This tutorial is about programming with regular expressions. Specifically, it compares "textbook" regular expressions (TREs) to Python regular expressions (PREs), which are similar to those used in Java, Perl, and many other programming languages. The tutorial also gives you some idea of the extra features that PREs have. These features make regular expressions 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 regular expressions. This tutorial is far from exhaustive. Here are some solid web resources.

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

1 TREs in Python

In this section we implement TREs in terms of PREs. There are three big differences between the two systems, of which you must be aware.

First, a PRE doesn't just return "match" or "not match"; it produces *match object* that describes *how* it matches. Second, by default a PRE doesn't have to match the whole string; matches to substrings are recorded as matches. Let's make a wrapper function to hide these differences. Enter the following code into the Python interpreter or a program file.

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

This `matches()` function takes in two strings — the first being a PRE 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 TRE conventions. First, it wraps the given PRE in the special codes `\A` and `\Z`, which together force matches to whole strings, not just substrings. Second, the function simply returns `True` or `False`, instead of a match object.

The third difference between TREs and PREs is in the regular expression syntax itself: Because there is no $\cup$ key on the keyboard, PREs use `|` instead of $\cup$. For example, the TRE $(ab \cup c \cup d)^+ \cup e^*$ is entered into Python as the string `(ab|c|d)+|e*`. (White space inside PREs is taken seriously. Don't put it there unless you really want it there.)

You should now be able to translate anything from the textbook into Python. For example, to test whether $(a \cup b)^+c^*$ matches *ababcccab*, enter

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

2 Shortcuts and Special Characters

In moving beyond the textbook to programming problems, we will find a number of shortcuts helpful. In addition to the `+` operator, which means "repeat 1 or more times", PREs offer several similar shortcuts:

- `?` means "repeat 0 times or 1 time, but no more".
- `{n}` means "repeat exactly n times".
- `{m,n}` means "repeat between m and n times".

For example, the PRE `a(bc){2,3}` matches `abcbc` but not `abc` or `abcabc`.

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`. The PRE `[0-9a-zA-Z]` matches exactly the alphanumeric characters.

PREs include a limited complementation operation, in that character classes can be complemented using `^`. For example, `[^a-g]` matches all characters other than those in `[a-g]`. For a more useful example, `"[^"]+"` matches strings of length at least 3 that begin and end with `"` and contain no `"` in between. Perhaps you can see how this kind of PRE would be used to pick out string literals in a computer program.

The characters `.`, `^`, `$`, `*`, `+`, `?`, `{`, `}`, `\`, `[`, `]`, `|`, `(`, `)` all have special meanings in PREs. So, if you want your PRE to match any of these characters, you have to escape them with a backslash. For example, to match single-word sentences, you might use the PRE `[A-Z][a-z]*(\.|\?|!)`. There are also various special character sequences beginning with `?`. See the Python documentation for more detail.

3 Backslashes

The alphabet available to PREs 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 characters: backslash and n.\\n'
```

Another solution is to make a *raw* Python string:

```
>>> mystring = r'After this sentence are two characters: backslash and n.\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 PREs. We have already seen that `\A` and `\Z` in a PRE indicate that the match must occur at the start and end of the string. 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 command

```
>>> matches(r'\d{1,2}\\\d{1,2}\\\d{2,4}', s)
```

returns `True` for strings `s` such as `9\11\2001` and `08\13\76`. These are supposed to be dates, which are usually written with slashes instead of backslashes. I've used backslashes just to illustrate my point. And my point is that using a raw string saves me from having to enter this craziness:

```
>>> matches('\\d{1,2}\\\\\\d{1,2}\\\\\\d{2,4}', s)
```

(By the way, I'm typing this document in LaTeX, which also uses backslash for a metasymbol...) I almost always use raw strings to enter PREs into Python.

4 Groups

PRE functions such as `re.search()` don't just return whether the given regular expression matched the given string. They tell us which parts of the regular expression matched which parts of the string. These parts are called *groups*. They are delimited by `()`. For example,

```
>>> re.search(r'g*([ab]*)g*', 'gggabaabbbgabaaaa').groups()
('abaabbb',)
```

Why did this result occur? Well, first the PRE matches the substring `gggabaabbbg`. (We haven't included the `\A` and `\Z` codes that force the whole string to match.) 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 PRE can contain multiple groups to extract multiple parts of the string. In the following example, there are three groups. We ask the match object for a particular group. We also ask it which indices in the string produced the substring that the group matched.

```
>>> mo = re.search(r'(ab*)c((de+)f)', r'abbbbcdef')
>>> mo.groups()
```

```
('abbbb', 'def', 'de')
>>> mo.group(1)
'abbbb'
>>> mo.span(1)
(0, 5)
```

Sometimes you want to place parentheses into a PRE to express precedence, without declaring that a group should be captured. PREs offer a simple way to make parentheses non-capturing: Instead of `(...)`, you enter `(?:...)`. For example, compare this transcript to the similar transcript above.

```
>>> re.search(r'(ab*)c(?:(de+)f)', r'abbbbcdef').groups()
('abbbb', 'de')
```

You can even refer to groups within the PRE, using the codes `\1`, `\2`, etc. For example,

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

Why did this happen? The group matches `abaabbb`, and this same substring is matched again by the `\1`, so the whole PRE matches. And then the match object reports the string matched by the group. Carefully compare that example to the following examples.

```
>>> re.search(r'g+([ab]+)g+\1g+', 'gggabaabbbgabaabbgggg').groups()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'groups'
>>> re.search(r'g+([ab]+)g+([ab]+)g+', 'gggabaabbbgabaabbbgggg').groups()
('abaabbb', 'abaabbb')
>>> re.search(r'g+([ab]+)g+([ab]+)g+', 'gggabaabbbgabaabbgggg').groups()
('abaabbb', 'abaabb')
```

5 Other utility functions

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 handle more real-world cases. For example, it should be made case-insensitive.)

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

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

```
>>> re.split(r'(\.|\?|!)', 'Corn is great! Must the onions be carameliz
ed? Johnny Depp smiled at my houseplants.')
['Corn is great', '!', ' Must the onions be caramelized', '?', ' Johnny
 Depp smiled at my houseplants', '.', '']
```

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

```
>>> re.sub(r'clever', 'not too bright', 'The professor is clever.')
'The professor is not too bright.'
```

I've written this tutorial just to convey the essentials of PREs, so that we can do practical regular expression exercises in CS 254. There's a lot more to learn about programming with regular expressions. See the online references/tutorials.