LING115 Lecture Note
Session #7: Regular Expressions

1. Introduction

We need to refer to a set of strings for various reasons: to ignore case-distinction, to refer to a set of files that share a common substring, etc. A regular expression is a formal notation that refers to a set of strings. For example, the regular expression `'[Ll]inguistics'` is equivalent to {Linguistics, linguistics}. The idea is that it is much more convenient to use regular expressions than to enumerate all the strings, if possible.

Formal systems like regular expressions depend on what the users of the system agree on or its *conventions*: what the individual symbols mean, what the operators that attach to the symbols mean, etc. There are a small number of different kinds of regular expressions that follow different conventions: POSIX-basic, POSIX-extended, Perl-derivative, etc. We will learn the Perl-derivative regular expressions in this class.

2. Basics

In its simplest form, a regular expression is a single character like `'a'`, which basically refers to the string `a`. There are three basic ways a regular expression can become more complex from this: concatenation, disjunction, and Kleene-star.

2.1. Concatenation

We can concatenate regular expressions to derive a regular expression that refers to a longer string. For example, concatenating two regular expressions `'a'` and `'b'` will result in `'ab'`, which now refers to the string `ab`. We can, of course, concatenate `'ab'` and `'c'` to get `'abc'`, which refers to `abc`.

2.2. Disjunction

We can combine two or more regular expressions by disjunction to refer to a larger set of strings. Each regular expression refers to a set of strings. Disjunction of two or more regular expressions refers to the union of sets of strings, where each set is a set of strings specified by each regular expression.

There are two ways to expression disjunction: `|` and `[]`. Two regular expressions conjoined by `|` refer to the union of two sets each represented by a regular expression. For example, `'regular|expression'` means either `regular` or `expression`. On the other hand, `[]` is used to refer to the any one of the characters inside the brackets. For example, `'[regular]'` is equal to `{r, e, g, u, l, a}`.

2.3. Kleene-star

It helps to specify a repetition of characters or strings. In regular expression, this is done by the Kleene star: zero or more instances of what immediately comes before the star. For example, `regular*` is equal to `{regula, regular, regularr, regularrr, ...}`.

2.4. Scope

`|` and `*` are operators. Regular expression becomes more powerful if we control the scope of these operators. We use the parentheses to identify the scope. For example, the parentheses in `(regular)*` means that the Kleene star applies to regular, not just `r`. So the regular expression means zero or more repetitions of `regular`. Similarly, `reg(ular|ister)` means that the disjunction operator applies to `ular` and `ister`, rather than `regular` and `ister`. So the regular expression means either `regular` or `register`.

3. Special characters

There are special characters which allow us to use regular expression more easily. Below is a list of useful characters.

Character	Meaning	Example
`.`	Any single character	`'.*'` = zero or more repetitions of any character
`+`	One or more instances of preceding character	`'regular+'` = {regular, regu larr, regularrr, ...}
`?`	Zero or one instance of the preceding character	`'linguistics?'` = {linguistic,linguistics}
`-`	Range of characters when used with characters in brackets	`'[a-z]'` = {a,b,c,d,...,z}
`{m,n}`	from m through n instances of preceding character	`'reguar{2,4}'` = {regularr, regularrr, regularrr}
`{m,}`	m or more instances of preceding character	`'regualar{2,}'` = {‘regularr’, ‘regularrr’, ...}
`{,n}`	up to n instances of preceding character	`'regulaar{,3}'` = {regula, regular, regularr, regularrr}
`{m}`	exactly m instances of preceding character	`'regualar{2}'` = ‘regularr’
`^`	Beginning of a line	`'^ab'` = ab where a begins a line
`^`	Any character except the characters listed in brackets. This meaning takes effect only when ^ is the first character in brackets.	`'[^abc]'` = any character but {a,b,c}
`$`	End of a line	`'ab$'` = ab where b ends a line
`\d`	Any digit	`'\d{2}'` = {00,01,...,99}
`\D`	Any non-digit character	`'\D'` = `'[^0-9]'`
`\w`	Any alphanumeric character	`'\w'` = `'[a-zA-Z0-9_]'`

	or underscore	
`\W`	Any character but \w	`'\W' = '[^a-zA-Z0-9_]'`

In order to refer to these special characters literally in a regular expression, we add a backslash before the characters. For example, `'\.'` would literally refer to the dot character, rather than any single character.

4. Python `re` module

Python provides a module named `re` with various functions for manipulating regular expressions: searching text for that match a regular expression, substituting a string for a string that matches the regular expression, etc. Keep in mind that as these functions are defined in the `re` module, we must first `import re` before we use them.

4.1. Matching

There are several functions for matching strings against the regular expression we specify. Keep in mind that in all of the functions here, we specify the regular expression as strings. We will frequently use the following three functions:

```
re.match(regex,string)
```

Match regular expression at the beginning of the string. If the two match, the function returns a regular-expression object. Otherwise, it returns `None`.

```
>>> re.match('ab+','abc')
>>> re.match('bc','abc')
```

```
re.search(regex,string)
```

Scan through the string and look for a substring that matches the regular expression. If the two match, the function returns a regular-expression object. Otherwise, it returns `None`.

```
>>> re.search('bc','abc')
```

```
re.findall(regex,string)
```

Find all substrings of the string that match the regular expression. The function returns a list of all matching substrings.

```
>>> re.findall('bc','abcdcbc')
```

4.2. Substitution

We can rewrite parts of a string that match a regular expression as some other string.

```
re.sub(regex,replacement,string)
```

Replace parts of string that match `regex` by replacement.

```
>>> re.sub('ab','xy','sabcdabdw')
```

4.3. Regular expression objects and grouping

Recall that `re.match()` and `re.search()` return a regular expression object as the result of matching. Similar to other data-types we have seen, the `re` module provides several methods specific to the regular expression objects. Among those methods, there are a few that are related to grouping the part of the string that matches the regular expression.

`group()`

Return the string that matches the regular expression. Again, the output of `re.match()` or `re.search()` is a regular expression object, which doesn't tell us much by itself. For easier interpretation, it can be quite useful to read the result of matching as a string.

```
>>> vowel=re.search('[aeiou]','linguistics')
>>> vowel.group()
```

`start()`

Return the staring index of the part of the string that matches the regular expression.

```
>>> vowel=re.search('[aeiou]','linguistics')
>>> vowel.start()
```

`end()`

Return the end index of the part of the string that matches the regular expression.

```
>>> vowel=re.search('[aeiou]','linguistics')
>>> vowel.end()
```

`span()`

Return a tuple consisting of the starting index and the end index of the part of the string that matches the regular expression.

```
>>> vowel=re.search('[aeiou]','linguistics')
>>> vowel.span()
```

4.4. Greedy vs. Non-greedy

Regular expression functions in Python are greedy in the sense that they look for longest possible substring that matches the regular expression. Consider the following, for example:

```
>>> line='<a href="http://www.sjsu.edu">Welcome to SJSU</a>'
>>> re.sub('<.+>','',line)
```

Running the re.sub function will completely delete the line. If we wanted to only remove the first html tag, for example, we would need to make `re.sub()` less greedy and stop scanning the line as soon as it finds the first >. In Python, we use `?` to make the regular expression functions non-greedy. For example,

```
>>> re.sub('<.+?>','',line>
```