Regular expressions

This lab covers

- [Understanding regular expressions]
- [Creating regular expressions with special characters]
- [Using raw strings in regular expressions]
- [Extracting matched text from strings]
- [Substituting text with regular expressions]

Some might wonder why I'm discussing regular expressions in this course at all. Regular expressions are implemented by a single Python module and are advanced enough that they don't even come as part of the standard library in languages like C or Java. But if you're using Python, you're probably doing text parsing; if you're doing that, regular expressions are too useful to be ignored. If you've used Perl, Tcl, or Linux/UNIX, you may be familiar with regular expressions; if not, this lab goes into them in some detail.

What is a regular expression?

A *regular expression* (regex) is a way of recognizing and often extracting data from certain patterns of text. A regex that recognizes a piece of text or a string is said to *match* that text or string. A regex is defined by a string in which certain characters (the so-called *metacharacters*) can have a special meaning, which enables a single regex to match many different specific strings.

It's easier to understand this through example than through explanation. Here's a program with a regular expression that counts how many lines in a text file contain the word *hello*. A line that contains *hello* more than once is counted only once:

```
import re
regexp = re.compile("hello")
count = 0
file = open("textfile", 'r')
for line in file.readlines():
    if regexp.search(line):
        count = count + 1
file.close()
print(count)
```

The program starts by importing the Python regular expression module, called [re]. Then it takes the text string ["hello"] as a *textual regular expression* and compiles it into a *compiled regular expression*, using the [re.compile] function. This compilation isn't strictly necessary, but compiled regular expressions can significantly increase a program's speed, so they're almost always used in programs that process large amounts of text.

What can the regex compiled from ["hello"] be used for? You can use it to recognize other instances of the word ["hello"] within another string; in other words, you can use it to determine whether another string contains ["hello"] as a substring. This task is accomplished by the [search] method, which returns [None] if the regular expression isn't found in the string argument; Python interprets [None] as [false] in a Boolean context. If the regular expression is found in the string, Python returns a special object that you can use to determine various things about the match (such as where in the string it occurred). I discuss this topic later.

Regular expressions with special characters

The previous example has a small flaw: It counts how many lines contain ["hello"] but ignores lines that contain ["Hello"] because it doesn't take capitalization into account.

One way to solve this problem would be to use two regular expressions---one for ["hello"] and one for ["Hello"]---and test each against every line. A better way is to use the more advanced features of regular expressions. For the second line in the program, substitute

```
regexp = re.compile("hello|Hello")
```

This regular expression uses the vertical-bar special character [|]. A *special character* is a character in a regex that isn't interpreted as itself; it has some special meaning. [|] means *or*, so the regular expression matches ["hello"] *or* ["Hello"].

Another way of solving this problem is to use

```
regexp = re.compile("(h|H)ello")
```

In addition to using [|], this regular expression uses the *parentheses* special characters to group things, which in this case means that the [|] chooses between a small or capital *H*. The resulting regex matches either an *h* or an *H*, followed by *ello*.

Another way to perform the match is

```
regexp = re.compile("[hH]ello")
```

The special characters [[] and []] take a string of characters between them and match any single character in that string. There's a special shorthand to denote ranges of characters in [[] and []]; [[a-z]] match a single character between *a* and *z*, [[0-9A-Z]] match any digit or any uppercase character, and so forth. Sometimes, you may want to include a real hyphen in the [[]], in which case you should put it as the first character to avoid defining a range; [[-012]] match a hyphen, a *0*, a *1*, or a *2*, and nothing else.

Quite a few special characters are available in Python regular expressions, and describing all of the subtleties of using them in regular expressions is beyond the scope of this course. A complete list of the special characters available in Python regular expressions, as well as descriptions of what they mean, is in the online documentation of the regular expression [re] module in the standard library. For the remainder of this lab, I describe the special characters I use as they appear.

Quick Check: Special characters in regular expressions
What regular expression would you use to match strings that represent the numbers -5 through 5?

What regular expression would you use to match a hexadecimal digit? Assume that allowed hexadecimal digits are 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, a, B, b, C, c, D, d, E, e, F, and f.

Regular expressions and raw strings

The functions that compile regular expressions, or search for matches to regular expressions, understand that certain character sequences in strings have special meanings in the context of regular expressions. regex functions understand that [\n] represents a newline character, for example. But if you use normal Python strings as regular expressions, the regex functions typically never see such special sequences, because many of these sequences also possess a special meaning in normal strings. [\n], for example, also means newline in the context of a normal Python string, and Python automatically replaces the string sequence [\n] with a newline character before the [regex] function ever sees that sequence. The [regex] function, as a result, compiles strings with embedded newline characters---not with embedded [\n] sequences.

In the case of [\n], this situation makes no difference because [regex] functions interpret a newline character as exactly that and do the expected thing: attempt to match the character with another newline character in the text being searched.

Now look at another special sequence, [\\], which represents a *single* backslash to regular expressions. Assume that you want to search text for an occurrence of the string ["\ten"]. Because you know that you have to represent a backslash as a double backslash, you might try

```
regexp = re.compile("\\ten")
```

This example compiles without complaining, but it's wrong. The problem is that [\\] also means a single backslash in Python strings. Before [re.compile] is invoked, Python interprets the string you typed as meaning [\ten], which is what is passed to [re.compile]. In the context of regular expressions, [\t] means *tab*, so your compiled regular expression searches for a tab character followed by the two characters *en*.

To fix this problem while using regular Python strings, you need four backslashes. Python interprets the first two backslashes as a special sequence representing a single backslash, and likewise for the second pair of backslashes, resulting in two *actual* backslashes in the Python string. Then that string is passed in to [re.compile], which interprets the two actual backslashes as a regex special sequence representing a single backslash. Your code looks like this:

```
regexp = re.compile("\\\\ten")
```

That seems confusing, and it's why Python has a way of defining strings that doesn't apply the normal Python rules to special characters. Strings defined this way are called *raw strings*.

Raw strings to the rescue

A raw string looks similar to a normal string except that it has a leading *r* character immediately preceding the initial quotation mark of the string. Here are some raw strings:

```
r"Hello"
r"""\tTo be\n\tor not to be"""
r'Goodbye'
r'''12345'''
```

As you can see, you can use raw strings with either the single or double quotation marks and with the regular or triple-quoting convention. You can also use a leading *R* instead of *r* if you want to. No matter how you do it, raw-string notation can be taken as an instruction to Python saying "Don't process special sequences in this string." In the previous examples, all the raw strings are equivalent to their normal string counterparts except the second example, in which the [\t] and [\n] sequences aren't interpreted as tabs or newlines but are left as two-string character sequences beginning with a backslash.

Raw strings aren't different types of strings. They represent a different way of *defining* strings. It's easy to see what's happening by running a few examples interactively:

```
>>> r"Hello" == "Hello"
True
>>> r"\the" == "\\the"
True
>>> r"\the" == "\the"
False
>>> print(r"\the")
\the
>>> print("\the")
        he
```

Using raw strings with regular expressions means that you don't need to worry about any funny interactions between string special sequences and regex special sequences. You use the regex special sequences. Then the previous regex example becomes

```
regexp = re.compile(r"\\ten")
```

which works as expected. The compiled regex looks for a single backslash followed by the letters *ten*.

You should get into the habit of using raw strings whenever defining regular expressions, and you'll do so for the remainder of this lab.

Extracting matched text from strings

One of the most common uses of regular expressions is to perform simple pattern-based parsing on text. This task is something you should know how to do, and it's also a good way to learn more regex special characters.

Assume that you have a list of people and phone numbers in a text file. Each line of the file looks like this:

```
surname, firstname middlename: phonenumber
```

You have a surname followed by a comma and space, followed by a first name, followed by a space, followed by a middle name, followed by colon and a space, followed by a phone number.

But to make things complicated, a middle name may not exist, and a phone number may not have an area code. (It might be 800-123-4567 or 123-4567.) You *could* write code to explicitly parse data out from such a line, but that job would be tedious and error-prone. Regular expressions provide a simpler answer.

Start by coming up with a regex that matches lines of the given form. The next few paragraphs throw quite a few special characters at you. Don't worry if you don't get them all on the first read; as long as you understand the gist of things, that's all right.

For simplicity's sake, assume that first names, surnames, and middle names consist of letters and possibly hyphens. You can use the [[]] special characters discussed in the previous section to define a pattern that defines only name characters:

```
[-a-zA-z]
```

This pattern matches a single hyphen, a single lowercase letter, or a single uppercase letter.

To match a full name (such as McDonald), you need to repeat this pattern. The [+] metacharacter repeats whatever comes before it one or more times as necessary to match the string being processed. So the pattern

```
[-a-zA-Z]+
```

matches a single name, such as Kenneth or McDonald or Perkin-Elmer. It also matches some strings that aren't names, such as --- or -a-b-c-, but that's all right for purposes of this example.

Now, what about the phone number? The special sequence [\d] matches any digit, and a hyphen outside [[]] is a normal hyphen. A good pattern to match the phone number is

```
\d\d\d-\d\d\d-\d\d\d\d
```

That's three digits followed by a hyphen, followed by three digits, followed by a hyphen, followed by four digits. This pattern matches only phone numbers with an area code, and your list may contain numbers that don't have one. The best solution is to enclose the area-code part of the pattern in [()]; group it; and follow that group with a [?] special character, which says that the thing coming immediately before the [?] is optional:

```
(\d\d\d-)?\d\d\d-\d\d\d\d
```

This pattern matches a phone number that may or may not contain an area code. You can use the same sort of trick to account for the fact that some of the people in your list have middle names (or initials) included and others don't. (To do so, make the middle name optional by using grouping and the [?] special character.)

You can also use [{}] to indicate the number of times that a pattern should repeat, so for the phone-number examples above, you could use:

```
(\d{3}-)?\d{3}-\d{4}
```

This pattern also means an optional group of three digits plus a hyphen, three digits followed by a hyphen, and then four digits.

Commas, colons, and spaces don't have any special meanings in regular expressions; they mean themselves.

Putting everything together, you come up with a pattern that looks like this:

```
[-a-zA-Z]+, [-a-zA-Z]+( [-a-zA-Z]+)?: (\d{3}-)?\d{3}-\d{4}
```

A real pattern probably would be a bit more complex, because you wouldn't assume that there's exactly one space after the comma, exactly one space after the first and middle names, and exactly one space after the colon. But that's easy to add later.

The problem is that, whereas the above pattern lets you check to see whether a line has the anticipated format, you can't extract any data yet. All you can do is write a program like this:

```
import re
regexp = re.compile(r"[-a-zA-Z]+,"
                    r" [-a-zA-Z]+"
                    r"( [-a-zA-Z]+)?"
                    r": (\d{3}-)?\d{3}-\d{4}"
                   )
file = open("textfile", 'r')
for line in file.readlines():
    if regexp.search(line):
        print("Yeah, I found a line with a name and number. So what?")
file.close()
```

Notice that you've split your regex pattern, using the fact that Python implicitly concatenates any set of strings separated by whitespace. As your pattern grows, this technique can be a great aid in keeping the pattern maintainable and understandable. It also solves the problem with the line length possibly increasing beyond the right edge of the screen.

Fortunately, you can use regular expressions to extract data from patterns, as well as to see whether the patterns exist. The first step is to group each subpattern corresponding to a piece of data you want to extract by using the [()] special characters. Then give each subpattern a unique name with the special sequence [?P<name>], like this:

```
(?P<last>[-a-zA-Z]+), (?P<first>[-a-zA-Z]+)( (?P<middle>([-a-zA-Z]+)))?:
(?P<phone>(\d{3}-)?\d{3}-\d{4}
```

(Please note that you should enter these lines as a single line, with no line breaks. Due to space constraints, the code can't be represented here in that manner.)

There's an obvious point of confusion here: The question marks in [?P<...>] and the question-mark special characters indicating that the middle name and area code are optional have nothing to do with one another. It's an unfortunate semi-coincidence that they happen to be the same character.

Now that you've named the elements of the pattern, you can extract the matches for those elements by using the [group] method. You can do so because when the [search] function returns a successful match, it doesn't return just a truth value; it also returns a data structure that records what was matched. You can write a simple program to extract names and phone numbers from your list and print them out again, as follows:

```
import re
regexp = re.compile(r"(?P<last>[-a-zA-Z]+),"
                    r" (?P<first>[-a-zA-Z]+)"
                    r"( (?P<middle>([-a-zA-Z]+)))?"
                    r": (?P<phone>(\(\d{3}-)?\d{3}-\d{4})"
                   )
file = open("textfile", 'r')
for line in file.readlines():
    result = regexp.search(line)
    if result == None:
        print("Oops, I don't think this is a record")
    else:
        lastname = result.group('last')
        firstname = result.group('first')
        middlename = result.group('middle')
        if middlename == None:
                  middlename = ""
        phonenumber = result.group('phone')
    print('Name:', firstname, middlename, lastname,' Number:', phonenumber)
file.close()
```

There are some points of interest here:

- [You can find out whether a match succeeded by checking the value returned by [search]. If the value is [None], the match failed; otherwise, the match succeeded, and you can extract information from the object returned by [search].]
- [[group] is used to extract whatever data matched your named subpatterns. You pass in the name of the subpattern you're interested in.]
- [Because the middle subpattern is optional, you can't count on it to have a value, even if the match as a whole is successful. If the match succeeds, but the match for the middle name doesn't, using [group] to access the data associated with the middle subpattern returns the value [None].]
- [Part of the phone number is optional, but part isn't. If the match succeeds, the phone subpattern must have some associated text, so you don't have to worry about it having a value of [None].]

Try this: Extracting matched text

Making international calls usually requires a + and the country code. Assuming that the country code is two digits, how would you modify the code above to extract the + and the country code as part of the number? (Again, not all numbers have a country code.) How would you make the code handle country codes of one to three digits?

Substituting text with regular expressions

In addition to extracting strings from text, you can use Python's regex module to find strings in text and substitute other strings in place of those that were found. You accomplish this task by using the regular substitution method [sub]. The following example replaces instances of ["the the"] (presumably, a typo) with single instances of ["the"]:

```
>>> import re
>>> string = "If the the problem is textual, use the the re module"
>>> pattern = r"the the"
>>> regexp = re.compile(pattern)
>>> regexp.sub("the", string)
'If the problem is textual, use the re module'
```

The [sub] method uses the invoking regex ([regexp], in this case) to scan its second argument ([string], in the example) and produces a new string by replacing all matching substrings with the value of the first argument (["the"], in this example).

But what if you want to replace the matched substrings with new ones that reflect the value of those that matched? This is where the elegance of Python comes into play. The first argument to [sub]---the replacement substring, ["the"] in the example---doesn't have to be a string at all. Instead, it can be a function. If it's a function, Python calls it with the current match object; then it lets that function compute and return a replacement string.

To see this function in action, build an example that takes a string containing integer values (no decimal point or decimal part) and returns a string with the same numerical values but as floating numbers (with a trailing decimal point and zero):

```
>>> import re
>>> int_string = "1 2 3 4 5"
>>> def int_match_to_float(match_obj):
...     return(match_obj.group('num') + ".0")
...
>>> pattern = r"(?P<num>[0-9]+)"
>>> regexp = re.compile(pattern)
>>> regexp.sub(int_match_to_float, int_string)
'1.0 2.0 3.0 4.0 5.0'
```

In this case, the pattern looks for a number consisting of one or more digits (the [[0-9]+] part). But it's also given a name (the [?P<num>...] part) so that the replacement string function can extract any matched substring by referring to that name. Then the [sub] method scans down the argument string ["1 2 3 4 5"], looking for anything that matches [[0-9]+]. When [sub] finds a substring that matches, it makes a match object defining exactly which substring matched the pattern, and it calls the [int_match_to_float] function with that match object as the sole argument. [int_match_to_float] uses [group] to extract the matching substring from the match object (by referring to the group name [num]) and produces a new string by concatenating the matched substring with a [".0"]. [sub] returns the new string and incorporates it as a substring into the overall result. Finally, [sub] starts scanning again right after the place where it found the last matching substring, and it keeps going like that until it can't find any more matching substrings.

Try this: Replacing text

In the checkpoint in [section 16.4], you extended a phone-number regular expression to also recognize a country code. How would you use a function to make any numbers that didn't have a country code now have +1 (the country code for the United States and Canada)?

Lab 16: Phone-Number normalizer

In the United States and Canada, phone numbers consist of ten digits, usually separated into a three-digit area code, a three-digit exchange code, and a four-digit station code. As mentioned in [section 16.4], they may or may not be preceded by +1, the country code. In practice, however, you have many ways to format a phone number, such as (NNN) NNN-NNNN, NNN-NNN-NNNN, NNN NNN-NNNN, NNN.NNN.NNNN, and NNN NNN NNNN, to name a few. Also, the country code may not be present, may not have a +, and usually (not always) is separated from the number by a space or dash. Whew!

In this lab, your task is to create a phone-number normalizer that takes any of the formats and returns a normalized phone number 1-NNN-NNN-NNNN.

The following are all possible phone numbers:

+1 223-456-7890 1-223-456-7890 +1 223 456-7890 (223) 456-7890 1 223 456 7890 223.456.7890

Bonus: The first digit of the area code and the exchange code can only be 2-9, and the second digit of an area code can't be 9. Use this information to validate the input and return a [ValueError] exception of [invalid phone number] if the number is invalid.

Summary

- [For a complete list and explanation of the regex special characters, refer to the Python documentation.]
- [In addition to the [search] and [sub] methods, many other methods can be used to split strings, extract more information from [match] objects, look for the positions of substrings in the main argument string, and precisely control the iteration of a regex search over an argument string.]
- [Besides the [\d] special sequence, which can be used to indicate a digit character, many other special sequences are listed in the documentation.]
- [There are also regex flags, which you can use to control some of the more esoteric aspects of how extremely sophisticated matches are carried out.]