Regex

Metacharater

```
[] >> Used to specify character classes
[abc] = [a-c] >> will match any characters a,b,c
[a-z] >> will match lowercase letters

^ >> used to get the complementing set (caret key)
[^5] >> will match any character EXCEPT 5

\ >> the most important metacharacter (backslash)
\ >> used to escape other metacharacter

# Special sequences which begin with a backslash that represent predefined sets of characters.
\d = [0-9]                      >> special sequences for decimal digits
\D = [^0-9]                     >> special sequences for non-decimal digits
\s = spaces (tabs, newlines)    >> will match any white space character
\S = non-spaces                 >> will match any non-space character
\w = [a-zA-Z0-9_]               >> will match any alphanumeric character (word character)
\W = [^a-zA-Z0-9_]              >> will match any non-alphanumeric character (non-word character)

# \ can be combined in sequences
[\s.,]                          >> will match whitespace, period, and the comma


# Other uses for metacharaters
^ >> match patterns at the beginning of the sequence (alternative use of caret key ^)
^ >> When at the beginning of a sqeuence, it denotes the pattern should be matched at the beginning of the line
$ >> match patterns at the end of the sequence
. >> match anything EXCEPT the newline character

# Quatifier
*     >> specifies the previous character can be matched ZERO or MORE times
?     >> will match patterns with the previous character available ZERO or ONE time
+     >> will match patterns with the previous character apperating ONE or MORE times
{m,n} >> specify the count of the previous character that we want to use

ca*t = ct, cat, caat, caaat, etc...
ca?t = ct, cat ... not caat
ca{3,5}t = caaat, caaaat, caaaaat ... not ct, cat, caat

# special operator: "or"
|     >> or

# Grouping the components together: ()
# Grouping can dissect string and divides them into subgroups which match different components
# Denote group using parentheses ()
(ab)* = ab, abab, ababab, etc...  >> We can match ab, zero or multiple times
```

Pattern Example

```
vowels = '[aeiou]'
two_or_more_vowels = vowels + '{2,}'   # Result: [aeiou]{2,}
```

```
# special operator: "or"
adjectives = "lazy|brown"
print(f"Scanning `{input_string}` for adjectives, `{adjectives}`:")
for match_adjective in re.finditer(adjectives, input_string):
    print(match_adjective)

# Result:
Scanning `The quick brown fox jumps over the lazy dog` for adjectives, `lazy|brown`:
<re.Match object; span=(10, 15), match='brown'>
<re.Match object; span=(35, 39), match='lazy'>
```

```
# Predefined character classes
three_digits = '\d\d\d'
print(re.findall(three_digits, "My number is 555-123-4567"))

# Result:
['555', '123', '456']
```

SSN Pattern

```
Recall
1. \d = [0-9]
2. use {m,n} to denote the count
3. ^ corresponds with it starts with
4. $ corresponds with it ends with

# SSN pattern
^\d{3}-\d{2}-\d{4}$

# Use group to separate the digits
^(\d{3})-(\d{2})-(\d{4})$
```

Regular Expressions

A *regular expression* is a specially formatted pattern, written as a string. Matching patterns with regular expressions has 3 steps:

1. You come up with a **pattern** to find.
2. You **compile** it into a ***pattern object***

   ```
   pattern_object = re.compile(pattern)
   ```

3. You apply the pattern object to a string to **find *matches***, i.e., instances of the pattern within the string.

   ```
   matches = pattern_object.search(input)
   ```

4. Query matches for more information.

   ```
   print(matches.group())
   print(matches.start())
   print(matches.end())
   print(matches.span())
   ```

Example.

```
# 1.come up with a pattern
pattern = 'fox'

# 2.compile it into a pattern object
pattern_matcher = re.compile(pattern)

# 3.apply the pattern object to a string to find matches
input_string = 'The quick brown fox jumps over the lazy dog'
matches = pattern_matcher.search(input_string)
print(matches)

# 4.Query matches for more information.
print(matches.group())
print(matches.start())
print(matches.end())
print(matches.span())
```

```
# Result:
<re.Match object; span=(16, 19), match='fox'>

fox
16
19
(16, 19)
```

Module-level searching. For infrequently used patterns, you can also skip creating the pattern object and just call the module-level search function, `re.search()`.

```
matches_2 = re.search('jump', input_string)
assert matches_2 is not None
print ("Found", matches_2.group(), "@", matches_2.span())
```

Other Search Methods. Besides `search()`, there are several other pattern-matching procedures:

1. `match()` - Determine if the regular expression (RE) matches at the beginning of the string.
2. `search()` - Scan through a string, looking for any location where this RE matches.
3. `findall()` - Find all substrings where the RE matches, and returns them as a list.
4. `finditer()` - Find all substrings where the RE matches, and returns them as an iterator.