Chapter 1

Appendix: Regular Expressions

This section explores regular expressions in detail, with examples drawn from language processing. It builds on the brief overview given in the introductory programming chapter.

We have already noted that a text can be viewed as a string of characters. What kinds of processing are performed at the character level? Perhaps word games are the most familiar example of such processing. In completing a crossword we may want to know which 3-letter English words end with the letter `c` (e.g. `arc`). We might want to know how many words can be formed from the letters: `a`, `c`, `e`, `o`, and `n` (e.g. `ocean`). We may want to find out which unique English word contains the substring `gnt` (left as an exercise for the reader). In all these examples, we are considering which word - drawn from a large set of candidates - matches a given pattern. To put this in a more computational framework, we could imagine searching through a large digital corpus in order to find all words that match a particular pattern. There are many serious uses of this so-called *pattern matching*.

One instructive example is the task of finding all doubled words in a text; an example would be the string `for for example`. Notice that we would be particularly interested in finding cases where the words were split across a linebreak (in practice, most erroneously doubled words occur in this context). Consequently, even with such a relatively banal task, we need to be able to describe patterns which refer not just to "ordinary" characters, but also to formatting information.

There are conventions for indicating structure in strings, also known as *formatting*. For example, there are a number of alternative ways of formatting a "date string", such as `23/06/2002`, `6/23/02`, or `2002-06-23`. Whole texts may be formatted, such as an email message which contains header fields followed by the message body. Another familiar form of formatting involves visual structure, such as tabular format and bulleted lists.

Finally, texts may contain explicit "markup", such as `<abbrev>Phil</abbrev>`, which provides information about the interpretation or presentation of some piece of text. To summarize, in language processing, strings are ubiquitous, and they often contain important structure.

So far we have seen elementary examples of pattern matching, the matching of individual characters. More often we are interested in matching *sequences* of characters. For example, part of the operation of a naive spell-checker could be to remove a word-final `s` from a suspect word token, in case the word is a plural, and see if the putative singular form exists in the dictionary. For this we must locate `s` and remove it, but only if it precedes a word boundary. This requires matching a pattern consisting of two characters.

Beyond this pattern matching on the *content* of a text, we often want to process the *formatting* and *markup* of a text. We may want to check the formatting of a document (e.g. to ensure that every sentence begins with a capital letter) or to reformat a document (e.g. replacing sequences of space characters with a single space). We may want to find all date strings and extract the year. We may want to extract all words contained inside the `<abbrev> </abbrev>` markup in order to construct a list of abbreviations.

Processing the content, format and markup of strings is a central task in most kinds of NLP. The most widespread method for string processing uses *regular expressions*.

1.1 Simple Regular Expressions

In this section we will see the building blocks for simple regular expressions, along with a selection of linguistic examples. We can think of a regular expression as *a specialized notation for describing patterns that we want to match*. In order to make explicit when we are talking about a pattern *patt*, we will use the notation «`patt`». The first thing to say about regular expressions is that most letters match themselves. For example, the pattern «`sing`» exactly matches the string `sing`. In addition, regular expressions provide us with a set of *special characters*[1] which give us a way to match *sets of strings*, and we will now look at these.

1.1.1 The Wildcard

The "`.`" symbol is called a *wildcard*: it matches any single character. For example, the regular expression «`s.ng`» matches the following English words: `sang`, `sing`, `song`, and `sung`. Note that «`.`» will match not only alphabetic characters, but also numeric and whitespace characters. Consequently, «`s.ng`» will also match non-words such as `s3ng`.

We can also use the wildcard symbol for counting characters. For instance «`....zy`» matches six-letter strings that end in `zy`. The pattern «`....berry`» finds words like `cranberry`. In our text from Wall Street Journal below, the pattern «`t...`» will match the words `that` and `term`, and will also match the word sequence `to a` (since the third "`.`" in the pattern can match the space character):

```
Paragraph 12 from wsj_0034:
It's probably worth paying a premium for funds that invest in markets
that are partially closed to foreign investors, such as South Korea,
some specialists say.  But some European funds recently have
skyrocketed; Spain Fund has surged to a startling 120% premium.  It has
been targeted by Japanese investors as a good long-term play tied to
1992's European economic integration.  And several new funds that aren't
even fully invested yet have jumped to trade at big premiums.

"I'm very alarmed to see these rich valuations," says Smith Barney's
Mr. Porter.
```

Note

Note that the wildcard matches *exactly* one character, and must be repeated for as many characters as should be matched. To match a variable number of characters we must use notation for *optionality*.

We can see exactly where a regular expression matches against a string using NLTK's `re_show` function. Readers are encouraged to use `re_show` to explore the behavior of regular expressions.

[1] These are often called *meta-characters*; that is, characters which express properties of (ordinary) characters.

```
>>> string = """
... It's probably worth paying a premium for funds that invest in markets
... that are partially closed to foreign investors, such as South Korea, ...
... """
>>> nltk.re_show('t...', string)
I{t's }probably wor{th p}aying a premium for funds {that} inves{t in} markets
{that} are par{tial}ly closed {to f}oreign inves{tors}, such as Sou{th K}orea, ...
```

1.1.2 Optionality and Repeatability

The "?" symbol indicates that the immediately preceding regular expression is optional. The regular expression «colou?r» matches both British and American spellings, colour and color. The expression that precedes the ? may be punctuation, such as an optional hyphen. For instance «e-?mail» matches both e-mail and email.

The "+" symbol indicates that the immediately preceding expression is repeatable, up to an arbitrary number of times. For example, the regular expression «coo+l» matches cool, coool, and so on. This symbol is particularly effective when combined with the . symbol. For example, «f.+f» matches all strings of length greater than two, that begin and end with the letter f (e.g. foolproof). The expression «.+ed» finds strings that potentially have the past-tense -ed suffix.

The "*" symbol indicates that the immediately preceding expression is both optional and repeatable. For example «.*gnt.*» matches all strings that contain gnt.

Occasionally we need to match material that spans a line-break. For example, we may want to strip out the HTML markup from a document. To do this we must delete material between angle brackets. The most obvious expression is: «<.*>». However, this has two problems: it will not match an HTML tag that contains a line-break, and the «.*» will consume as much material as possible (including the > character). To permit matching over a line-break we must use Python's DOTALL flag, and to ensure that the > matches against the first instance of the character we must do non-greedy matching using *?:

```
>>> text = """one two three <font
...           color="red">four</font> five"""
>>> re.sub(r'<.*?>', ' ', text, re.DOTALL)
```

1.1.3 Choices

Patterns using the wildcard symbol are very effective, but there are many instances where we want to limit the set of characters that the wildcard can match. In such cases we can use the [] notation, which enumerates the set of characters to be matched - this is called a *character class*. For example, we can match any English vowel, but no consonant, using «[aeiou]». Note that this pattern can be interpreted as saying "match a or e or ... or u"; that is, the pattern resembles the wildcard in only matching a string of length one; unlike the wildcard, it restricts the characters matched to a specific class (in this case, the vowels). Note that the order of vowels in the regular expression is insignificant, and we would have had the same result with the expression «[uoiea]». As a second example, the expression «p[aeiou]t» matches the words: pat, pet, pit, pot, and put.

We can combine the [] notation with our notation for repeatability. For example, expression «p[aeiou]+t» matches the words listed above, along with: peat, poet, and pout.

Often the choices we want to describe cannot be expressed at the level of individual characters. As discussed in the tagging tutorial, different parts of speech are often *tagged* using labels from a tagset. In the Brown tagset, for example, singular nouns have the tag NN1, while plural nouns have the tag NN2, while nouns which are unspecified for number (e.g., `aircraft`) are tagged `NN0`. So we might use «`NN.*`» as a pattern which will match any nominal tag. Now, suppose we were processing the output of a tagger to extract string of tokens corresponding to noun phrases, we might want to find all nouns (`NN.*`), adjectives (`JJ.*`) and determiners (`DT`), while excluding all other word types (e.g. verbs `VB.*`). It is possible, using a single regular expression, to search for this set of candidates using the *choice operator* "`|`" as follows: «`NN.*|JJ.*|DT`». This says: match `NN.*` *or* `JJ.*` *or* `DT`.

As another example of multi-character choices, suppose that we wanted to create a program to simplify English prose, replacing rare words (like `abode`) with a more frequent, synonymous word (like `home`). In this situation, we need to map from a potentially large set of words to an individual word. We can match the set of words using the choice operator. In the case of the word `home`, we would want to match the regular expression «`dwelling|domicile|abode`».

Note

Note that the choice operator has wide scope, so that «`123|456`» is a choice between `123` and `456`, and not between `12356` and `12456`. The latter choice must be written using parentheses: «`12(3|4)56`».

1.2 More Complex Regular Expressions

In this section we will cover operators which can be used to construct more powerful and useful regular expressions.

1.2.1 Ranges

Earlier we saw how the `[]` notation could be used to express a set of choices between individual characters. Instead of listing each character, it is also possible to express a *range* of characters, using the `-` operator. For example, «`[a-z]`» matches any lowercase letter. This allows us to avoid the over-permissive matching we noted above with the pattern «`t...`». If we were to use the pattern «`t[a-z][a-z][a-z]`», then we would no longer match the two word sequence `to a`.

As expected, ranges can be combined with other operators. For example «`[A-Z][a-z]*`» matches words that have an initial capital letter followed by any number of lowercase letters. The pattern «`20[0-4][0-9]`» matches year expressions in the range 2000 to 2049.

Ranges can be combined, e.g. «`[a-zA-Z]`» which matches any lowercase or uppercase letter. The expression «`[b-df-hj-np-tv-z]+`» matches words consisting only of consonants (e.g. `pygmy`).

1.2.2 Complementation

We just saw that the character class «`[b-df-hj-np-tv-z]+`» allows us to match sequences of consonants. However, this expression is quite cumbersome. A better alternative is to say: let's match anything which isn't a vowel. To do this, we need a way of expressing *complementation*. We do this using the symbol "`^`" as the first character inside a class expression `[]`. Let's look at an example. The regular expression « `[^aeiou]` » is just like our earlier character class «`[aeiou]`», except now the set of vowels is preceded by `^`. The expression as a whole is interpreted as matching anything which *fails* to match «`[aeiou]`». In other words, it matches all lowercase consonants (plus all uppercase letters and non-alphabetic characters).

As another example, suppose we want to match any string which is enclosed by the HTML tags for boldface, namely `<B>` and `</B>`. We might try something like this: «`<B>.*</B>`». This would successfully match `<B>important</B>`, but would also match `<B>important</B> and <B>urgent</B>`, since the «`.*`» sub-pattern will happily match all the characters from the end of `important` to the end of `urgent`. One way of ensuring that we only look at matched pairs of tags would be to use the expression «`<B>[^<]*</B>`», where the character class matches anything other than a left angle bracket.

Finally, note that character class complementation also works with ranges. Thus «`[^a-z]`» matches anything other than the lower case alphabetic characters `a` through `z`.

1.2.3 Common Special Symbols

So far, we have only looked at patterns which match with the content of character strings. However, it is also useful to be able to refer to formatting properties of texts. Two important symbols in this regard are "`^`" and "`$`" which are used to *anchor* matches to the beginnings or ends of lines in a file.

Note

"`^`" has two quite distinct uses: it is interpreted as complementation when it occurs as the first symbol within a character class, and as matching the beginning of lines when it occurs elsewhere in a pattern.

For example, suppose we wanted to find all the words that occur at the beginning of lines in the WSJ text above. Our first attempt might look like «`^[A-Za-z]+`». This says: starting at the beginning of a line, look for one or more alphabetic characters (upper or lower case), followed by a space. This will match the words `that`, `some`, `been`, and `even`. However, it fails to match `It's`, since `'` isn't an alphabetic character. A second attempt might be «`^[^ ]+`», which says to match any string starting at the beginning of a line, followed by one or more characters which are *not* the space character, followed by a space. This matches all the previous words, together with `It's`, `skyrocketed`, `1992s`, `I'm` and `"Mr.`. As a second example, «`[a-z]*s$`» will match words ending in `s` that occur at the end of a line. Finally, consider the pattern «`^$`»; this matches strings where no character occurs between the beginning and the end of a line - in other words, empty lines!

As we have seen, special characters like "`.`", "`*`", "`+`" and "`$`" give us powerful means to generalize over character strings. But suppose we wanted to match against a string which itself contains one or more special characters? An example would be the arithmetic statement `$5.00 * ($3.05 + $0.85)`. In this case, we need to resort to the so-called *escape* character "`\`" ("backslash"). For example, to match a dollar amount, we might use «`\$[1-9][0-9]*\.[0-9][0-9]`». The same goes for matching other special characters.

Special Sequences	
`\b`	Word boundary (zero width)
`\d`	Any decimal digit (equivalent to `[0-9]`)
`\D`	Any non-digit character (equivalent to `[^0-9]`)
`\s`	Any whitespace character (equivalent to `[ \t\n\r\f\v]`
`\S`	Any non-whitespace character (equivalent to `[^ \t\n\r\f\v]`)
`\w`	Any alphanumeric character (equivalent to `[a-zA-Z0-9_]`)
`\W`	Any non-alphanumeric character (equivalent to `[^a-zA-Z0-9_]`)

Table 1.1:

1.3 Python Interface

The Python re module provides a convenient interface to an underlying regular expression engine. The module allows a regular expression pattern to be compiled into a object whose methods can then be called. Let's begin by getting a list of English words:

```
>>> wordlist = nltk.corpus.words.words('en')
>>> len(wordlist)
45378
```

Now we can compile a regular expression for words containing a sequence of two 'a's and find the matches:

```
>>> r1 = re.compile('.*aa.*')
>>> [w for w in wordlist if r1.match(w)]
['Afrikaans', 'bazaar', 'bazaars', 'Canaan', 'Haag', 'Haas', 'Isaac', 'Isaacs', 'Is
```

Suppose now that we want to find all three-letter words ending in the letter "c". Our first attempt might be as follows:

```
>>> r1 = re.compile('..c')
>>> [w for w in wordlist if r1.match(w)][:10]
['accede', 'acceded', 'accedes', 'accelerate', 'accelerated', 'accelerates', 'accel
```

The problem is that we have matched words containing three-letter sequences ending in "c" which occur *anywhere within a word*. For example, the pattern will match "c" in words like aback, Aerobacter and albacore. Instead, we must revise our pattern so that it is anchored to the beginning and ends of the word: «^...$»:

```
>>> r2 = re.compile('^..c$')
>>> [w for w in wordlist if r2.match(w)]
['arc', 'Doc', 'Lac', 'Mac', 'Vic']
```

In the section on complementation, we briefly looked at the task of matching strings which were enclosed by HTML markup. Our first attempt is illustrated in the following code example, where we incorrectly match the whole string, rather than just the substring "<B>important</B>".

```
>>> html = '<B>important</B> and <B>urgent</B>'
>>> r2 = re.compile('<B>.*</B>')
>>> print r2.findall(html)
['<B>important</B> and <B>urgent</B>']
```

As we pointed out, one solution is to use a character class which matches with the complement of "<":

```
>>> r4 = re.compile('<B>[^<]*</B>')
>>> print r4.findall(html)
['<B>important</B>', '<B>urgent</B>']
```

However, there is another way of approaching this problem. «<B>.*</B>» gets the wrong results because the «*» operator tries to consume as much input as possible. That is, the matching is said to be *greedy*. In the current case, «*» matches everything after the first <B>, including the following </B> and <B>. If we instead use the non-greedy star operator «*?», we get the desired match, since «*?» tries to consume as little input as possible.

1.3.1 Exercises

2. *Pig Latin* is a simple transliteration of English: words starting with a vowel have `way` appended (e.g. `is` becomes `isway`); words beginning with a consonant have all consonants up to the first vowel moved to the end of the word, and then `ay` is appended (e.g. `start` becomes `artstay`).

   a) Write a program to convert English text to Pig Latin.

   b) Extend the program to convert text, instead of individual words.

   c) Extend it further to preserve capitalization, to keep `qu` together (i.e. so that `quiet` becomes `ietquay`), and to detect when `y` is used as a consonant (e.g. `yellow`) vs a vowel (e.g. `style`).

3. An interesting challenge for tokenization is words that have been split across a line-break. E.g. if *long-term* is split, then we have the string `long-\nterm`.

   a) Write a regular expression that identifies words that are hyphenated at a line-break. The expression will need to include the `\n` character.

   b) Use `re.sub()` to remove the `\n` character from these words.

4. Write a utility function that takes a URL as its argument, and returns the contents of the URL, with all HTML markup removed. Use `urllib.urlopen` to access the contents of the URL, e.g. `raw_contents = urllib.urlopen('http://nltk.org/').read()`.

5. Write a program to guess the number of syllables from the orthographic representation of words (e.g. English text).

6. Download some text from a language that has vowel harmony (e.g. Hungarian), extract the vowel sequences of words, and create a vowel bigram table.

7. Obtain a pronunciation lexicon, and try generating nonsense rhymes.

1.4 Further Reading

A.M. Kuchling. *Regular Expression HOWTO*, http://www.amk.ca/python/howto/regex/

About this document...