PY4E (https://www.py4e.com)

Chapter 11: Regex

Regular expressions

So far we have been reading through files, looking for patterns and extracting various bits of lines that we find interesting. We have been using string methods like `split` and `find` and using lists and string slicing to extract portions of the lines.

This task of searching and extracting is so common that Python has a very powerful module called *regular expressions* that handles many of these tasks quite elegantly. The reason we have not introduced regular expressions earlier in the book is because while they are very powerful, they are a little complicated and their syntax takes some getting used to.

Regular expressions are almost their own little programming language for searching and parsing strings. As a matter of fact, entire books have been written on the topic of regular expressions. In this chapter, we will only cover the basics of regular expressions. For more detail on regular expressions, see:

https://en.wikipedia.org/wiki/Regular_expression (https://en.wikipedia.org/wiki/Regular_expression)

https://docs.python.org/library/re.html (https://docs.python.org/library/re.html)

The regular expression module `re` must be imported into your program before you can use it. The simplest use of the regular expression module is the `search()` function. The following program demonstrates a trivial use of the search function.

```
# Search for lines that contain 'From'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line):
        print(line)

# Code: http://www.py4e.com/code3/re01.py
```

We open the file, loop through each line, and use the regular expression `search()` to only print out lines that contain the string “From:”. This program does not use the real power of regular expressions, since we could have just as easily used `line.find()` to accomplish the same result.

Select Language ▼

The power of the regular expressions comes when we add special characters to the search string that allow us to more precisely control which lines match the string. Adding these special characters to our regular expression allow us to do sophisticated matching and extraction while writing very little code.

For example, the caret character is used in regular expressions to match “the beginning” of a line. We could change our program to only match lines where “From:” was at the beginning of the line as follows:

```
# Search for lines that start with 'From'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line):
        print(line)

# Code: http://www.py4e.com/code3/re02.py
```

Now we will only match lines that *start with* the string “From:”. This is still a very simple example that we could have done equivalently with the `startswith()` method from the string module. But it serves to introduce the notion that regular expressions contain special action characters that give us more control as to what will match the regular expression.

Character matching in regular expressions

There are a number of other special characters that let us build even more powerful regular expressions. The most commonly used special character is the period or full stop, which matches any character.

In the following example, the regular expression `F..m:` would match any of the strings “From:”, “Fxxm:”, “F12m:”, or “F!@m:” since the period characters in the regular expression match any character.

```
# Search for lines that start with 'F', followed by
# 2 characters, followed by 'm:'
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^F..m:', line):
        print(line)

# Code: http://www.py4e.com/code3/re03.py
```

This is particularly powerful when combined with the ability to indicate that a character can be repeated any number of times using the `*` or `+` characters in your regular expression. These special characters mean that instead of matching a single character in the search string, they match zero-or-more characters (in the case of the

asterisk) or one-or-more of the characters (in the case of the plus sign).

We can further narrow down the lines that we match using a repeated *wild card* character in the following example:

```
# Search for lines that start with From and have an at sign
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:.+@', line):
        print(line)

# Code: http://www.py4e.com/code3/re04.py
```

The search string `^From:.+@` will successfully match lines that start with “From:”, followed by one or more characters (`.+`), followed by an at-sign. So this will match the following line:

```
From: stephen.marquard@uct.ac.za
```

You can think of the `.+` wildcard as expanding to match all the characters between the colon character and the at-sign.

```
From:.+@
```

It is good to think of the plus and asterisk characters as “pushy”. For example, the following string would match the last at-sign in the string as the `.+` pushes outwards, as shown below:

```
From: stephen.marquard@uct.ac.za, csev@umich.edu, and cwen @iupui.edu
```

It is possible to tell an asterisk or plus sign not to be so “greedy” by adding another character. See the detailed documentation for information on turning off the greedy behavior.

Extracting data using regular expressions

If we want to extract data from a string in Python we can use the `findall()` method to extract all of the substrings which match a regular expression. Let’s use the example of wanting to extract anything that looks like an email address from any line regardless of format. For example, we want to pull the email addresses from each of the following lines:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
          for <source@collab.sakaiproject.org>;
Received: (from apache@localhost)
Author: stephen.marquard@uct.ac.za
```

We don't want to write code for each of the types of lines, splitting and slicing differently for each line. This following program uses `findall()` to find the lines with email addresses in them and extract one or more addresses from each of those lines.

```
import re
s = 'A message from csev@umich.edu to cwen@iupui.edu about meeting @2PM'
lst = re.findall('\S+@\S+', s)
print(lst)

# Code: http://www.py4e.com/code3/re05.py
```

The `findall()` method searches the string in the second argument and returns a list of all of the strings that look like email addresses. We are using a two-character sequence that matches a non-whitespace character (`\S`).

The output of the program would be:

```
['csev@umich.edu', 'cwen@iupui.edu']
```

Translating the regular expression, we are looking for substrings that have at least one non-whitespace character, followed by an at-sign, followed by at least one more non-whitespace character. The `\S+` matches as many non-whitespace characters as possible.

The regular expression would match twice (csev@umich.edu and cwen@iupui.edu), but it would not match the string "@2PM" because there are no non-blank characters *before* the at-sign. We can use this regular expression in a program to read all the lines in a file and print out anything that looks like an email address as follows:

```
# Search for lines that have an at sign between characters
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('\S+@\S+', line)
    if len(x) > 0:
        print(x)

# Code: http://www.py4e.com/code3/re06.py
```

We read each line and then extract all the substrings that match our regular expression. Since `findall()` returns a list, we simply check if the number of elements in our returned list is more than zero to print only lines where we found at least one substring that looks like an email address.

If we run the program on *mbox-short.txt* we get the following output:

```
['<source@collab.sakaiproject.org>;']
['<source@collab.sakaiproject.org>;']
['apache@localhost)']
['source@collab.sakaiproject.org;']
['cwen@iupui.edu']
['source@collab.sakaiproject.org']
['cwen@iupui.edu']
['cwen@iupui.edu']
['wagnermr@iupui.edu']
```

Some of our email addresses have incorrect characters like "<" or ";" at the beginning or end. Let's declare that we are only interested in the portion of the string that starts and ends with a letter or a number.

To do this, we use another feature of regular expressions. Square brackets are used to indicate a set of multiple acceptable characters we are willing to consider matching. In a sense, the `\S` is asking to match the set of "non-whitespace characters". Now we will be a little more explicit in terms of the characters we will match.

Here is our new regular expression:

```
[a-zA-Z0-9]\S*@\S*[a-zA-Z]
```

This is getting a little complicated and you can begin to see why regular expressions are their own little language unto themselves. Translating this regular expression, we are looking for substrings that start with a *single* lowercase letter, uppercase letter, or number "[a-zA-Z0-9]", followed by zero or more non-blank characters (`\S*`), followed by an at-sign, followed by zero or more non-blank characters (`\S*`), followed by an uppercase or lowercase letter. Note that we switched from `+` to `*` to indicate zero or more non-blank characters since `[a-zA-Z0-9]` is already one non-blank character. Remember that the `*` or `+` applies to the single character immediately to the left of the plus or asterisk.

If we use this expression in our program, our data is much cleaner:

```
# Search for lines that have an at sign between characters
# The characters must be a letter or number
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('[a-zA-Z0-9]\S*@\S*[a-zA-Z]', line)
    if len(x) > 0:
        print(x)

# Code: http://www.py4e.com/code3/re07.py
```

```
['wagnerm@iupui.edu']
['cwen@iupui.edu']
['postmaster@collab.sakaiproject.org']
['200801032122.m03LMFo4005148@nakamura.uits.iupui.edu']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['source@collab.sakaiproject.org']
['apache@localhost']
```

Notice that on the `source@collab.sakaiproject.org` lines, our regular expression eliminated two letters at the end of the string (">;"). This is because when we append `[a-zA-Z]` to the end of our regular expression, we are demanding that whatever string the regular expression parser finds must end with a letter. So when it sees the ">" at the end of "sakaiproject.org>;" it simply stops at the last "matching" letter it found (i.e., the "g" was the last good match).

Also note that the output of the program is a Python list that has a string as the single element in the list.

Combining searching and extracting

If we want to find numbers on lines that start with the string "X-" such as:

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
```

we don't just want any floating-point numbers from any lines. We only want to extract numbers from lines that have the above syntax.

We can construct the following regular expression to select the lines:

```
^X-.*: [0-9.]+
```

Translating this, we are saying, we want lines that start with `X-`, followed by zero or more characters (`.*`), followed by a colon (`:`) and then a space. After the space we are looking for one or more characters that are either a digit (0-9) or a period `[0-9.]+`. Note that inside the square brackets, the period matches an actual period (i.e., it is not a wildcard between the square brackets).

This is a very tight expression that will pretty much match only the lines we are interested in as follows:

```
# Search for lines that start with 'X' followed by any non
# whitespace characters and ':'
# followed by a space and any number.
# The number can include a decimal.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line):
        print(line)

# Code: http://www.py4e.com/code3/re10.py
```

When we run the program, we see the data nicely filtered to show only the lines we are looking for.

```
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000
X-DSPAM-Confidence: 0.6178
X-DSPAM-Probability: 0.0000
...
```

But now we have to solve the problem of extracting the numbers. While it would be simple enough to use `split`, we can use another feature of regular expressions to both search and parse the line at the same time.

Parentheses are another special character in regular expressions. When you add parentheses to a regular expression, they are ignored when matching the string. But when you are using `findall()`, parentheses indicate that while you want the whole expression to match, you only are interested in extracting a portion of the substring that matches the regular expression.

So we make the following change to our program:

```
# Search for lines that start with 'X' followed by any
# non whitespace characters and ':' followed by a space
# and any number. The number can include a decimal.
# Then print the number if it is greater than zero.
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^X\S*: ([0-9.]+)', line)
    if len(x) > 0:
        print(x)

# Code: http://www.py4e.com/code3/re11.py
```

Instead of calling `search()`, we add parentheses around the part of the regular expression that represents the floating-point number to indicate we only want `findall()` to give us back the floating-point number portion of the matching string.

The output from this program is as follows:

```
['0.8475']
['0.0000']
['0.6178']
['0.0000']
['0.6961']
['0.0000']
...
```

The numbers are still in a list and need to be converted from strings to floating point, but we have used the power of regular expressions to both search and extract the information we found interesting.

As another example of this technique, if you look at the file there are a number of lines of the form:

```
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

If we wanted to extract all of the revision numbers (the integer number at the end of these lines) using the same technique as above, we could write the following program:

```
# Search for lines that start with 'Details: rev='
# followed by numbers
# Then print the number if one is found
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^Details:.*rev=([0-9]+)', line)
    if len(x) > 0:
        print(x)

# Code: http://www.py4e.com/code3/re12.py
```

Translating our regular expression, we are looking for lines that start with `Details:`, followed by any number of characters (`.*`), followed by `rev=`, and then by one or more digits. We want to find lines that match the entire expression but we only want to extract the integer number at the end of the line, so we surround `[0-9]+` with parentheses.

When we run the program, we get the following output:

```
['39772']
['39771']
['39770']
['39769']
...
```

Remember that the `[0-9]+` is "greedy" and it tries to make as large a string of digits as possible before extracting those digits. This "greedy" behavior is why we get all five digits for each number. The regular expression module expands in both directions until it encounters a non-digit, or the beginning or the end of a line.

Now we can use regular expressions to redo an exercise from earlier in the book where we were interested in the time of day of each mail message. We looked for lines of the form:

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

and wanted to extract the hour of the day for each line. Previously we did this with two calls to `split`. First the line was split into words and then we pulled out the fifth word and split it again on the colon character to pull out the two characters we were interested in.

While this worked, it actually results in pretty brittle code that is assuming the lines are nicely formatted. If you were to add enough error checking (or a big try/except block) to insure that your program never failed when presented with incorrectly formatted lines, the code would balloon to 10-15 lines of code that was pretty hard to read.

We can do this in a far simpler way with the following regular expression:

```
^From .* [0-9][0-9]:
```

The translation of this regular expression is that we are looking for lines that start with `From` (note the space), followed by any number of characters (`.*`), followed by a space, followed by two digits `[0-9][0-9]`, followed by a colon character. This is the definition of the kinds of lines we are looking for.

In order to pull out only the hour using `findall()`, we add parentheses around the two digits as follows:

```
^From .* ([0-9][0-9]):
```

This results in the following program:

```
# Search for lines that start with From and a character
# followed by a two digit number between 00 and 99 followed by ':'
# Then print the number if one is found
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    x = re.findall('^From .* ([0-9][0-9]):', line)
    if len(x) > 0: print(x)

# Code: http://www.py4e.com/code3/re13.py
```

When the program runs, it produces the following output:

```
['09']
['18']
['16']
['15']
...
```

Escape character

Since we use special characters in regular expressions to match the beginning or end of a line or specify wild cards, we need a way to indicate that these characters are “normal” and we want to match the actual character such as a dollar sign or caret.

We can indicate that we want to simply match a character by prefixing that character with a backslash. For example, we can find money amounts with the following regular expression.

```
import re
x = 'We just received $10.00 for cookies.'
y = re.findall('\$[0-9.]+',x)
```

Since we prefix the dollar sign with a backslash, it actually matches the dollar sign in the input string instead of matching the “end of line”, and the rest of the regular expression matches one or more digits or the period character. *Note:* Inside square brackets, characters are not “special”. So when we say `[0-9.]`, it really means digits or a period. Outside of square brackets, a period is the “wild-card” character and matches any character. Inside square brackets, the period is a period.

Summary

While this only scratched the surface of regular expressions, we have learned a bit about the language of regular expressions. They are search strings with special characters in them that communicate your wishes to the regular expression system as to what defines "matching" and what is extracted from the matched strings. Here are some of those special characters and character sequences:

`^` Matches the beginning of the line.

`$` Matches the end of the line.

`.` Matches any character (a wildcard).

`\s` Matches a whitespace character.

`\S` Matches a non-whitespace character (opposite of \s).

`*` Applies to the immediately preceding character(s) and indicates to match zero or more times.

`*?` Applies to the immediately preceding character(s) and indicates to match zero or more times in "non-greedy mode".

`+` Applies to the immediately preceding character(s) and indicates to match one or more times.

`+?` Applies to the immediately preceding character(s) and indicates to match one or more times in "non-greedy mode".

`?` Applies to the immediately preceding character(s) and indicates to match zero or one time.

`??` Applies to the immediately preceding character(s) and indicates to match zero or one time in "non-greedy mode".

`[aeiou]` Matches a single character as long as that character is in the specified set. In this example, it would match "a", "e", "i", "o", or "u", but no other characters.

`[a-z0-9]` You can specify ranges of characters using the minus sign. This example is a single character that must be a lowercase letter or a digit.

`[^A-Za-z]` When the first character in the set notation is a caret, it inverts the logic. This example matches a single character that is anything *other than* an uppercase or lowercase letter.

`( )` When parentheses are added to a regular expression, they are ignored for the purpose of matching, but allow you to extract a particular subset of the matched string rather than the whole string when using `findall()`.

`\b` Matches the empty string, but only at the start or end of a word.

`\B` Matches the empty string, but not at the start or end of a word.

`\d` Matches any decimal digit; equivalent to the set [0-9].

`\D` Matches any non-digit character; equivalent to the set [^0-9].

PY4E (https://www.py4e.com)

Bonus section for Unix / Linux users

Support for searching files using regular expressions was built into the Unix operating system since the 1960s and it is available in nearly all programming languages in one form or another.

As a matter of fact, there is a command-line program built into Unix called *grep* (Generalized Regular Expression Parser) that does pretty much the same as the `search()` examples in this chapter. So if you have a Macintosh or Linux system, you can try the following commands in your command-line window.

```
$ grep '^From:' mbox-short.txt
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
```

This tells `grep` to show you lines that start with the string "From:" in the file *mbox-short.txt*. If you experiment with the `grep` command a bit and read the documentation for `grep`, you will find some subtle differences between the regular expression support in Python and the regular expression support in `grep`. As an example, `grep` does not support the non-blank character `\S` so you will need to use the slightly more complex set notation `[^ ]`, which simply means match a character that is anything other than a space.

Debugging

Python has some simple and rudimentary built-in documentation that can be quite helpful if you need a quick refresher to trigger your memory about the exact name of a particular method. This documentation can be viewed in the Python interpreter in interactive mode.

You can bring up an interactive help system using `help()`.

```
>>> help()

help> modules
```

If you know what module you want to use, you can use the `dir()` command to find the methods in the module as follows:

```
>>> import re
>>> dir(re)
[.. 'compile', 'copy_reg', 'error', 'escape', 'findall',
'finditer', 'match', 'purge', 'search', 'split', 'sre_compile',
'sre_parse', 'sub', 'subn', 'sys', 'template']
```

You can also get a small amount of documentation on a particular method using the dir command.

```
>>> help (re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.
>>>
```

The built-in documentation is not very extensive, but it can be helpful when you are in a hurry or don't have access to a web browser or search engine.

Glossary

brittle code
Code that works when the input data is in a particular format but is prone to breakage if there is some deviation from the correct format. We call this "brittle code" because it is easily broken.
greedy matching
The notion that the `+` and `*` characters in a regular expression expand outward to match the largest possible string.
grep
A command available in most Unix systems that searches through text files looking for lines that match regular expressions. The command name stands for "Generalized Regular Expression Parser".
regular expression
A language for expressing more complex search strings. A regular expression may contain special characters that indicate that a search only matches at the beginning or end of a line or many other similar capabilities.
wild card
A special character that matches any character. In regular expressions the wild-card character is the period.

Exercises

Exercise 1: Write a simple program to simulate the operation of the `grep` command on Unix. Ask the user to enter a regular expression and count the number of lines that matched the regular expression:

```
$ python grep.py
Enter a regular expression: ^Author
mbox.txt had 1798 lines that matched ^Author

$ python grep.py
Enter a regular expression: ^X-
mbox.txt had 14368 lines that matched ^X-

$ python grep.py
Enter a regular expression: java$
mbox.txt had 4175 lines that matched java$
```

Exercise 2: Write a program to look for lines of the form:

```
New Revision: 39772
```

Extract the number from each of the lines using a regular expression and the `findall()` method. Compute the average of the numbers and print out the average as an integer.

```
Enter file:mbox.txt
38549

Enter file:mbox-short.txt
39756
```

If you find a mistake in this book, feel free to send me a fix using Github.