Regular Expressions (V 2.0)

A regular expression can be implemented with a finite-state automation. Regular expressions exist in web searches. The following uses the PERL syntax. A regular expression requires a corpus and a pattern. The following assumes that the text is encapsulated in lines.

Basic Pattern *enlcosed in /’…..’/*

|  |  |
| --- | --- |
| /woodchucks/ | Interesting links to woodchucks and lemurs. |

Disjunction of characters *[abc]*

|  |  |
| --- | --- |
| /[wW]oodchuck/ | Matches Woodchuck or woodchuck |
| /[1234567890]/ | Matches any digit. “Plenty of 7 to 5” |

Character Ranges *[a-z]*

|  |  |
| --- | --- |
| /[2-5]/ | Any number between 2 and 5 (inclusive) |
| /[A-Z]/ | Upper case “We should call this doggy Hachiko” |
| [-A-Z] | This includes the dash. |

Negation of a single character */[^a]/*

|  |  |
| --- | --- |
| /[^a]/ *if the caret occurs any where else (not after the first opening bracket) it usually stands for caret* | Maches any single character except for a. |
| [^A-Z] | Not an upper case letter. |
| [^Ss] | Not ‘S’ or ‘s’ |
| [^\.] | Not a period. |

Optional – *the preceding character or nothing* */colou?r/*

|  |  |
| --- | --- |
| /woodchucks?/ | woodchuck or woodchucks |
| /colou?r/ | colour or color |

Kleene \* – *0 or more /a\*/*

|  |  |
| --- | --- |
| /a\*/ | Any string of zero or more a’s. Will match a or aaaaaa but will also match ‘Off Minor’ |
| /aa\*/ | Matches one or more a’s. |
|  | One or more digits |

Kleene + – *1 or more /a+/*

|  |  |
| --- | --- |
| /[0-9]+/ == /[0-9][0-9]\*/ | One or more digits. |

Wildcard . */fu.k/*

|  |  |
| --- | --- |
| /beg.n/ | Matches: begin or begun. |
| /aardvark.\*aardvark/ | Matches “aardvark” occuring twice. |

Anchors *^ or $*

|  |  |
| --- | --- |
| /^The/ | Only at the start of a line |
| /^The dog\.$/ | Matches a line containing only “The Dog.” |

Word Boundary */\bthe\b/*

|  |  |
| --- | --- |
| /\bthe\b/ | Matches “the” but not “other” Perl defines a word as any sequence of digits, underscores or letters. |
| /\b99\b/ | “The cost is $99” “$” is not a digit nor an underscore nor a letter. |

Disjunction and Parenthesis */cat|dog/*

|  |  |
| --- | --- |
| /cat|dog/ | This matches the string “cat” or “dog” |
| /guppy|ies/ | Matches “guppy” or “ies” |
| /gupp(y|ies) | Matches guppy or guppies. |
| /the\*/ | Matches *“theeeeee”* but not *“thethe”* |
| /(the)\*/ | Matches *“thethe” but not “theeeeee”* |

Greedy expressions

|  |  |
| --- | --- |
| /[a-z]\*/ | This matches zero or more letters. When matching against *“Once upon a time”* This could match nothing, or just the first letter or the whole thing. In cases where the regular expression matches the largest string that they can, we say that the expression is ***greedy.*** |

Simple Example (matching *‘the’*)

|  |  |
| --- | --- |
| /the/ | Will not match “The” at the start of a sentence |
| /[Tt]he/ | This still returns cases where “the” is embedded in other words. Such as ‘other’ |
| /\b[Tt]he\b/ | This will not treat underscores and numbers as word boundaries which we may require. *We want to specify that we want instances in which there are no alphabetic letters on either side of the ‘the’* |
| /[^a-zA-Z][Tt]he[^a-zA-Z]/ | But if ‘the’ is at the start of a line or the end of a line then this will not match as it assumes that a non alphabetic charcter always precedes or procedes the character ‘the’ |
| /(^|[^a-zA-Z])[Tt]he([^a-zA-Z]|$)/ | Finally we have a regular expression that avoids:  False Positives: Finding words such as “other” and “there”  False Negatives: Missing words |

Specific Ranges {n,m}

|  |  |
| --- | --- |
| /a\.{24}z/ | Will match a followed by 24 dots followed by z |
| {n,m} | From n to m occurrences of previous character or expression |
| {n,} | At least n occurences |

Advanced Operators

|  |  |  |
| --- | --- | --- |
| Regular Expression | Expansion | Result |
| \d | [0-9] | Digit |
| \D | [^0-9] | Non digit |
| \w | [a-zA-Z0-9\_] | Alphanumeric & underscore |
| \W | [^\W] | Non-alphanumeric |
| \s | [<space>\r\t\n\f] | Whitespace |
| \S | [^\s] | Non whitespace |

*Regular Expression Substitution and Memory*

The following substitutes ‘color’ to replace ‘colour’:

s/colour/color

*Regular Expression Memory*

If we wanted to put angle brackets around all integers in a text, for example changing ’35 boxes’ to ‘<35> boxes’ well we can do this with the number operator \1 in the replacement pattern to refer back.

s/([0-9]+)/<\1>/

*Specifying repeating patterns with pronumerals (called registers in RE vernacular)*

Suppose we are looking for the pattern: *“the Xer they were, the Xer they will be”* We do this by surrounding the first X with the parenthesis operator and then replacing the second X with the number operator. As follows:

/the (.\*)er they were, the \1er they will be/

The above will match: “*The bigger the were, the bigger they will be”* but not *The bigger they were, the faster they will be*

The following is another possibility:

/the (.\*)er they (.\*), the \1er we \2/

The above will match *“The faster they ran, the faster we ran”* but not *“The faster they ran, the faster we ate”*

Quick Python Summary

The basic code is the following. The *regexp.search(stringToSearch)* just produces a true or a false.

import re

regexp = re.compile("hello")

count = 0

file = open("textfile", 'r')

for line in file.readlines():

if regexp.search(line):

count = count + 1

file.close()

print(count)

*Raw Strings*

When entering ‘\\’ as a Python string, only the ‘\’ will be printed. So, if we wanted to have a regular expression to match ‘\ten’ , we need to supply ‘\\ten’ as a regular expression. But if we entered this as a Python string, we would end up with ‘\ten’ which means find a tab follwed by ‘en’. So, we use raw strings. An example is shown below:

>>> print('\\ten') #normal string

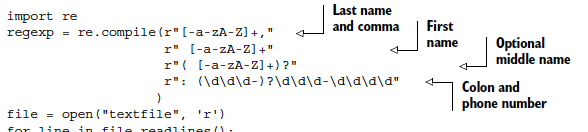
\ten

>>> print(r'\\ten') #raw string

\\ten

*Extracting Matched Text From Strings*

Here is an example of the pattern:



We refer to each component using the syntax *‘(?P<variableName> RegularExpresion)’* So, we originally had r"[-a-zA-Z]+," . Now, this becomes: r"(?P<last>[-a-zA-Z]+)," The entire code snippet is shown below:

import re

regexp = re.compile(r"(?P<last>[-a-zA-Z]+),"

r" (?P<first>[-a-zA-Z]+)"

r"( (?P<middle>([-a-zA-Z]+)))?"

r": (?P<phone>(\d\d\d-)?\d\d\d-\d\d\d\d)"

)

file = open("textfile", 'r') for line in file.readlines():

result = regexp.search(line)

if result == None:

print("Oops, I don't think this is a record") else:

lastname = result.group('last') #get the Last Name

firstname = result.group('first') #get the first Name

middlename = result.group('middle') #get the middle name

if middlename == None:

middlename = ""

phonenumber = result.group('phone')

*Substituting Text With Regular Expressions*

Here is simple example, the syntax is regexp.sub(“replacement text”, textToSearch):

>>> import re

>>> string = "if the the problem is textual, use the re module"

>>> pattern = r'the the'

>>> regexp = re.compile(pattern)

>>> regexp.sub("the", string)

'if the problem is textual, use the re module'

Here is another more complex example:

import re

strIntegers = "1 2 3 mark 4 5"

def fnIntMatchTo(objMatch):

print(objMatch.group('num') + ".0")

pattern = r"(?P<num>[0-9]+)"

regexp = re.compile(pattern)

regexp.sub(fnIntMatchTo, strIntegers)

The line: regexp.sub(fnIntMatchTo, strIntegers) results in fnIntMatchTo(objMatch) being called each time a match is found.