Pattern Matching and Regular Expressions

REVIEW OF REGEX SYMBOLS IN THESE NOTES:

- The ? matches zero or one of the preceding group.
- The * matches zero or more of the preceding group.
- The + matches one or more of the preceding group.
- The {n} matches exactly *n* of the preceding group.
- The {n,} matches *n* or more of the preceding group.
- The {,m} matches 0 to *m* of the preceding group.
- The {n,m} matches at least *n* and at most *m* of the preceding group.
- {n,m}? or *? or +? performs a nongreedy match of the preceding group.
- ^spam means the string must begin with *spam*.
- spam$ means the string must end with *spam*.
- The . matches any character, except newline characters.
- \d, \w, and \s match a digit, word, or space character, respectively.
- \D, \W, and \S match anything except a digit, word, or space character, respectively.
- [abc] matches any character between the brackets (such as *a*, *b*, or *c*).
- [^abc] matches any character that isn't between the brackets.

"You may be familiar with searching for text by pressing CTRL-F and typing in the words you're looking for. Regular expressions go one step further: They allow you to specify a pattern of text to search for."

*Regex is the shorthand way of writing regular expression

Regex objects

Import with:

```
>>> import re
```

“Passing a string value representing your regular expression to re.compile() returns a Regex pattern object (or simply, a Regex object).”

“\d\d\d-\d\d\d-\d\d\d\d is the regular expression for the correct phone number pattern.)”

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

Now the phoneNumRegex variable contains a Regex object.

Passing raw strings to re.compile()

“Remember that escape characters in Python use the backslash (\). The string value '\n' represents a single newline character, not a backslash followed by a lowercase n. You need to enter the escape character \\ to print a single backslash. So '\\n' is the string that represents a backslash followed by a lowercase n. However, by putting an r before the first quote of the string value, you can mark the string as a raw string, which does not escape characters.

Since regular expressions frequently use backslashes in them, it is convenient to pass raw strings to the re.compile() function instead of typing extra backslashes. Typing r'\d\d\d-\d\d\d-\d\d\d\d' is much easier than typing '\\d\\d\\d-\\d\\d\\d-\\d\\d\\d\\d'.”

Matching regex objects

“A Regex object’s search() method searches the string it is passed for any matches to the regex. The search() method will return None if the regex pattern is not found in the string. ”

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
```

```
>>> print('Phone number found: ' + mo.group())
Phone number found: 415-555-4242
```

This example might seem complicated at first, **but it is much shorter than the earlier isPhoneNumber.py program and does the same thing.**

“Writing mo.group() inside our print statement displays the whole match, 415-555-4242.”

Review of Regular Expression Matching

While there are several steps to using regular expressions in Python, each step is fairly simple:

```
1. Import the regex module with import re.
2. Create a Regex object with the re.compile() function. (Remember to
use a raw string.)
3. Pass the string you want to search into the Regex object's
search() method. This returns a Match object.
4. Call the Match object's group() method to return a string of the
actual matched text.
```

Grouping with parentheses

“Say you want to separate the area code from the rest of the phone number. Adding parentheses will create groups in the regex: (\d\d\d)-(\d\d\d-\d\d\d\d). Then you can use the group() match object method to grab the matching text from just one group.”

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
```

```
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
```

The **groups()** method returns all at once (hence the plurality of the term)

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

If your regular expression has parenthesis, you need to escape the () characters with a backslash like so:

```
>>> phoneNumRegex = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My phone number is (415) 555-4242.')
>>> mo.group(1)
'(415)'
>>> mo.group(2)
'555-4242'
```

Matching multiple groups with the pipe |

You can use the pipe to match one of many expressions. “For example, the regular expression r'Batman|Tina Fey' will match either 'Batman' or 'Tina Fey'.”

“When both Batman and Tina Fey occur in the searched string, the first occurrence of matching text will be returned as the Match object.”

```
>>> heroRegex = re.compile (r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman and Tina Fey.')
>>> mo1.group()
'Batman'

>>> mo2 = heroRegex.search('Tina Fey and Batman.')
>>> mo2.group()
'Tina Fey'
```

Note: you can find all matching occurrences with the **findall()** method that is discussed here

“You can also use the pipe to match one of several patterns as part of your regex. For example, say you wanted to match any of the strings 'Batman', 'Batmobile', 'Batcopter', and 'Batbat'. Since all these strings start with Bat, it would be nice if you could specify that prefix only once. This can be done with parentheses.”

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
>>> mo.group(1)
'mobile'
```

“The method call mo.group() returns the full matched text 'Batmobile', while mo.group(1) returns just the part of the matched text inside the first parentheses group, 'mobile'. By using the pipe character and grouping parentheses, you can specify several alternative patterns you would like your regex to match.

If you need to match an actual pipe character, escape it with a backslash, like \|.”
Optional matching with the question mark

“Sometimes there is a pattern that you want to match only optionally. **That is, the regex should find a match whether or not that bit of text is there**.”

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
```

```
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

“The (wo)? part of the regular expression means that the pattern wo is an optional group. The regex will match text that has zero instances or one instance of wo in it. This is why the regex matches both 'Batwoman' and 'Batman'.”

You can use this same approach to look for phone numbers that do or don’t have a specific area code like so:

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
>>> mo1 = phoneRegex.search('My number is 415-555-4242')
>>> mo1.group()
'415-555-4242'

>>> mo2 = phoneRegex.search('My number is 555-4242')
>>> mo2.group()
'555-4242'
```

Matching zero or more with the star *

* = “match zero or more”

The group before the store can occur any number of times in the text (including zero). See the previous Batman example but add:

```
>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

For 'Batman', the (wo)* part of the regex matches zero instances of wo in the string; for 'Batwoman', the (wo)* matches one instance of wo; and for 'Batwowowowoman', (wo)* matches four instances of wo.

Matching one or more with plus +

+ = “match one or more”

The group preceding the plus must occur at least once.

```
>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

“The regex Bat(wo)+man will not match the string 'The Adventures of Batman' because at least one wo is required by the plus sign.”

Matching specific repetitions with curly brackets

You can specify a certain number of repetitions or a range of repetitions using curly brackets. For example, the regex (Ha){3} will match the string 'HaHaHa', but it will not match 'HaHa', since the latter has only two repeats of the (Ha) group.

You can also specify a range by writing a minimum, a comma, and a maximum in between the curly brackets. For example, the regex (Ha){3,5} will match 'HaHaHa', 'HaHaHaHa', and 'HaHaHaHaHa'.

If you want to leave the number unbounded on either side, you can leave out the first or second number in a range. For example, (Ha){3,} will match three or more instances of the (Ha) group, while (Ha){,5} will match zero to five instances.

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
```

```
>>> mo1.group()
'HaHaHa'

>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

Greedy and nongreedy matching

“Python’s regular expressions are greedy by default, which means that in ambiguous situations they will match the longest string possible. The non-greedy version of the curly brackets, which matches the shortest string possible, has the closing curly bracket followed by a question mark.”

Greedy

```
>>> greedyHaRegex = re.compile(r'(Ha){3,5}')
>>> mo1 = greedyHaRegex.search('HaHaHaHaHa')
>>> mo1.group()
'HaHaHaHaHa'
```

Nongreedy

```
>>> nongreedyHaRegex = re.compile(r'(Ha){3,5}?')
>>> mo2 = nongreedyHaRegex.search('HaHaHaHaHa')
>>> mo2.group()
'HaHaHa'
```

Note that the question mark can have two meanings in regular expressions: declaring a nongreedy match or flagging an optional group. **These meanings are entirely unrelated.**

The Findall() method

While search() will return a Match object of the first matched text in the searched string, **the findall() method will return the strings of every match in the searched string.**

search()

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('Cell: 415-555-9999 Work:
212-555-0000')
>>> mo.group()
'415-555-9999'
```

findall()

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no
groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

If there are groups in the regular expression, then findall() will return a list of tuples. Each tuple represents a found match, and its items are the matched strings for each group in the regex

findall() in action:

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has
groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
[('415', '555', '9999'), ('212', '555', '0000')]
```

To summarize what the findall() method returns, remember the following:

1. When called on a regex with no groups, such as \d\d\d-\d\d\d-\d\d\d\d, the method findall() returns a list of string matches, such as ['415-555-9999', '212-555-0000'].

2. When called on a regex that has groups, such as (\d\d\d)-(\d\d\d)-(\d\ d\d\d), the method findall() returns a list of tuples of strings (one string for each group), such as [('415', '555', '9999'), ('212', '555', '0000')].

Character classes

In the earlier phone number regex example, you learned that \d could stand for any numeric digit. That is, \d is shorthand for the regular expression (0|1|2|3|4|5|6|7|8|9). There are many such shorthand character classes, as shown in Table 7-1.

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8
maids, 7
swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7
swans', '6
geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

“he regular expression \d+\s\w+ will match text that has one or more numeric digits (\d+), followed by a whitespace character (\s), followed by one or more letter/digit/underscore characters (\w+). The findall() method returns all matching strings of the regex pattern in a list.”

Making your own character classes

You can define your own character classes using square brackets. For example, the character class [aeiouAEIOU] will match any vowel, both lowercase and uppercase.

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('Robocop eats baby food. BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

Note that inside the square brackets, the normal regular expression symbols are not interpreted as such. **This means you do not need to escape the ., *, ?, or () characters with a preceding backslash.**

The caret and dollar sign characters

^ = indicates that that a match must occur at the beginning of the searched text
$ = indicates that that a match must occur at the end of the searched text

- These can be put together to indicate that that the entire string must match the regex

Carat in action:

```
>>> beginsWithHello = re.compile(r'^Hello')
>>> beginsWithHello.search('Hello world!')
<_sre.SRE_Match object; span=(0, 5), match='Hello'>
>>> beginsWithHello.search('He said hello.') == None
True
```

Dollar sign in action:

```
>>> endsWithNumber = re.compile(r'\d$')
>>> endsWithNumber.search('Your number is 42')
<_sre.SRE_Match object; span=(16, 17), match='2'>
>>> endsWithNumber.search('Your number is forty two.') == None
True
```

Both in action:

```
>>> wholeStringIsNum = re.compile(r'^\d+$')
>>> wholeStringIsNum.search('1234567890')
<_sre.SRE_Match object; span=(0, 10), match='1234567890'>
>>> wholeStringIsNum.search('12345xyz67890') == None
True
>>> wholeStringIsNum.search('12 34567890') == None
True
```

Remember: *carrots cost dollars* (carats indicate beginning, dollar signs indicate ending)

The Wildcard character

. = “wildcard”. It will match any character except for a newline.

```
>>> atRegex = re.compile(r'.at')
```

```
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

The dot will only match one character, hence the “lat”
*Remember, you can always match the actual dot with backslash \

Matching everything with dot-star
.* matches everything in a string

```
>>> nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')
>>> mo = nameRegex.search('First Name: Al Last Name: Sweigart')
>>> mo.group(1)
'Al'
>>> mo.group(2)
'Sweigart'
```

The dot star is **greedy**, meaning it will always match as much text as possible.

-To match any and all text in a nongreedy fashion, use the dot, star, and question mark (.*?). Like with curly brackets, the question mark tells Python to match in a nongreedy way.

Nongreedy

```
>>> nongreedyRegex = re.compile(r'<.*?>')
>>> mo = nongreedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man>'
```

Greedy

```
>>> greedyRegex = re.compile(r'<.*>')
>>> mo = greedyRegex.search('<To serve man> for dinner.>')
>>> mo.group()
'<To serve man> for dinner.>'
```

“Both regexes roughly translate to “Match an opening angle bracket, followed by anything, followed by a closing angle bracket.” But the string '<To serve man> for dinner.>' has two possible matches for the closing angle bracket. In the nongreedy version of the regex, Python matches the shortest possible string: '<To serve man>'. In

the greedy version, Python matches the longest possible string: '<To serve man> for dinner.>'."

Matching newlines with the Dot character

"The dot-star will match everything except a newline. By passing re.DOTALL as the second argument to re.compile(), you can make the dot character match all characters, including the newline character."

```
>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the
innocent.
\nUphold the law.').group()
'Serve the public trust.'
```

```
>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the
innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

The regex noNewlineRegex, which did not have re.DOTALL passed to the re.compile() call that created it, will match everything only up to the first newline character, whereas newlineRegex, which did have re.DOTALL passed to re.compile(), matches everything. This is why the newlineRegex.search() call matches the full string, including its newline characters.

Case-insensitive matching

“Normally, regular expressions match text with the exact casing you specify. For example, the following regexes match completely different strings:”

```
>>> regex1 = re.compile('Robocop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('robOcop')
>>> regex4 = re.compile('RobocOp')
```

“But sometimes you care only about matching the letters without worrying whether they’re uppercase or lowercase. To make your regex case-insensitive, you can pass re.IGNORECASE or re.I as a second argument to re.compile().”

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('Robocop is part man, part machine, all
cop.').group()
'Robocop'
```

```
>>> robocop.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'
```

```
>>> robocop.search('Al, why does your programming book talk about
robocop so much?').group()
'robocop'
```

Substituting Strings with the sub() Method

Regular expressions can not only find text patterns but **can also substitute new text in place of those patterns**. The sub() method for Regex objects is passed two arguments. The first argument is a string to replace any matches. The second is the string for the regular expression. **The sub() method returns a string with the substitutions applied.**

```
>>> namesRegex = re.compile(r'Agent \w+')
>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents
to Agent Bob.')
'CENSORED gave the secret documents to CENSORED.'
```

“Sometimes you may need to use the matched text itself as part of the substitution. In the first argument to sub(), you can type \1, \2, \3, and so on, to mean “Enter the text of group 1, 2, 3, and so on, in the substitution.”

For example, say you want to censor the names of the secret agents by showing just the first letters of their names. To do this, you could use the regex Agent (\w)\w* and pass r'\1****' as the first argument to sub(). The \1 in that string will be replaced by whatever text was matched by group 1—that is, the (\w) group of the regular expression.”

```
>>> agentNamesRegex = re.compile(r'Agent (\w)\w*')
>>> agentNamesRegex.sub(r'\1****', 'Agent Alice told Agent Carol that
Agent
Eve knew Agent Bob was a double agent.')
A**** told C**** that E**** knew B**** was a double agent.'
```

Managing complex regexes

“Regular expressions are fine if the text pattern you need to match is simple. But matching complicated text patterns might require long, convoluted regular expressions. **You can mitigate this by telling the re.compile() function to ignore whitespace and comments inside the regular expression string. This “verbose mode” can be enabled by passing the variable re.VERBOSE as the second argument to re.compile().”**

You can take this:

```
phoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4}
(\s*(ext|x|ext.)\s*\d{2,5})?)')
```

And turn it into:

```
phoneRegex = re.compile(r'''(
    (\d{3}|\(\d{3}\))?            # area code
    (\s|-|\.)?                    # separator
    \d{3}                         # first 3 digits
    (\s|-|\.)                     # separator
    \d{4}                         # last 4 digits
    (\s*(ext|x|ext.)\s*\d{2,5})?  # extension
    )''', re.VERBOSE)
```

Note how the previous example uses the triple-quote syntax (''') to create a multiline string so that you can spread the regular expression definition over many lines, making it much more legible.

The comment rules inside the regular expression string are the same as regular Python code: The # symbol and everything after it to the end of the line are ignored. Also, the extra spaces inside the multiline string for the regular expression are not considered part of the text pattern to be matched. This lets you organize the regular expression so it’s easier to read.

Combining re.IGNORECASE, re.DOTALL, and re.VERBOSE

“What if you want to use re.VERBOSE to write comments in your regular expression but also want to use re.IGNORECASE to ignore capitalization? Unfortunately, the re.compile() function takes only a single value as its second argument. You can get around this limitation by combining the re.IGNORECASE, re.DOTALL, and re.VERBOSE variables using the pipe character (|), which in this context is known as the bitwise or operator.”

“So if you want a regular expression that’s case-insensitive and includes newlines to match the dot character, you would form your re.compile() call like this:”

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL)
```

All three options for the second argument will look like this:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL |
re.VERBOSE)
```

“This syntax is a little old-fashioned and originates from early versions of Python. The details of the bitwise operators are beyond the scope of this book, but check out the resources at http://nostarch.com/automatestuff/ for more information. You can also pass other options for the second argument; they’re uncommon, but you can read more about them in the resources, too.”