Regular Expressions

Regular expressions are a small programming language over strings that allow us to succinctly and compactly represent classes of strings. This is particularly useful in pattern matching where we wish to find the occurrence of patterns in a string. We represent regular expressions as *strings of symbols*. Some of the symbols are characters in an alphabet and some serve as operators. For example, the string abc is a regular expression representing the class of strings containing the single string abc. The regular expression ab^*c represents the class of strings that start with an a, end with a c and contain 0 or more b characters in between. The star is actually called the *Kleene star* and its typical usage is in automata and language theory where, when applied to a set, means *take the cartesian product of the set with itself 0 or more times*.

Regular Expressions in Python

The Theory of Computation defines regular expressions in a rigorous mathematical framework. Here we will work with them more informally and simply treat them as a string of symbols where some symbols act as operators. We will also focus on regular expressions as they appear in Python. This knowledge should transfer well to other programming languages.

Let's start with an example.

Example 1. *Suppose we have some text* `t` *and a we want to search it to find any pattern that starts with an arbitrary number of as, followed by a b, followed by an arbitrary number of cs. In regular expression land this is just* `a*bc*`*. So if* `t=''dog abc cat aabcc bird aac''` *then* `a*bc*` *matches* `t` *at* `abc`*,* `aabcc`*, and* `b`*.*

Escaping

The following characters carry special meaning in regular expressions:

```
. ^ $ * + ? { } [ ] \ | ( )
```

To use one of these characters, we have to escape it with the \ character. For example suppose you wanted to match the string `$100.00`. The correct regular expression is `\$100\.00` and expressed as a Python string it would be `'\$100\.00'`. Now, suppose that you wanted match the expression `\\`. Because Python uses the backslash character as an escape character we must write the string `'\\\\\\\\''` to escape our escaped string. This is ridiculous, so Python supports the idea of a *raw* string literal where escaping does not occur. A raw string can be prefaced with an `r` character and we can write `r'\\\\'`.

Blocks and Dot

One piece of notational convenience provided Python is grouping a set of characters together into a common class. For example, to match any of the characters `abc...xyz` we could write `[a-z]`. There are shorthands for many groups. For example

- `\d` matches `[0-9]`,
- `\s` matches any white space character `[ \t\n\r\f\v]`, and
- `\w` matches any alphanumeric character `[a-zA-Z0-9_]`

We also mention the dot character `.`, which matches any character except the newline character.

Greedy Matching of Star

Suppose we have the regular expression `do[sgn]*s` and the text `dogsgs`. The matching algorithm matches `d` and then `o`. Then it matches `gsgs`. It tries to match the final `s` but fails, so it backs one location and successfully matches the `s`.

If one prefers that the algorithm match minimally, then you can qualify the star with a question mark. For example the regular expression `do[sgn]*?s` matches the text `dogsgs` on `dogs`.

Finding Patterns in Text

Python provides four primary methods to search text for patterns expressed as regular expressions.

`match` checks if the regular expression matches at the *beginning* of the text;

`search` searches all location of the text for the first occurence of a pattern;

`findall` finds all the occurences of the pattern within the text and returns them as a list;

`finditer` finds all the locations of of the pattern within the text and returns an iterator.

All the methods except for `findall` return `match` objects, which encapsulate information about where the regular expression matches the given text. Furthermore, the `match` and `search` methods only match the *first* match. Let's suppose we are interested in finding the domain name of a bunch of web addresses using the methods described above.

For example, suppose that we have the following list

```
www = ["http://math.williams.edu/best-jobs-2015/",
             "http://www.williams.edu/registrar",
              "http://magazine.williams.edu/2015/spring/study/the-body-as-book/"]
```

`match`

The `match` method will tell us if a pattern occurs at the beginning of a string, in which case it returns a `Match` object, or it returns `None`.

```
>>> re.match('http://', www[0])
<_sre.SRE_Match object; span=(0, 7), match='http://'>
>>> re.match('www', w[0])
>>>
```

Notice that the `Match` object encapsulates the beginning and ending positions of the match as well as the match text.

`search`

Like `match` but searches the entire string for a match.

`findall`

Returns a list of all matching strings in the original text.

`finditer`

Like `search` but returns an iterator over all matches in the string. This method returns a `Match` object.

Groups

Groups allow you to isolate text inside a larger matched pattern. For example, suppose we wanted to pull our just the domain name from the list of web addresses we have above.

```
>>> [re.match("http://(.*?)/",w).group(1) for w in www]
['math.williams.edu', 'www.williams.edu', 'magazine.williams.edu']
```

Notice that we used `group(1)` here—`group(0)` refers to the entire match—and each subsequent index refers to the next open parentheses. For example, suppose we wanted to match two things—the subdomain along with the domain.

```
>>> [re.match("http://(.*?)\.(.*?)/",w).group(0) for w in www]
['http://math.williams.edu/',
 'http://www.williams.edu/',
 'http://magazine.williams.edu/']
>>> [re.match("http://(.*?)\.(.*?)/",w).group(1) for w in www]
['math', 'www', 'magazine']
>>> [re.match("http://(.*?)\.(.*?)/",w).group(2) for w in www]
['williams.edu', 'williams.edu', 'williams.edu']
>>>
```

Other Regular Expression Syntax

Here are some other regular expression syntactical structures along with their semantics (i.e. symbols and their meaning).

? means the previous character in the regular expression is optional. For example `0?01` matches `001` and `01`. Remember that `?` following a `*` (or a +) means be minimally greedy in the match.

{m} means match exact `m` copies of the previous character.

{m,n} means match between `m` and `n` characters. The final `n` may be committed (but the comma must remain) to give a lower bound on the number of characters. One can also append the `?` to this (e.g., `{3,5}?`) to minimally match the requirement.

^ is used to preface part of a pattern that only matches at the start of the text.

$ is used to indicate that a pattern should reach the end of the text.