This tutorial is about programming with regular expressions (REs). Specifically, it compares "textbook" REs — the kind discussed in theoretical textbooks — to Python REs, which are based on Perl's and similar to many other programming languages'. It also gives you some idea of the extra features that Python REs have. These features make REs an efficient and concise way of solving (some) real programming problems. For example, many web applications rely heavily on text processing, and much of the text processing is done by REs.

This tutorial is far from exhaustive. Here are some decent web resources.

- `http://www.amk.ca/python/howto/regex/`
- `http://docs.python.org/library/re.html`
- `http://docs.activestate.com/komodo/4.4/regex-intro.html`

1 Textbook REs vs. Python REs

In this section we implement our textbook's REs in terms of Python's REs, assuming a simple alphabet of $\Sigma = \{\texttt{0}, \ldots, \texttt{9}, \texttt{a}, \ldots, \texttt{z}, \texttt{A}, \ldots, \texttt{Z}\}$. Enter the following Python code.

```
import re
def matches(regexp, string):
    return re.match(r'\A' + regexp + r'\Z', string) != None
```

This `matches()` function takes in two strings — the first being an RE and the second being a string to match — and outputs either `True` or `False`, indicating a match or not. The function does two things to mimic our textbook's REs. First, it wraps the given RE in `\A` and `\Z`; together these codes require the RE to match the entire string, instead of Python's default behavior of allowing matches with substrings. Second, the function simply returns `True` or `False`, instead of Python's default behavior of returning a *matching object*.

As far as the REs themselves go, the most significant difference between Python's REs and our textbook's is that Python uses `|` where the textbook uses `+`. For example,

```
matches('(ab|c|d)*|e*', 'ababcccab')
```

is equivalent to asking whether `(ab + c + d)* + e*` matches `ababcccab`. (It does.) Notice that I've omitted all unnecessary spaces from the Python RE. Include spaces only if you really mean them to be there; white space is taken seriously.

For the limited alphabet of $\Sigma = \{\texttt{0}, \ldots, \texttt{9}, \texttt{a}, \ldots, \texttt{z}, \texttt{A}, \ldots, \texttt{Z}\}$ and the limited set of operations provided by our textbook REs, this is all there is to say about Python REs. You should now be able to translate anything from the textbook into Python.

In moving beyond the textbook to serious programming problems we will find a number of other Python RE features convenient. First, using `|` for `+` frees up the `+` metasymbol to act like our book's unary `+`, meaning "repeat 1 or more times". For example,

```
matches('(a|b)+c*', 'aabaccc')
```

returns `True` because there are 1 or more `a`s and `b`s before 0 or more `c`s. Python REs also have a `?` metasymbol that indicates "repeat 0 times or 1 time, but no more", an `{n}` code that indicates "repeat exactly n times", and an `{m,n}` code that indicates "repeat between m and n times".

Another convenience is the *character class* concept. For example, `[aeiouA-C3-7]` is equivalent to `a|e|i|o|u|A|B|C|3|4|5|6|7`. We'll cover more character classes in a moment.

2 The Alphabet and the Infamous Backslashes

Python's alphabet is much larger than just `[0-9a-zA-Z]`. I'm not sure, but I think it contains all of ASCII and even all of Unicode. This means that you can access weird characters such as the newline character. You typically enter a newline into a Python string like this:

```
mystring = 'After this sentence is a newline character.\n'
```

If you really want a string containing the two characters `\` and `n`, then you have to escape the backslash with another backslash:

```
mystring = 'After this sentence are two extra characters.\\n'
```

Another solution is to make a raw Python string:

```
mystring = r'After this sentence are two extra characters.\n'
```

In short, the backslash is special metasymbol in Python strings, but you can turn off its specialness by prefacing the string with `r`.

This is handy, because the backslash is also a metasymbol in REs. We have already seen that `\A` and `\Z` in an RE indicate that the start and end of the string are to be matched. Another special code is `\w`; it is the character class of all alphanumeric characters. Similarly, `\d` matches decimal digits and `\s` matches whitespace characters. To match the backslash itself, you use `\\`. So for example the Python code

```
re.match(r'\A\d{1,2}\\\d{1,2}\\\d{2,4}\Z', s) != None
```

returns `True` for strings s such as `9\11\2001` and `08\13\76`. These are supposed to be dates; obviously I've used backslashes instead of slashes just to illustrate my point. If I hadn't used a raw Python string, I'd've needed many more backslashes in there. (Aside: I'm typing this document in LaTeX, which also uses backslash for a metasymbol...)

Now that we've discussed the alphabet we can talk about complementing character classes. The construction `[^a-g]` describes all characters other than those in `[a-g]`. For a more complicated example,

```
re.match(r'<a\s+href\s*=\s*"[^"]+">', s) != None
```

returns `True` for strings such as `<a href="www.carleton.edu">`, as long as there's at least one non-`"` character between the two `"`s.

See the Python documentation for more special characters and character classes.

3 Beyond Returning `True` and `False`

A Python RE-matching function such as `re.match()` doesn't just return whether the RE matched the string; it tells us which parts of the RE matched which parts of the string. These parts are called *groups*. They are delimited by `()`. For example,

```
re.match('g*([ab]*)g*', 'gggabaabbbgabaaaa').groups()
```

returns `('abaabbb',)`. Why? The RE matches just the substring `gggabaabbbg`; by default it doesn't insist on matching all the way to the end of the input string. In this substring the part corresponding to the group `([ab]*)` was `abaabbb`, so that was returned in the length-1 tuple of results.

A sophisticated RE can contain multiple groups to extract multiple parts of the string; see the documentation. You can even refer to groups within the regular expression, using the codes `\1`, `\2`, etc. For example,

```
re.match(r'g+([ab]+)g+\1g+', 'gggabaabbbgabaabbbgggg').groups()
```

returns `('abaabbb',)`. Why? The group matches `abaabbb`, and this same substring is matched again by the `\1`, so the whole RE matches. But only the group delimited by parentheses is returned.

If you want to find all non-ovelapping substrings that match a given regular expression, try `re.findall()`. For example, the following code returns all URLs from a given string of HTML. Notice how the entire HTML anchor element is matched, but only the URL within that element is returned, because only it is in a group. (This regular expression could be improved to be case-insensitive.)

```
re.findall(r'<a\s+href\s*=\s*"([^"]+)">', htmlString)
```

Python strings come with a rudimentary `split()` method, but the RE library's `split()` lets you split according to any regular expression. In this simple example we split a string according to white space.

```
>>> re.split(r'\s+', 'The lazy brown dog\nate\tthe quick silver fox.')
['The', 'lazy', 'brown', 'dog', 'ate', 'the', 'quick', 'silver', 'fox.']
```

You can also use regular expressions to alter the contents of strings. Here's a simple search-and-replace.

```
>>> re.sub(r'French', 'freedom', 'These French fries are delicious.')
'These freedom fries are delicious.'
```

This tutorial is just a start; there's a lot more to learn about programming with REs. See the online references/tutorials.