cole Kleene
- First implemented in code in the 1960s
- 2 flavors of regex: The Unix or POSIX version, and the Perl version
- The Unix version is older, having its start way back in the early days of Unix - 1960s and is used in various Unix tools like grep, sed, awk, and lex.
- The Perl version came about in the late 1980’s and is much more powerful.
- Today there is support for regex in a number of languages.

How to Regex in Perl

- Search pattern definitions are enclosed using // or any “pair” of non alphanumeric non-white-space characters. Ex: !!, ##, [], {}
- For example /hello/
- The binding operator =~ (equals sign followed by a tilde) is the Perl operator for performing a regex match test.
- The syntax pattern is <input string> =~ /search pattern string/
- When placed in an if statement, this expression returns true if a match is found, and false otherwise (in scalar context)
- Ex: if (“one tacocat is like any other tacocat” =~ /tacocat/)
- There is also the !~ operator that negates whatever is returned by =~

How to Regex in Python

- Regex in Python is done using the re module
- Search pattern strings are declared as raw strings using the ‘r’ prefix. Raw strings are taken as literal. Ex: ‘\n’ would be just \ and n without any special meaning.
- One regex syntax is “match = re.search(search_pattern, input_string)”
- When a match is made, a MatchObject instance is returned
- If no matches are found, then None is returned
- The MatchObject class has several methods for working with the characteristics of the match, such as where in the input string the match occurred.

The Mechanics of Matching

- The matching algorithm can be visualized by the idea of starting at the beginning of the input string and checking for matches against each character in the search pattern.
- Another way of describing the matching process is through the notion of the regex consuming the characters of the input string as the matching between the regex and the input string proceeds.
- But keep in mind that these are mental aids in helping to understand the matching process. The actual implementations used by a regex engine involve some advanced computer science concepts surrounding data structures and algorithms.

The Mechanics of Matching

- When there exist alternatives in a regex for scoring a match with an input string, the regex engine seeks the earliest possible match and, as soon as the engine is successful, stops trying out any remaining alternatives, even if one of the remaining alternatives provides what seems like a better match.
- Ex: $input_string = ‘hellosweetsie’; $regex = h(ey|ello|i)(sweet|sweetsie) will result in a match on the string hellosweet, even though hellosweetsie is present in the input string.
- A global modifier can be used to match more than a single occurrence of a regex in the same input string.

Regex Metacharacters

- Regex patterns can contain special metacharacters that affect how the matching is carried out.
- These metacharacters are embedded within the pattern matching string.
- Metacharacters can be used to match more than one occurrence, match only at the start of a string, match on certain classes of characters like only numeric, and so forth.
- They add flexibility and power to matching while maintaining conciseness.
- The good news is that both Perl and Python (as well as many other regex-capable programming languages) use the same set of metacharacters.

Matching at line and word boundaries

- Anchor metacharacters - metacharacters that control the location of where in an input string the matches can occur.
- ^ forces the match to take place at the beginning of the input string.
  - Ex: ^abra matches abracadabra but not cabradababra
- $ forces the match to take place at the end of the input string.
  - Ex: dabra$ matches abracadabra but not dabracababra
- \b matches either on a word-to-nonword boundary, or a nonword-to-word boundary.
  - Nonword to word: \bwhat matches “whatever man” but not “somewhat delirious”
  - Word to nonword: ever\b matches “whatever will” but not “everywhere to me”

Character Classes

- So far all the search patterns have been literal strings. But what if you wanted to specify a more flexible search pattern?
- Character Classes specify a set of choices to match on at a specific character position in the regex.
- Square brackets [] are used to enclose the set of choices and the position of the [] in the regex determines where these characters are matched.
- Ex: [cgl]arry would match on carry, garry, or larry

Metacharacters used by Character Classes

- - the range operator. Allows for a sequence of numerically or alphabetically contiguous characters. Loses its special meaning if first or last character inside the square brackets. Ex: [a-f] or [0-9]
- ^ the negation operator. Negates the entire character class. Loses its special meaning if appears anywhere except the beginning of the character class. Note: Does NOT imply an absence of a character at its position in the regex. Ex: [^z]ap will match cap, lap, or map, but not ap or zap.

Metacharacters used by Character Classes

-] marks the end of the character class. Ex: [^c]at
- \ escapes the special meaning of the next character in a character class. Ex: [\^\]-]at would match ^at or]at or -at.
- Perl allows for string interpolation inside of a regex so it understands what the $ means when used with a variable inside a regex. Ex: ^[{$hy}y]ello would match “Jello” if $hy had the value ‘J’
- Another metacharacter is the | or operator used inside of parentheses to create a choice between matching the left or right: Ex: \bJo(e|seph)\b

Extracting substrings from an input string

- Parentheses in a regex are known as the grouping metacharacters.
- Both Perl and Python store the substrings from the input-string that match the parenthesized portions of a regex.
- In Perl these stored strings are called match variables and are named $1, $2, $3, $4,. . .$n.
- In Python the stored strings are accessed with the re.group(1), re.group(2), re.group(3), re.group(4),. . .re.group(n) syntax.
- For Perl and Python, the substrings from the input string that match the parenthesized portions of a regex are made available inside the regex itself through what are known as backreferences.

Quantifier Metacharacters

- Quantifier metacharacters control the number of repetitions of the immediately preceding smallest possible subexpression in a regex.
- * means zero or more repetitions of the preceding portion of the regex.
- + means at least one occurrence of the preceding subexpression.
- ? means one or zero, the subexpression can appear once or not at all.
- {} can bound the number of repetitions to a range. Ex: a{m,n}
- * is equivalent to {0,}
- + is equivalent to {1,}
- ? is equivalent to {0,1}

Greediness of Quantifiers

- All the quantifiers we have looked at so far are greedy.
- Greediness in this context means that a quantifier will consume a maximum number of characters from the input string during the matching process, even when a smaller number of characters could have fulfilled the constraints imposed by the quantifier.
- The need to match at the earliest possible position in an input string dominates over the needs of the quantifiers to be greedy.
- The greedy approach can sometimes have unintended consequences.

Nongreedy Quantifiers

- If the greedy behavior of the quantifiers produces undesirable string matching results, the nongreedy versions may be used.
- The nongreedy quantifiers are the greedy quantifiers with an extra ? tagged on the end.
- *? - nongreedy *
- +? - nongreedy +
- ?? - nongreedy ?
- {}? - nongreedy {}
- The difference is these quantifiers will choose as few characters as possible to match on, instead of the most possible.

Match Modifiers

- Match modifiers control various aspects of the matching operation.
- Perl uses the //i modifier for case-insensitive matching.
  - /gross/i would match Gross, gRoss, GroSS, etc
- In Python the regex would be compiled with the re.IGNORECASE flag.
  - pattern = r’gross’
  - regex = re.compile(pattern, re.IGNORECASE)
  - m = re.search(regex, input_string)

Global Matching

- Instead of a regex matching on only the first match it comes across scanning the input string left to right, what if you wanted the regex engine to continue chugging along and scan the entire input string for all possible positions where there exist matches with the regex.
- The global option will do just that.
- Perl uses the //g match modifier to match globally.
- Python uses the re.findall()

Multi-line input strings

- So far all the input strings we have matched on were a single line, i.e. if a newline character was present, it was at the end of the string.
- But what if you wanted to match on a string that had newline characters in the middle? Ex: $input_string = “planes\ntrains\nautomobiles\n”;
- One solution would be to modify the behavior of the anchor metacharacters ^ and $ and the . character class.
- Perl uses the modifiers //m and //s to accomplish this.
- Python uses the re.DOTALL and re.MULTILINE for the same purpose.

Multi-line Regex

- Sometimes a very long regex can be made to be more readable if split over multiple lines.
- Perl provides the //x match modifier to allow a regex to span multiple lines as well as include trailing comments on each line.
- Python uses the re.VERBOSE flag passed to the re.compile() method to create a compiled version of a multiline regex. Comments at the end of each line are also supported.
- Recall that a multiline string in Python uses the triple-quote form. This also applies to multiline regex definitions.

Splitting Strings

- Perl provides the split() function. First argument is a regex defining the pattern of the separator. Second argument is the string to be split.

  my $pattern = ‘\s+’;

  my @words = split /$pattern/, $input_string;

- Python uses the re.split() function in a similar fashion.

  regex = r’\s+’

  words = re.split(regex, input_string)

Search and Replace

- Perl has the s/// operator for searching and replacing matched text with a replacement string.

  s/pickle/jalapeno/

- Perl also has the //e modifier that will evaluate an expression and use the result as the replacement text.
- Python uses the re.sub() function to replace matched text. Usage is re.sub(regex, replacement_string, input_string)
- Python does not have an equivalent to Perl’s //e modifier.

Regex Assignments

- Read this webpage https://developers.google.com/edu/python/regular-expressions

- Watch this video https://www.youtube.com/watch?v=kWyoYtvJpe4 and follow along by running the code examples on your machine in your terminal window