The Matching Characters

When you want to match a character in a string, in most cases you can just use that character or that sub-string. So if we wanted to match "dog", then we would use the letters **dog**. Of course, there are some characters that are reserved for regular expressions. These are known as *metacharacters*. The following is a complete list of the metacharacters that Python's regular expression implementation supports:

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

Let's spend a few moments looking at how some of these work. One of the most common pairs of metacharacters you will encounter are the square braces: [and]. There are used for creating a "character class", which is a set of characters that you would like to match on. You may list the characters individually like this: **[xyz]**. This will match any of the characters listed between the braces. You can also use a dash to express a range of characters to match against: **[a-g]**. In this example, we would match against any of the letters a through g.

To actually do a search though, we would need to add a beginning character to look for and an ending character. To make this easier, we can use the asterisk which allows repetitions. Instead of matching *, the * will tell the regular expression that the previous character may be matched zero or more times.

It always helps to see a simple example:

```
'a[b-f]*f
```

This regular expression pattern means that we are going to look for the letter

a, zero or more letters from our class, [b-f] and it needs to end with an *f*. Let's try using this expression in Python:

```
import re
text = 'abcdfghijk'
parser = re.search('a[b-f]*f', text)
#<_sre.SRE_Match object; span=(0, 5), match='abcdf'>

parser.group()
#'abcdf'
```


```
import re
text = 'abcdfghijk'
parser = re.search('a[b-f]*f', text)
print (parser)
#<_sre.SRE_Match object; span=(0, 5), match='abcdf'>

print (parser.group())
#'abcdf'
```


Basically this expression will look at the entire string we pass to it, which in this case is **abcdfghijk**. It will find the *a* at the beginning match against that. Then because it has a character class with an asterisk on the end, it will actually read in the rest of the string to see if it matches. If it doesn't, them it will backtrack one character at a time attempting to find a match.

All this magic happens when we call the **re** module's **search** function. If we don't find a match, then **None** is returned. Otherwise, we will get a **Match** object, which you can see above. To actually see what the match looks like, you need to call the **group** method.

There's another repeating metacharacter which is similar to *. It is +, which will match one or more times. This is a subtle difference from * which matches **zero** or more times. The + requires at least one occurrence of the character it is looking for.

The last two repeating metacharacters work a bit differently. There is the

question mark, ?, which will match either once or zero times. It sort of marks the character before it as optional. A simple example would be “co-?op”. This would match both “coop” and “co-op”.

The final repeated metacharacter is {a,b} where a and b are decimal integers. What this means is that there must be at least a repetitions and at most b. You might want to try out something like this:

```
xb{1,4}z
```

This is a pretty silly example, but what it says is that we will match things like **xbz**, **xbbz**, **xbbbz** and **xbbbbz**, but not **xz** because it doesn’t have a “b”.

The next metacharacter that we’ll learn about is ^. This character will allow us to match the characters that are not listed inside our class. In other words, it will complement our class. This will only work if we actually put the ^ inside our class. If it’s outside the class, then we will be attempting to actually match against ^. A good example would be something like this: [^a]. This will match any character except the letter ‘a’.

The ^ is also used as an anchor in that it is usually used for matches at the beginning of string. There is a corresponding anchor for the end of the string, which is *$*.

We’ve spent a lot of time introducing various concepts of regular expressions. In the next few sections, we’ll dig into some more real code examples!