Regular Expressions

The term "regular expression" is a formal, linguistic term that describes the ability to desc you might be interested to read about (https://en.wikipedia.org/wiki/Regular_language). For our purposes, regular expressions (AKA "regexes" or a "regex") is a way to formally describe some string that we want to find. Regexes are a DSL (domain-specific language) that we use inside Python, just like in the previous chapter we use SQL statements to communite with SQLite. We can `import re` to use the Python regular expression module and use it to search text.

In the tic-tac-toe exercise, we needed to see if the `--player` argument was exactly one character that was either an 'X' or an 'O'. Here's code that can do that:

```
>>> player = 'X'
>>> if len(player) == 1 and (player == 'X' or player == 'O'):
...   print('OK')
...
OK
>>> player = 'B'
>>> if len(player) == 1 and (player == 'X' or player == 'O'):
...   print('OK')
...
```

A shorter way to write this could be:

```
if len(player) == 1 and player in 'XO':
```

It's not too onerous, but it quickly gets worse as we get more complicated requirements. In that same exercise, we needed to check if `--state` was exactly 9 characters composed entirely of '.', 'X', 'O':

```
>>> state = 'XXX...OOO'
>>> 'OK' if len(state) == 9 and all(map(lambda x: x in 'XO.', state)) else 'No'
'OK'
>>> state = 'XXX...OOA'
>>> 'OK' if len(state) == 9 and all(map(lambda x: x in 'XO.', state)) else 'No'
'No'
```

A regular expression allows us to **describe** what we want rather than **implement** the code to find what we want. We can create a class of allowed characters with `[XO]` and additionally constraint it to be exactly one character wide with `{1}` after the class. (Note that `{}` for match length can be in the format `{exactly}`, `{min,max}`, `{min,}`, or `{,max}`.)

```
>>> import re
>>> player = 'X'
>>> re.match('[XO]{1}', player)
```

```
<_sre.SRE_Match object; span=(0, 1), match='X'>
>>> player = 'A'
>>> re.match('[XO]{1}', player)
```

We can extend this to our state problem:

```
>>> state = 'XXX...OOO'
>>> re.match('[XO.]{9}', state)
<_sre.SRE_Match object; span=(0, 9), match='XXX...OOO'>
>>> state = 'XXX...OOA'
>>> re.match('[XO.]{9}', state)
```

When we were starting out with the Unix command line, one exercise had us using `grep` to look for lines that start with vowels. One solution was:

```
$ grep -io '^[aeiou]' scarlet.txt | sort | uniq -c
  59 A
  10 E
  91 I
  20 O
   6 U
 651 a
 199 e
 356 i
 358 o
 106 u
```

We used square brackets `[]` to enumerate all the vowels `[aeiou]` and used the `-i` flag to `grep` to indicate it should match case **insensitively**. Additionally, the `^` indicated that the match should occur at the start of the string.

ENA Metadata

Let's examine the ENA metadata from the XML parsing example. We see there are many ways that latitude/longitude have been represented:

```
$ ./xml_ena.py *.xml | grep lat_lon
attr.lat_lon             : 27.83387,-65.4906
attr.lat_lon             : 29.3 N 122.08 E
attr.lat_lon             : 28.56_-88.70377
attr.lat_lon             : 39.283N 76.611 W
attr.lat_lon             : 78 N 5 E
attr.lat_lon             : missing
attr.lat_lon             : 0.00 N, 170.00 W
attr.lat_lon             : 11.46'45.7" 93.01'22.3"
```

How can we go about parsing all the various ways this data has been encoded?

Regular expressions provide us a way to describe in very specific way what we want.

Let's start just with the idea of matching a number (where "number" is a string that could be parsed into a number) like "27.83387":

```
>>> import re
>>> re.search('\d', '27.83387')
<_sre.SRE_Match object; span=(0, 1), match='2'>
```

The `\d` pattern means "any number" which is the same as `[0-9]` where the `[]` creates a class of characters and `0-9` expands to all the numbers from zero to nine. The problem is that it only matches one number, `2`. Change it to `\d+` to indicate "one or more numbers":

```
>>> re.search('\d+', '27.83387')
<_sre.SRE_Match object; span=(0, 2), match='27'>
```

Now let's capture the decimal point:

```
>>> re.search('\d+.', '27.83387')
<_sre.SRE_Match object; span=(0, 3), match='27.'>
```

You migth think that's perfect, but the `.` has a special meaning in regex. It means "one of anything", so it matches this, too:

```
>>> re.search('\d+.', '27x83387')
<_sre.SRE_Match object; span=(0, 3), match='27x'>
```

To indicate we want a literal `.` we have to make it `\.` (backslash-escape):

```
>>> re.search('\d+\.', '27.83387')
<_sre.SRE_Match object; span=(0, 3), match='27.'>
>>> re.search('\d+\.', '27x83387')
```

Notice that the second try returns nothing.

To capture the bit after the `.`, add more numbers:

```
>>> re.search('\d+\.\d+', '27.83387')
<_sre.SRE_Match object; span=(0, 8), match='27.83387'>
```

But we won't always see floats. Can we make this regex match integers, too? We can indicate that part of a pattern is optional by putting a `?` after it. Since we need more than one thing to be optional, we need to wrap it in parens:

```
>>> re.search('\d+\.\d+', '27')
>>> re.search('\d+(\.\d+)?', '27')
<_sre.SRE_Match object; span=(0, 2), match='27'>
>>> re.search('\d+(\.\d+)?', n1)
<_sre.SRE_Match object; span=(0, 8), match='27.83387'>
```

What if there is a negative symbol in front? Add `-?` (an optional dash) at the beginning:

```
>>> re.search('-?\d+(\.\d+)?', '-27.83387')
<_sre.SRE_Match object; span=(0, 9), match='-27.83387'>
>>> re.search('-?\d+(\.\d+)?', '27.83387')
<_sre.SRE_Match object; span=(0, 8), match='27.83387'>
>>> re.search('-?\d+(\.\d+)?', '-27')
<_sre.SRE_Match object; span=(0, 3), match='-27'>
>>> re.search('-?\d+(\.\d+)?', '27')
<_sre.SRE_Match object; span=(0, 2), match='27'>
```

Sometimes we actually find a `+` at the beginning, so we can make an optional character class `[+-]?`:

```
>>> re.search('[+-]?\d+(\.\d+)?', '-27.83387')
<_sre.SRE_Match object; span=(0, 9), match='-27.83387'>
>>> re.search('[+-]?\d+(\.\d+)?', '+27.83387')
<_sre.SRE_Match object; span=(0, 9), match='+27.83387'>
>>> re.search('[+-]?\d+(\.\d+)?', '27.83387')
<_sre.SRE_Match object; span=(0, 8), match='27.83387'>
```

Now we can match things that basically look like a floating point number or an integer, both positive and negative.

There are many resources you can use to thoroughly learn regular expressions, so I won't try to cover them completely here. I will mostly try to introduce the general idea and show you some useful regexes you could steal.

Here is an example of how you can embed regexes in your Python code. This version can parse all the versions of latitude/longitude shown above:

```
$ cat -n ena_re.py
     1  #!/usr/bin/env python3
     2  """
     3  Author : kyclark
     4  Date   : 2019-02-22
     5  Purpose: Rock the Casbah
     6  """
     7
     8  import os
     9  import re
    10  import sys
    11
    12
    13  # --------------------------------------------------
    14  def main():
    15      args = sys.argv[1:]
    16
    17      if len(args) != 1:
    18          print('Usage: {} FILE'.format(os.path.basename(sys.argv[0])))
    19          sys.exit(1)
```

```
20
21      file = args[0]
22
23      float_ = r'[+-]?\d+\.*\d*'
24      line_re = re.compile('^attr\.([^\s]+)\s+:\s+(.+)$')
25      ll1 = re.compile('(' + float_ + ')\s*[,_]\s*(' + float_ + ')')
26      ll2 = re.compile('(' + float_ + ')(?:\s*([NS]))?(?:\s*,)?\s+(' + float_ +
27                       ')(?:\s*([EW])?)')
28
29      loc_hms = r"""
30      \d+\.\d+'\d+\.\d+"
31      """.strip()
32      ll3 = re.compile('(' + loc_hms + ')\s+(' + loc_hms + ')')
33
34      for line in open(file):
35          match = line_re.search(line)
36          if match:
37              fld, val = match.group(1), match.group(2)
38              print('{} = {}'.format(fld, val))
39
40              if fld == 'lat_lon':
41                  ll_match1 = ll1.search(val)
42                  ll_match2 = ll2.search(val)
43                  ll_match3 = ll3.search(val)
44
45                  if ll_match1:
46                      lat, lon = ll_match1.group(1), ll_match1.group(2)
47                      lat = float(lat)
48                      lon = float(lon)
49                      print('lat = {}, lon = {}'.format(lat, lon))
50                  elif ll_match2:
51                      lat, lat_dir, lon, lon_dir = ll_match2.group(
52                          1), ll_match2.group(2), ll_match2.group(
53                              3), ll_match2.group(4)
54                      lat = float(lat)
55                      lon = float(lon)
56
57                      if lat_dir == 'S':
58                          lat *= -1
59
60                      if lon_dir == 'W':
61                          lon *= -1
62                      print('lat = {}, lon = {}'.format(lat, lon))
63                  elif ll_match3:
64                      lat, lon = ll_match3.group(1), ll_match3.group(2)
65                      print('lat = {}, lon = {}'.format(lat, lon))
```

```
    66                  else:
    67                      print('No match')
    68
    69
    70  # -----------------------------------------------
    71  main()
$ cat re.txt
attr.lat_lon             : 27.83387,-65.4906
attr.lat_lon             : 29.3 N 122.08 E
attr.lat_lon             : 28.56_-88.70377
This line will not be included
attr.lat_lon             : 39.283N 76.611 W
attr.lat_lon             : 78 N 5 E
attr.lat_lon             : missing
attr.lat_lon             : 0.00 N, 170.00 W
attr.lat_lon             : 11.46'45.7" 93.01'22.3"
$ ./ena_re.py re.txt
lat_lon = 27.83387,-65.4906
lat = 27.83387, lon = -65.4906
lat_lon = 29.3 N 122.08 E
lat = 29.3, lon = 122.08
lat_lon = 28.56_-88.70377
lat = 28.56, lon = -88.70377
lat_lon = 39.283N 76.611 W
lat = 39.283, lon = -76.611
lat_lon = 78 N 5 E
lat = 78.0, lon = 5.0
lat_lon = missing
No match
lat_lon = 0.00 N, 170.00 W
lat = 0.0, lon = -170.0
lat_lon = 11.46'45.7" 93.01'22.3"
lat = 11.46'45.7", lon = 93.01'22.3"
```

We see a similar problem with "collection_date":

```
$ ./xml_ena.py *.xml | grep collection
attr.collection_date     : March 24, 2014
attr.collection_date     : 2013-08-15/2013-08-28
attr.collection_date     : 20100910
attr.collection_date     : 02-May-2012
attr.collection_date     : Jul-2009
attr.collection_date     : missing
attr.collection_date     : 2013-12-23
attr.collection_date     : 5/04/2012
```

Imagine how you might go about parsing all these various representations of

dates. Be aware that parsing date/time formats is so problematic and ubiquitous that many people have already written modules to assist you!

To run this code, you will need to install the `dateparser` module:

```
$ python3 -m pip install dateparser
```

Et voila!

```
>>> import dateparser as p
>>> p.parse('March 24, 2014')
datetime.datetime(2014, 3, 24, 0, 0)
>>> p.parse('2013-08-15')
datetime.datetime(2013, 8, 15, 0, 0)
>>> p.parse('20100910')
datetime.datetime(2000, 2, 1, 9, 1)
>>> p.parse('02-May-2012')
datetime.datetime(2012, 5, 2, 0, 0)
>>> p.parse('Jul-2009')
datetime.datetime(2009, 7, 23, 0, 0)
>>> p.parse('5/04/2012')
datetime.datetime(2012, 5, 4, 0, 0)
```

You can see it's not perfect, e.g., "Jul-2009" should not resolve to the 23rd of July, but, honestly, what should it be? (Is the 1st any better?!) Still, this saves you writing a lot of code. And, trust me, **THIS IS REAL DATA**! While trying to parse latitude, longitude, collection date, and depth for 35K marine metagenomes from the ENA, I wrote a hundreds of lines of code and dozens of regular expressions!