Python – S10

Contents

Regular Expressions

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 library called regular expressions that handles many of these tasks quite elegantly.
- Regular expressions [a little difficult and complicated]are almost their own little programming language for searching and parsing strings.
- Here, we will only cover the basics of regular expressions.

- For more detail on regular expressions, see:
- http://en.wikipedia.org/wiki/Regular_expression
- http://docs.python.org/library/re.html
- http://www.regexpal.com

Regular Expressions

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

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

- 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.
- rstrip([chars]) – Return a copy of the string with trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or None, the *chars* argument defaults to removing whitespace.

Regular Expressions

- The power of the regular expressions comes when we add to 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 application to only match lines where "From:" was at the beginning of the line as follows:

```
import re
hand = open('mbox-short.txt')
for line in hand:
        if re.search('ˆFrom:', line) :
        print (line)
```

- Now we will only match lines that *start with* the string "From:".

Regular Expressions

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 character 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.

```
import re
hand = open('mbox-short.txt')
for line in hand:
    if re.search('ˆF..m:', line) :
        print (line)
```

Regular Expressions

Character matching in regular expressions

- 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 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:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    if re.search('ˆFrom:.+@', line) :
        print (line)
```

Regular Expressions

Character matching in regular expressions

•The search string “ˆFrom:.+@” 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”.

Regular Expressions

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 e-mail address from any line regardless of format.
- For example, we want to pull the e-mail 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
```

Regular Expressions

Extracting Data using Regular expressions

- 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 e-mail addresses in them and extract one or more addresses from each of those lines.

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

- The findall() method searches the string in the second argument and returns a list of all of the strings that look like e-mail 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']
```

Regular Expressions

Extracting Data using Regular expressions

- 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-white space characters.
- Also, the “\S+” matches as many non-whitespace characters as possible (this is called “greedy” matching in regular expressions).
- 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 e-mail address as follows:

```
import re
hand = open('mbox-short.txt')
for line in hand:
    x = re.findall('\S+@\S+', line)
    if len(x) > 0 :
        print (x)
```

Regular Expressions

Extracting Data using Regular expressions

- 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 e-mail address.
- If we run the program on mbox.txt we get the following output:

```
['wagnermr@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)']
['source@collab.sakaiproject.org;']
```

Regular Expressions

Extracting Data using Regular expressions

- We notice that Some of our E-mail 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]
```

Regular Expressions

Extracting Data using Regular expressions

- 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, upper case 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 upper or lower case 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:

Regular Expressions

Extracting Data using Regular expressions

```
import re
hand = open('mbox-short.txt')
for line in hand:
    x = re.findall('[a-zA-Z0-9]\S*@\S*[a-zA-Z]', line)
    if len(x) > 0 :
        print (x)
...

['wagnermr@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']
```

Regular Expressions

Extracting Data using Regular expressions

- 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, it must end with a letter.
- So when it sees the “>” after “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.

Regular Expressions

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 to extract numbers from lines that have the above syntax.
- We can construct the following regular expression to select the lines:

ˆX-.*: [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 in between the square braces, the period matches an actual period (i.e. it is not a wildcard between the square brackets).

Regular Expressions

<u>Combining Searching and Extracting</u>

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

```
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^X\S*: [0-9.]+', line) :
        print (line)
```

- 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
```

Regular Expressions

Combining Searching and Extracting

- But now we have to solve the problem of extracting the numbers using split.
- 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:

Regular Expressions

Combining Searching and Extracting

```
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)
```

- 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'] ..
```

Regular Expressions

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)
```

Regular Expressions

Escape Character

- 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:* In between 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. In between square brackets, the period is a period.

- We will not discuss any more details. Please refer to documentation for further information/examples.
- You can also refer to the book : Introducing Regular Expressions – Michael Fitzgerald [O’Reilly]

Regular Expressions

Cheat Sheet

Character classes

. any character except newline

\w \d \s word, digit, whitespace

\W \D \S not word, digit, whitespace

[abc] any of a, b, or c

[^abc] not a, b, or c

[a-g] character between a & g

Anchors

^abc$ start / end of the string

\b word boundary

Escaped characters

\. * \\ escaped special characters

\t \n \r tab, linefeed, carriage return

\u00A9 unicode escaped ©

Regular Expressions

Cheat Sheet

Groups & Lookaround

(abc) capture group

\1 backreference to group #1

(?:abc) non-capturing group

(?=abc) positive lookahead

(?!abc) negative lookahead

Quantifiers & Alternation

a* a+ a? 0 or more, 1 or more, 0 or 1

a{5} a{2,} exactly five, two or more

a{1,3} between one & three

a+? a{2,}? match as few as possible

ab|cd match ab or cd

Regular Expressions in Python

RegEX

- *Regular expressions* are a tool for matching text by looking for a pattern (rather than looking for a text string) in an easy and straightforward manner.
- For example, you could check for the presence of an exact text string within another text string simply by using the Python in keyword, as shown here:

```
>>> haystack = 'My phone number is 213-867-5309.'
>>> '213-867-5309' in haystack
True
```

- Sometimes, however, you do not have the exact text you want to match.
- For example, what if you want to know whether *any* valid phone number is present in a string? To take that one step further, what if you want to know whether any valid phone number is present in the string, *and* also want to know what that phone number is?

RegEx

- This is where regular expressions are useful.
- Their purpose is to specify a pattern of text to identify within a bigger text string.
- Regular expressions can identify the presence or absence of text matching the pattern, and also split a pattern into one or more subpatterns, delivering the specific text within each.

- We will learn now a little more about Regular Expressions (RegEx) and the ‘re’ module in Python

Why use RegEx ?

- You use regular expressions for two common reasons.

- The first reason is data mining—that is, when you want to find a pile of text (matching a given pattern) in a bigger pile of text.
- It is very common to need to identify text that looks like a given type of information (for example, an e-mail address, a URL, a phone number, or the like).

- The second reason is validation.
- You can use regular expressions to establish that you got the data that you expected.
- It is generally wise to consider "outside" data to be untrustworthy, especially data from users.
- Regular expressions can help determine whether or not untested data is valid.

Why use RegEx ?

- The corollary to this is that regular expressions are valuable tools for coercing data into a consistent format.
- For example, a phone number can be written in multiple valid ways, and if you are asking for user input, you likely want to accept all of them.
- However, you really only want to store the actual digits of the phone number, which can then be consistently formatted on display.

- In addition to being useful for validation, regular expressions are useful for this kind of data coercion.

RegEx in Python

- The Python standard library provides the ‘re’ module for using regular expressions.
- The primary function that the re module provides is search.
- Its purpose is to take a regular expression (the needle) and a string (the haystack), and return the first match found. If no match is found at all, re.search returns None.
- Consider re.search in action with the simplest regular expression possible, which is a simple alphanumeric string.

```
>>> import re
>>> re.search(r'fox', 'The quick brown fox jumped...')
<_sre.SRE_Match object; span=(16, 19), match='fox'>
```

- The regular expression parser’s job here is quite simple. It finds the word fox within the string, and returns a match object.

Note : *The* re *module also provides a function called* match *that appears to be very similar to* search*. It has one important difference: it only searches for a match that starts at the beginning of the string. It is easy (and common) to use re.match by mistake when you actually want to find something anywhere in a string.*

RegEx in Python

- Observant readers may note that the regular expression was specified slightly differently: r'fox'.
- The r character that precedes the string stands for "raw" (no, it does not stand for "regex").
- The difference between a raw string and a regular string is simply that raw strings do not interpret the \ character as an escape character.
- This means that, for example, it is not possible to escape a quote character to avoid concluding your string.
- However, raw strings are particularly useful for regular expressions because the regular expression engine itself needs the \ character for its own escaping at times.
- Therefore, using raw strings for regular expressions is very common and very useful.
- In fact, it is so common that some syntax-highlighting engines will actually provide regular-expression syntax highlighting within raw strings.

RegEx in Python

- Match objects have several methods to tell you things about the match.
- The group method is arguably the most important.
- It returns a string with the text of the match, as shown here:

```
>>> match = re.search(r'fox', 'The quick brown fox jumped...')
>>> match.group()
'fox'
```

- You may be curious why this method is named group.
- This is because regular expressions can be split into multiple subgroups that call out just a subsection of the match.
- You learn more about this shortly.
- Match objects have several other methods.
- The start method provides the index in the original string where the match began, and the end method provides the index in the original string where the match ended.

Finding more than one Match

- A limitation of re.search is that it only returns at most one match, in the form of a match object (discussed in more detail shortly).
- If multiple matches exist within the string, re.search will only return the first one.
- Often, this is exactly what you want.
- However, sometimes you want multiple matches if multiple matches exist.
- The re module provides two functions for this purpose: findall and finditer.
- Both of these methods return all non-overlapping matches, including empty matches.
- The re.findall method returns a list, and re.finditer returns a generator.
- However, there is a key difference here.
- These methods do not actually return a match object. Instead, they return simply the match itself, either as a string or a tuple, depending on the content of the regular expression.

Finding more than one Match

- Consider an example of findall:

```
>>> import re
>>> re.findall(r'o', 'The quick brown fox jumped...')
['o', 'o']
```

- In this case, it returns a list with two o characters, because the o character appears twice in the string.

Basic RegEx

- The simplest regular expression is one that contains plain alphanumeric characters—and nothing else.
- This is actually easy to overlook.
- Many regular expressions use direct text matching.
- The string Python is a valid regular expression.
- It matches that word, and nothing else. Regular expressions, by default, are also case-sensitive, so it will not match python or PYTHON.

```
>>> re.search(r'Python', 'python')
>>> re.search(r'Python', 'PYTHON')
```

- It will, however, match the word in a larger block of text. It will match the word in Python 3, or This is Python code, or the like, as shown here:

Basic RegEx

- There is also a match() function - The way the match() method works is that it will only find matches if they occur at the start of the string being searched.

```
>>> re.match(r'dog', 'dog cat dog')
<_sre.SRE_Match object at 0xb743e720<
>>> match = re.match(r'dog', 'dog cat dog')
>>> match.group(0)
'dog’
```

But, if we call match() on the same string, looking for the pattern ‘cat’, we won’t:

```
>>> re.match(r'cat', 'dog cat dog')
>>>
```

Basic RegEx

- The search() method is similar to match(), but search() doesn’t restrict us to only finding matches at the beginning of the string, so searching for ‘cat’ in our example string finds a match:

```
search(r'cat', 'dog cat dog')
>>> match.group(0)
'cat'
```

- The search() method, however, stops looking after it finds a match, so search()-ing for ‘dog’ in our example string only finds the first occurrence:

```
>>> match = re.search(r'dog', 'dog cat dog')
>>> match.group(0)
'dog'
```

Basic RegEx

```
>>> re.search(r'Python', 'Python 3')
<_sre.SRE_Match object; span=(0, 6), match='Python'>
>>> re.search(r'Python', 'This is Python code.')
<_sre.SRE_Match object; span=(8, 14), match='Python'>
```

- Of course, there is essentially no value in using regular expressions just to match plaintext regular expressions.
- After all, it would be trivially easy to use the in operator to test for the presence of a string within another string, and str.index is more than up to the task of telling you where in a larger string a substring occurs.
- The power of regular expressions lies in their capability to specify patterns of text to be matched.

Character Classes

- Character classes enable you to specify that a single character should match one of a set of possible characters, rather than just a single character.
- You can denote a character class by using square brackets and listing the possible characters within the brackets.
- For example, consider a regular expression that should match either Python or python: [Pp]ython.
- What is happening here? The first token in the regular expression is actually a character class with two options: P and p.
- Either character will match, but nothing else.
- The remaining five characters are just literal characters.
- What does the following regular expression match?

```
>>> re.search(r'[Pp]ython', 'Python 3')
<_sre.SRE_Match object; span=(0, 6), match='Python'>
>>> re.search(r'[Pp]ython', 'python 3')
<_sre.SRE_Match object; span=(0, 6), match='python'>
```

Character Classes

- This regular expression matches the word Python in the string Python 3 and the word python in the string python 3.
- It does not make the entire word case-insensitive, though.
- It does not match the word in all caps, for example.

```
>>> re.search(r'[Pp]ython', 'PYTHON 3')
>>>
```

- Another use for this kind of character class is for words with multiple spellings.
- The regular expression gr[ae]y will match either gray or grey, allowing you to quickly identify and extract either spelling.

```
>>> re.search(r'gr[ae]y', 'gray')
<_sre.SRE_Match object; span=(0, 4), match='gray'>
```

- It is also worth noting that character classes like this match *one and exactly one* character.

Character Classes

- But please also note the following:

```
>>> re.search(r'gr[ae]y', 'graey')
>>>
```

- Here, the regular expression engine successfully matches the literal g, then the literal r.
- Next, the engine is given the character class [ae], and matches it against the a.
- Now, the character class has been matched, and the engine moves on. The next character in the regular expression is a y, but the next character in the string is an e.
- This is not a match, so the regular expression parser moves on, starting over and looking for a starting g.
- When it gets to the end of the string and fails to find one, it returns None.

Ranges

- Some quite common character classes are very large.
- For example, consider trying to match any digit.
- It would be quite unwieldy to provide [0123456789] each time.
- It would be even more unwieldy to provide every letter, both capitalized and lowercase, each time.
- To accommodate for this, the regular expression engine uses the hyphen character (-) within character classes to denote ranges.
- A character class to match any digit could be written [0-9] instead.
- It is also possible to use more than one range within a character class, simply by providing the ranges next to one another.
- The [a-z] character class matches only lowercase letters, and the [A-Z] character class matches only capital letters.
- These can be combined—[A-Za-z] would match both lowercase and capital letters.

Ranges

```
>>> re.search(r'[a-zA-Z]', 'x')
<_sre.SRE_Match object; span=(0, 1), match='x'>
>>> re.search(r'[a-zA-Z]', 'B')
<_sre.SRE_Match object; span=(0, 1), match='B'>
```

- Of course, you may also want to match the literal hyphen character.
- This is surprisingly common. Many reasons exist to match (for example) alphanumeric characters, hyphen, and underscore.
- What happens when you want to do this?
- You can escape the hyphen: [A-Za-z0-9\-_].
- This will tell the regular expression engine that you want a literal hyphen.
- However, escaping generally makes things more difficult to read.
- You can also provide the hyphen as either the first or last character in the character class, as in [A-Za-z0-9_-].
- In this case, the engine will interpret the character as a literal hyphen.

Negation

- The character classes shown thus far are all defined by what characters may occur.
- However, you may want to define a character class by what characters may not occur.
- You can invert a character class (meaning that it will match any character other than those specified) by beginning the character class with a ^ character. [caret character]

```
>>> re.search(r'[^a-z]', '4')
<_sre.SRE_Match object; span=(0, 1), match='4'>
>>> re.search(r'[^a-z]', '#’)
<_sre.SRE_Match object; span=(0, 1), match='#'>
>>> re.search(r'[^a-z]', 'X')
<_sre.SRE_Match object; span=(0, 1), match='X'>
>>> re.search(r'[^a-z]', 'd')
>>>
```

Negation

- In this scenario, the regular expression parser looks for literally any character other than a through z.
- Therefore, it matches against numbers, capital letters, and symbols, but not lowercase letters.
- It is important to note specifically what the regular expression is looking for here.
- It is looking for the presence of a character that does not match any of the characters in the character class.
- It is not looking for (and will not match) the absence of a character.
- Consider the regular expression n[^e]. This means the character n followed by any character that is not an e.

```
>>> re.search(r'n[^e]', 'final')
<_sre.SRE_Match object; span=(2, 4), match='na'>
```

- In this case, it matches against the word final, and the match is na.
- The a character is part of the match, because it is a single character that is not an e.

Negation

- The regular expression will fail to match if it follows an n followed by an e, as you expect.

```
>>> re.search(r'n[^e]', 'jasmine')
>>>
```

- Here, the regular expression engine gets to the only n in the string but cannot match the next character, because it is an e, and thus there is no match.
- However, the regular expression also will not match against an n at the end of the string.

```
>>> re.search(r'n[^e]', 'Python')
>>>
```

- The regular expression finds the n in the word Python.
- However, that is as far as it gets.
- There is no character remaining in the string to match against [^e], and, therefore, the match fails.

Shortcuts

- Several common character classes also have predefined shortcuts within the regular expression engine.
- If you want to define "words," your instinct may be to use [A-Za-z]. However, many words use characters that fall outside of this range.
- The regular expression engine provides a shortcut, \w, which matches "any word character."
- How "any word character" is defined varies somewhat based on your environment.
- In Python 3, it will essentially match nearly any word character in any language.
- In Python 2, it will only match the English word characters.
- In both cases, it also matches digits, _, and -.
- The \d shortcut matches digit characters.
- In Python 3, it matches digit characters in other languages. In Python 2, it matches only [0-9].

Shortcuts

- The \s shortcut matches whitespace characters, such as space, tab, newline, and so on. The exact list of whitespace characters is greater in Python 3 than in Python 2.
- Finally, the \b shortcut matches a zero-length substring.
- However, it only matches it at the beginning or end of a word.
- This is called the *word boundary* character shortcut.

```
>>> re.search(r'\bcorn\b', 'corn')
<_sre.SRE_Match object; span=(0, 4), match='corn'>
>>> re.search(r'\bcorn\b', 'corner')
>>>
```

- The regular expression engine matches the word corn here when it is by itself, but fails to match the word corner, because the trailing \b does not match (because the next character is e, which is a word character).

Shortcuts

- It is worth noting that these shortcuts work both within character classes and outside of them.
- For example, the regular expression \w will match any word character.

```
>>> re.search(r'\w', 'Python 3')
<_sre.SRE_Match object; span=(0, 1), match='P'>
```

- Because re.search only returns the first match, it matches the P character and then completes.
- Consider the result of re.findall using the same regular expression and string.

```
>>> re.findall(r'\w', 'Python 3')
['P', 'y', 't', 'h', 'o', 'n', '3']
```

- Note that the regular expression matches every character in the string except the space. The \w shortcut does include digits in the Python regular expression engine.

Shortcuts

- The \w, \d, and \s shortcuts also include *negation shortcuts:* \W, \D, and \S.
- These shortcuts match any character other than the characters in the shortcut.
- Note again that these still require a character to be present. They do not match an empty string.
- There is also a negation shortcut for \b, but it works slightly differently.
- Whereas \b matches a zero-length substring at the beginning or end of a word, \B matches a zero-length substring that is not at the beginning or end of a word.
- This essentially reverses the corn and corner example from earlier.

```
>>> re.search(r'corn\B', 'corner')
<_sre.SRE_Match object; span=(0, 4), match='corn'

> >>> re.search(r'corn\B', 'corn')
>>>
```

Beginning and End of String

- Two special characters designate the beginning of a string and end of a string.
- The ^ character designates the beginning of a string, as shown here:

```
>>> re.search(r'^Python', 'This code is in Python.')

>>> re.search(r'^Python', 'Python 3')
<_sre.SRE_Match object; span=(0, 6), match='Python'>
```

- Notice that the first command fails to produce a match.
- This is because the string does not start with the word Python, and the ^ character requires that the regular expression match against the beginning of the string.

Beginning and End of String

- Similarly, the $ character designates the end of a string, as shown here:

```
>>> re.search(r'fox$', 'The quick brown fox jumped over the lazy dogs.')

>>> re.search(r'fox$', 'The quick brown fox')
<_sre.SRE_Match object; span=(16, 19), match='fox'>
```

- Again, notice that the first command fails to produce a match, because although the word fox appears, it is not at the end of the string, which the $ character requires.

Any Character

- The . character is the final shortcut character.
- It stands in for any single character.
- However, it only serves this role outside a bracketed character class.
- Consider the following simple regex using the . character:

```
>>> re.search(r'p.th.n', 'python 3')
<_sre.SRE_Match object; span=(0, 6), match='python'>
>>> re.search(r'p..hon', 'python 3')
<_sre.SRE_Match object; span=(0, 6), match='python'>
```

- In each of these cases, the period steps in for one single character.
- In the first example, the regular expression engine finds the character . in the regular expression.
- In the string, it sees a y, and matches and continues to the next character (a t against a t).

Any Character

- In the second case, the same fundamental thing is happening.
- Each period character matches *one and exactly one* character.
- It matches the y and the t, and then this consumes both of the periods, and the regular expression engine continues to the next character (this time, an h against an h).
- Note that there is one character that the . does not match, which is newline (\n).
- It is possible to make the . character match newline, however, which is discussed later.

Optional Characters

- Thus far, all of the regular expressions you have seen have involved a 1:1 correlation between characters in the regular expression itself and characters in the string being searched.
- Sometimes, however, a character may be optional.
- Consider again the example of a word with more than one correct spelling, but this time, the inclusion of a letter is what separates the two spellings, such as “color” and “colour,” or “honor” and “honour.”
- You can specify a character, character class, or other atomic unit within a regular expression as optional by using the ? character, which means that the regular expression engine will expect the token to occur either zero times or once.
- For example, you can match the word “honor” with its British spelling “honour” by using the regular expression honou?r.

Optional Characters

```
>>> import re
>>> re.search(r'honou?r', 'He served with honor and distinction.')
<_sre.SRE_Match object; span=(15, 20), match='honor'>
>>> re.search(r'honou?r', 'He served with honour and distinction.')
<_sre.SRE_Match object; span=(15, 21), match='honour'>
```

- In both cases, the regular expression contains four literal characters, hono.
- These match the hono in both honor and honour.
- The next thing that the regular expression hits is an optional u.
- In the first case, the u is absent, but this is okay because the regular expression marks it as optional.
- In the second case, the u is present, which is also okay.
- In both cases, the regular expression then seeks a literal r character, which it finds, therefore completing the match.

Repetition

- Thus far, you have learned only about characters (or character classes) that occur once and exactly once, or that are entirely optional (occurring zero times or once).
- However, sometimes you need the same character or character class to repeat.
- You may expect a character class to recur a set number of consecutive times, such as in a phone number.
- American phone numbers comprise the country code 1 (often omitted), an area code, which is three digits, then the seven-digit phone number, with the third and fourth digit of the latter separated by a hyphen, period, or similar.
- You can designate that a token must repeat a given number of times with {N}, where the N character corresponds to the number of times the token should repeat.

Repetition

- The following uses a regular expression to identify a seven-digit, local phone number (ignore the country code and area code for the moment): [\d]{3}-[\d]{4}.

```
>>> re.search(r'[\d]{3}-[\d]{4}', '867-5309 / Jenny')
<_sre.SRE_Match object; span=(0, 8), match='867-5309'>
```

- In this case, the regular expression engine starts by looking for three consecutive digits.
- It finds them (867), and then moves on to the literal hyphen character.
- Because this hyphen character is not within a character class, it carries no special meaning and simply matches the literal hyphen.
- The regular expression then finds the final four consecutive digits (5309) and returns the match.
- Sometimes, you may not know exactly how many times the token ought to repeat. Phone numbers may contain a static number of digits, but lots of numeric data is not standardized this way.

Repetition

- For example, consider credit card security codes.
- Credit cards contain a special security code on the back, often called a “CVV code.”
- Most credit card brands use three-digit security codes, which you can match with [\d]{3}.
- However, American Express uses four-digit security codes ([\d]{4}).
- What if you want to be able to match both of these cases?
- Repetition ranges come in handy here.
- The syntax here is {M,N}, where M is the lower bound and N is the upper bound.
- It is worth noting here that the bounds are inclusive.
- If you want to match three digits or four digits, the correct syntax is [\d]{3,4}.
- You might be tempted (based on using Python slices) to believe that the upper bound is exclusive (and that you should use {3,5} instead).
- However, regular expressions do not work this way.

Repetition

```
>>> re.search(r'[\d]{3,4}', '0421')
<_sre.SRE_Match object; span=(0, 4), match='0421'>
```

- In both cases, the regular expression engine finds a series of digits that matches what it expects, and returns a match.
- When given the choice to match three characters *or* four characters, where either is a valid match, how does the regular expression engine decide?
- The answer is that, under most circumstances, the regular expression engine is “greedy,” meaning that it will match as many characters as possible for as long as it can.
- In this simple case, that means that if there are four digits, four digits will be matched.

Repetition

- Occasionally, this behavior is undesirable.
- By placing a ? character immediately after the repetition operator, it causes that repetition to be considered "lazy," meaning that the engine will match as few characters as possible to return a valid match.

```
>>> re.search(r'[\d]{3,4}?', '0421')
<_sre.SRE_Match object; span=(0, 3), match='042'>
```

- The re-use of the ? character for another purpose does not cause any ambiguity for the parser, because the character comes after repetition syntax, rather than a token to be matched against.
- Note that the ? in this situation *does not* serve to make the repeated segment optional.
- It simply means that, given the opportunity to match three or four digits, it will elect only to match three.

- *Note that the* ? *character used to make a token optional is essentially an exact alias for* {0,1}.

Open-ended Ranges

- You also may encounter cases where there is no upper bound for the number of times that a token may repeat.
- For example, consider a traditional street address.
- This usually starts with a number (for the moment, hand-wave the exceptions and assert that they always do), but the number could be any arbitrary length.
- There is nothing technically invalid about an eight-digit street number.
- In these cases, you can leave off the upper bound, but retain the , character to designate that the upper bound is ∞.
- For example, {1,} designates one or more occurrences with no upper bound.

```
>>> re.search(r'[\d]{1,}', '1600 Pennsylvania Ave.')
<_sre.SRE_Match object; span=(0, 4), match='1600'>
```

- This syntax also works if you do not want to specify a lower bound, in which case, the lower bound is assumed to be 0.

Shorthand

- You can use two shorthand characters in designating common repetition situations.
- You can use the + character in lieu of specifying {1,} (one or more).
- Similarly, you can use the * character in lieu of specifying {0,} (zero or more).
- Therefore, the previous example could be rewritten using +, as shown here:

```
>>> re.search(r'[\d]+', '1600 Pennsylvania Ave.')
<_sre.SRE_Match object; span=(0, 4), match='1600'>
```

- Using + and * generally makes for a regular expression that is easier to read, and is the preferred syntax in cases where they are applicable.

Grouping

- Regular expressions provide a mechanism to split the expression into groups.
- When using groups, you are able to select each individual group within the match in addition to getting the entire match.
- You can specify groups within a regular expression by using parentheses.
- The following is an example of a simple, local phone number.
- However, this time, each set of digits is a group.

```
>>> match = re.search(r'([\d]{3})-([\d]{4})', '867-5309 / Jenny')
>>> match
<_sre.SRE_Match object; span=(0, 8), match='867-5309'>
```

- As before, you can use the group method on the match object to return the entire match.
- >>> match.group()
- '867-5309'

Grouping

- The re module’s match objects provide a method, groups, which returns a tuple corresponding to each individual group.

```
>>> match.groups()
('867', '5309')
```

- By breaking your regular expression into subgroups like this, you can quickly get not just the entire match, but specific bits of data within the match.
- It is also possible to get just a single group, by passing an argument to the group method corresponding to the group you want back (note that group numbers are 1-indexed).

```
>>> match.group(2)
'5309'
```

- By using groups, you can take a phone number formatted in a variety of different ways and extract only the data that matters, which is the actual digits of a phone number.

Grouping

```
>>> re.search(
...     r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})',
...     '(213) 867-5309')
<_sre.SRE_Match object; span=(0, 14), match='(213) 867-5309'>
>>> re.search(
...     r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})',
...     '213-867-5309')
<_sre.SRE_Match object; span=(0, 12), match='213-867-5309'>
>> re.search(
…     r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})',
…     '213.867.5309')
<_sre.SRE_Match object; span=(0, 12), match='213.867.5309'>
>>> re.search(
...     r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})',
...     '2138675309')
<_sre.SRE_Match object; span=(0, 10), match='2138675309'>
```

Grouping

```
>>> re.search(
...      r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})', '+1
...     (213) 867-5309')
<_sre.SRE_Match object; span=(0, 17), match='+1 (213) 867-5309'>
>>> re.search(
...     r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})', '1
...     (213) 867-5309')
<_sre.SRE_Match object; span=(0, 16), match='1 (213) 867-5309'>
>>> re.search(
...     r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})’,
...      '1-213-867-5309')
<_sre.SRE_Match object; span=(0, 14), match='1-213-867-5309'>
```

- This regular expression is a bit more complicated than what you have encountered already.
- Consider each distinct part by itself, however, and it is easier to parse.

Grouping

- The first segment is (\+?1)?[.-]?.
- This is first looking for the United States country code in almost any format you may encounter it (+1 or 1, and then possibly a hyphen).
- The second segment is \(?([\d]{3})\)?[.-]?, and it grabs the area code, and the optional hyphen or whitespace that may follow it.
- The area code may optionally be provided in parentheses (as is common with U.S. phone numbers).
- The remainder of the regular expression is the final seven digits of the phone number, and is the same as what you have already seen.
- Regardless of how the phone number is formatted, the regular expression is capable of matching it.
- And although the full match is still formatted based on the original data provided, the groups are consistently the same.

```
>>> match = re.search(
...     r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})',
...     ‘213-867-5309')
>>> match.groups()
(None, '213', '867', '5309')
```

Grouping

```
>>> match = re.search(
...     r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})',
…     ‘+1 213-867-5309')
>>> match.groups()
(‘+1’, '213', '867', '5309')
```

- The only difference between the groups is based on what was provided for the country code.
- If it is omitted, then it is not captured either, and None is provided in its place.
- The second through fourth groups consistently contain the three (intra-national) segments of the phone number.

- There are other details as well ----- we will not discuss here, except last 2 topics:

Substitution

- The regular expression engine is not limited to simply identifying whether a pattern exists within a string.
- It is also capable of performing string replacement, returning a new string based on the groups in the original one.
- The substitution method in Python is re.sub.
- It takes three arguments: the regular expression, the replacement string, and the source string being searched.
- Only the actual match is replaced, so if there is no match, re.sub ends up being a no-op.
- re.sub enables you to use the same backreferences from regular expression patterns within the replacement string.
- Consider the task of stripping irrelevant formatting data from a phone number:

```
>>> re.sub(r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})',
... r'\2\3\4',
... '213-867-5309')
'2138675309'
```

Substitution

- Because this regular expression matches nearly any phone number and groups only the actual digits of the phone number, you will get back the same data regardless of how the original number was formatted.

```
>>> re.sub(r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})', r'\2\3\4',
...  '213.867.5309')
'2138675309'
>>> re.sub(r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})',  r'\2\3\4',
...  '2138675309')
'2138675309'
>>> re.sub(r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})',  r'\2\3\4',
...  '(213) 867-5309')
'2138675309'
>>> re.sub(r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})',  r'\2\3\4',
...  '1 (213) 867-5309')
'2138675309'
>>> re.sub(r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})',  r'\2\3\4',
...  '+1 213-867-5309')
'2138675309'
```

Substitution

- The replacement string is not limited to just using the backreferences from the string; other characters are interpreted literally.
- Therefore, re.sub can also be used for formatting.
- For example, what if you want to display a phone number rather than store it, but you want to display it in a consistent format?
- re.sub can handle that, as shown here:

```
>>> re.sub(r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})',  r'(\2) \3-\4',
...  '+1 213-867-5309')
'(213) 867-5309'
```

- Everything here is the same as in the previous examples, except for the replacement string, which has gained the parentheses, space, and hyphen.
- Therefore, so has the result.

Compiled RegEx

- One final feature of Python’s regular expression implementation is *compiled regular expressions*.
- The re module contains a function, compile, which returns a compiled regular expression object, which can then be reused.
- The re module caches regular expressions that it compiles on thefl y, so in most situations, there is no substantial performance advantage to using compile.
- It can be extremely useful for passing regular expression objects around, however.
- The re.compile function returns a regular expression object, with the compiled regular expression as data.
- These objects have their own search and sub methods, which omit the first argument (the regular expression itself).

```
>>> regex = re.compile(
    ... r'(\+?1)?[ .-]?\(?([\d]{3})\)?[ .-]?([\d]{3})[ .-]?([\d]{4})'
    ... )
>>> regex.search('213-867-5309')
<_sre.SRE_Match object; span=(0, 12), match='213-867-5309'>
>>> regex.sub(r'(\2) \3-\4', '+1 213.867.5309')
'(213) 867-5309'
```

Compiled RegEx

- Also, there is one other advantage to using re.compile.
- The search method of regular expression objects actually allows for two additional arguments not available on re.search.
- These are the starting and ending positions of the string to be searched against, enabling you to exempt some of the string from consideration.

```
>>> regex = re.compile('[\d]+')
>>> regex.search('1 mile is equal to 5280 feet.')
<_sre.SRE_Match object; span=(0, 1), match='1'>
>>> regex.search('1 mile is equal to 5280 feet.', pos=2)
<_sre.SRE_Match object; span=(19, 23), match='5280'>
```

- The values sent are available as the pos and endpos attributes on the match objects returned.

- So --- Regular expressions are extremely useful tools for nding, parsing, and validating data. They often look intimidating to those who have not used them before, but they are manageable if taken piece by piece.
- In addition, mastering regular expressions will enable you to perform parsing and formatting tasks that are much more difficult without a pattern-matching algorithm.

Reference

- Learning Python – 5th Ed – Mark Lutz - [O’Reilly]
- Professional Python – Luke Sneeringer [Wrox]

Iterations Revisited

Iterations Revisited

- In iterations we have discussed the ‘while’ loop and the ‘for’ loop.
- The general syntax of while loop is :

```
while test:              # Loop test
    statements           # Loop body
else:                    # Optional else
    statements           # Run if didn't exit loop with break
```

- Let’s see an example where we are using the ‘else’ part:

```
x = y // 2   # Here ‘//’ is called floor division which truncates fractional remainders
while x > 1:
    if y % x == 0:
        print(y, 'has factor', x)
        break
    x -= 1
else:
    print(y, 'is prime')
```

Iterations Revisited

- The general syntax of the for loop is :

```
for target in object:          # Assign object items to target
    statements                 # Repeated loop body: use target
else:                          # Optional else part
    statements                 # If we didn't hit a 'break'
```

- An example:

```
items = ["aaa", 111, (4, 5), 2.01]
tests =  [(4, 5), 3.14]
for key in tests:
    for item in items:
        if item == key:
            print (key, 'was found')
            break
    else:
        print(key, 'not found!')
```

- Run would show:

```
(4, 5) was found
3.14 not found!
```

map Function

'map' built-in Function

- *map* is a very useful higher order function in Python.
- It takes one function and an iterator as input and then applies the function on each value of the iterator and returns a list of results.
- Consider an example

```
>>> lst = [1, 2, 3, 4, 5]
>>> def square(num):
...          "Returns the square of a given number."
...          return num * num
...
>>> print(list(map(square, lst)))

[1, 4, 9, 16, 25]
```

filter Function

‘filter’ built-in Function

- The function

  filter(function, sequence)

  offers an elegant way to filter out all the elements of a sequence "sequence", for which the function *function* returns True. i.e. an item will be produced by the iterator result of filter(function, sequence) if item is included in the sequence "sequence" and if function(item) returns True.

- In other words: The function filter(f,l) needs a function f as its first argument. f has to return a Boolean value, i.e. either True or False. This function will be applied to every element of the list *l*. Only if f returns True will the element be produced by the iterator, which is the return value of filter(function, sequence).
- In the following example, we filter out first the odd and then the even elements of the sequence of the first 11 Fibonacci numbers:

filter Function

‘filter’ built-in Function

```
>>> fibonacci = [0,1,1,2,3,5,8,13,21,34,55]
>>> odd_numbers = list(filter(lambda x: x % 2, fibonacci))
>>> print(odd_numbers)
 [1, 1, 3, 5, 13, 21, 55]

>>> even_numbers = list(filter(lambda x: x % 2 == 0, fibonacci))
>>> print(even_numbers)
[0, 2, 8, 34]

>>>
```

Reference

- Python for Informatics – C. Severance
- Think Python – A. B. Downey - [O’Reilly]
- Python Crash Course – Eric Matthes [No starch Press]
- A Byte of Python – Swaroop C H
- Introducing Python – Bill Lubanovic [O’Reilly]

- Learning Python – 5th Ed – Mark Lutz - [O’Reilly]

End of Presentation

Python – S10