Overview of Regular Expressions

Regular Expressions (sometimes called regex for short) allows a user to search for strings using almost any sort of rule they can come up. For example, finding all capital letters in a string, or finding a phone number in a document.

Regular expressions are notorious for their seemingly strange syntax. This strange syntax is a byproduct of their flexibility. Regular expressions have to be able to filter out any string pattern you can imagine, which is why they have a complex string pattern format.

Let's begin by explaining how to search for basic patterns in a string!

Searching for Basic Patterns

Let's imagine that we have the following string:

```
In [1]: text = "The person's phone number is 408-555-1234. Call soon!"
```

We'll start off by trying to find out if the string "phone" is inside the text string. Now we could quickly do this with:

```
In [2]: 'phone' in text

Out[2]: True
```

But let's show the format for regular expressions, because later on we will be searching for patterns that won't have such a simple solution.

```
In [3]: import re
```

```
In [4]: pattern = 'phone'
```

```
In [5]: re.search(pattern,text)

Out[5]: <_sre.SRE_Match object; span=(13, 18), match='phone'>
```

```
In [6]: pattern = "NOT IN TEXT"
```

```
In [7]: re.search(pattern,text)
```

Now we've seen that re.search() will take the pattern, scan the text, and then returns a Match object. If no pattern is found, a None is returned (in Jupyter Notebook this just means that nothing is output below the cell).

Let's take a closer look at this Match object.

```
In [8]: pattern = 'phone'
```

```
In [9]: match = re.search(pattern,text)
```

```
In [10]: match
```

```
Out[10]: <_sre.SRE_Match object; span=(13, 18), match='phone'>
```

Notice the span, there is also a start and end index information.

```
In [11]: match.span()
```

```
Out[11]: (13, 18)
```

```
In [12]: match.start()
```

```
Out[12]: 13
```

```
In [13]: match.end()
```

```
Out[13]: 18
```

But what if the pattern occurs more than once?

```
In [14]: text = "my phone is a new phone"
```

```
In [15]: match = re.search("phone",text)
```

```
In [16]: match.span()
```

```
Out[16]: (3, 8)
```

Notice it only matches the first instance. If we wanted a list of all matches, we can use .findall() method:

```
In [17]: matches = re.findall("phone",text)
```

```
In [18]: matches
```

```
Out[18]: ['phone', 'phone']
```

```
In [19]: len(matches)
```

```
Out[19]: 2
```

To get actual match objects, use the iterator:

```
In [20]: for match in re.finditer("phone",text):
             print(match.span())
```

```
(3, 8)
(18, 23)
```

If you wanted the actual text that matched, you can use the .group() method.

In [21]:
```python
match.group()
```

Out[21]:
```
'phone'
```

Patterns

So far we've learned how to search for a basic string. What about more complex examples? Such as trying to find a telephone number in a large string of text? Or an email address?

We could just use search method if we know the exact phone or email, but what if we don't know it? We may know the general format, and we can use that along with regular expressions to search the document for strings that match a particular pattern.

This is where the syntax may appear strange at first, but take your time with this, often its just a matter of looking up the pattern code.

Let' begin!

Identifiers for Characters in Patterns

Characters such as a digit or a single string have different codes that represent them. You can use these to build up a pattern string. Notice how these make heavy use of the backwards slash \ . Because of this when defining a pattern string for regular expression we use the format:

```
r'mypattern'
```

placing the r in front of the string allows python to understand that the \ in the pattern string are not meant to be escape slashes.

Below you can find a table of all the possible identifiers:

</table>

For example:

In [22]:
```python
text = "My telephone number is 408-555-1234"
```

In [23]:
```python
phone = re.search(r'\d\d\d-\d\d\d-\d\d\d\d',text)
```

In [24]:
```python
phone.group()
```

Out[24]:
```
'408-555-1234'
```

Notice the repetition of \d. That is a bit of an annoyance, especially if we are looking for very long strings of numbers. Let's explore the possible quantifiers.

Quantifiers

Now that we know the special character designations, we can use them along with quantifiers to define how many we expect.

Character	Description	Example Pattern Code	Exammple Match
\d	A digit	file_\d\d	file_25
\w	Alphanumeric	\w-\w\w\w	A-b_1
\s	White space	a\sb\sc	a b c
\D	A non digit	\D\D\D	ABC
\W	Non-alphanumeric	\W\W\W\W\W	*-+=)
\S	Non-whitespace	\S\S\S\S	Yoyo

</table>

Let's rewrite our pattern using these quantifiers:

In [25]:
```
re.search(r'\d{3}-\d{3}-\d{4}',text)
```

Out[25]:
```
<_sre.SRE_Match object; span=(23, 35), match='408-555-1234'>
```

Groups

What if we wanted to do two tasks, find phone numbers, but also be able to quickly extract their area code (the first three digits). We can use groups for any general task that involves grouping together regular expressions (so that we can later break them down).

Using the phone number example, we can separate groups of regular expressions using parenthesis:

In [26]:
```
phone_pattern = re.compile(r'(\d{3})-(\d{3})-(\d{4})')
```

In [27]:
```
results = re.search(phone_pattern,text)
```

In [28]:
```
# The entire result
results.group()
```

Out[28]:
```
'408-555-1234'
```

In [29]:
```
# Can then also call by group position.
# remember groups were separated by parenthesis ()
# Something to note is that group ordering starts at 1. Passing in 0 returns
everything
results.group(1)
```

Out[29]:
```
'408'
```

In [30]:
```
results.group(2)
```
Out[30]:
```
'555'
```
In [31]:
```
results.group(3)
```
Out[31]:
```
'1234'
```
In [32]:
```
# We only had three groups of parenthesis
results.group(4)
```
```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call la
st)
<ipython-input-32-866de7a94a57> in <module>()
      1 # We only had three groups of parenthesis
----> 2 results.group(4)

IndexError: no such group
```

Additional Regex Syntax

Or operator |

Use the pipe operator to have an **or** statment. For example

In [33]:
```
re.search(r"man|woman","This man was here.")
```
Out[33]:
```
<_sre.SRE_Match object; span=(5, 8), match='man'>
```
In [34]:
```
re.search(r"man|woman","This woman was here.")
```
Out[34]:
```
<_sre.SRE_Match object; span=(5, 10), match='woman'>
```

The Wildcard Character

Use a "wildcard" as a placement that will match any character placed there. You can use a simple period **.** for this. For example:

In [35]:
```
re.findall(r".at","The cat in the hat sat here.")
```
Out[35]:
```
['cat', 'hat', 'sat']
```
In [36]:
```
re.findall(r".at","The bat went splat")
```
Out[36]:
```
['bat', 'lat']
```

Notice how we only matched the first 3 letters, that is because we need a **.** for each wildcard letter. Or use the quantifiers described above to set its own rules.

In [37]:
```
re.findall(r"...at","The bat went splat")
```

```
Out[37]: ['e bat', 'splat']
```

However this still leads the problem to grabbing more beforehand. Really we only want words that end with "at".

```
In [38]: # One or more non-whitespace that ends with 'at'
         re.findall(r'\S+at',"The bat went splat")
```

```
Out[38]: ['bat', 'splat']
```

Starts with and Ends With

We can use the **^** to signal starts with, and the **$** to signal ends with:

```
In [39]: # Ends with a number
         re.findall(r'\d$','This ends with a number 2')
```

```
Out[39]: ['2']
```

```
In [40]: # Starts with a number
         re.findall(r'^\d','1 is the loneliest number.')
```

```
Out[40]: ['1']
```

Note that this is for the entire string, not individual words!

Exclusion

To exclude characters, we can use the **^** symbol in conjunction with a set of brackets **[]**. Anything inside the brackets is excluded. For example:

```
In [41]: phrase = "there are 3 numbers 34 inside 5 this sentence."
```

```
In [42]: re.findall(r'[^\d]',phrase)
```

```
Out[42]: ['t',
          'h',
          'e',
          'r',
          'e',
          ' ',
          'a',
          'r',
          'e',
          ' ',
          ' ',
          'n',
          'u',
          'm',
          'b',
          'e',
          'r',
          's',
          ' ',
          ' ',
          'i',
          'n',
          's',
          'i',
          'd',
          'e',
          ' ',
          ' ',
          't',
          'h',
          'i',
          's',
          ' ',
          's',
          'e',
          'n',
          't',
          'e',
          'n',
          'c',
          'e',
          '.']
```

To get the words back together, use a + sign

```
In [43]: re.findall(r'[^\d]+',phrase)
```

```
Out[43]: ['there are ', ' numbers ', ' inside ', ' this sentence.']
```

We can use this to remove punctuation from a sentence.

```
In [44]: test_phrase = 'This is a string! But it has punctuation. How can we remove
         it?'
```

```
In [45]: re.findall('[^!.? ]+',test_phrase)
```

```
Out[45]: ['This',
 'is',
 'a',
 'string',
 'But',
 'it',
 'has',
 'punctuation',
 'How',
 'can',
 'we',
 'remove',
 'it']
```

```
In [46]: clean = ' '.join(re.findall('[^!.? ]+',test_phrase))
```

```
In [47]: clean
```

```
Out[47]: 'This is a string But it has punctuation How can we remove it'
```

Brackets for Grouping

As we showed above we can use brackets to group together options, for example if we wanted to find hyphenated words:

```
In [48]: text = 'Only find the hypen-words in this sentence. But you do not know how long-ish they are'
```

```
In [49]: re.findall(r'[\w]+-[\w]+',text)
```

```
Out[49]: ['hypen-words', 'long-ish']
```

Parenthesis for Multiple Options

If we have multiple options for matching, we can use parenthesis to list out these options. For Example:

```
In [50]: # Find words that start with cat and end with one of these options: 'fish','nap', or 'claw'
text = 'Hello, would you like some catfish?'
texttwo = "Hello, would you like to take a catnap?"
textthree = "Hello, have you seen this caterpillar?"
```

```
In [51]: re.search(r'cat(fish|nap|claw)',text)
```

```
Out[51]: <_sre.SRE_Match object; span=(27, 34), match='catfish'>
```

```
In [52]: re.search(r'cat(fish|nap|claw)',texttwo)
```

```
Out[52]: <_sre.SRE_Match object; span=(32, 38), match='catnap'>
```

```
In [53]: # None returned
re.search(r'cat(fish|nap|claw)',textthree)
```

Conclusion

Excellent work! For full information on all possible patterns, check out:
https://docs.python.org/3/howto/regex.html

Character	Description	Example Pattern Code	Exammple Match
+	Occurs one or more times	Version \w-\w+	Version A-b1_1
{3}	Occurs exactly 3 times	\D{3}	abc
{2,4}	Occurs 2 to 4 times	\d{2,4}	123
{3,}	Occurs 3 or more	\w{3,}	anycharacters
*	Occurs zero or more times	A*B*C*	AAACC
?	Once or none	plurals?	plural