Introduction to Regular Expressions in Python

The term "regular expression" is a formal, linguistic term 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 of characters that we want to find. Regexes are an entirely separate DSL (domain-specific language) that we use inside Python, just like in the previous chapter we use SQL statements to communite with SQLite. While it's a bit of a drag to have to learn yet another language, the bonus is that you can use regular expressions in many places besides Python including with command line tools like `grep` and `awk` as well as within other languages like Perl and Rust.

We can `import re` to use the Python regular expression module and use it to search text. For instance, 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:

```
for player in ['X', 'A', 'O', '5']:
    if len(player) == 1 and (player == 'X' or player == 'O'):
        print('{} OK'.format(player))
    else:
        print('{} bad'.format(player))
```

```
X OK
A bad
O OK
5 bad
```

A shorter way to write this could be:

```
for player in ['X', 'A', 'B', '5']:
    if len(player) == 1 and player in 'XO':
        print('{} OK'.format(player))
    else:
        print('{} bad'.format(player))
```

```
X OK
A bad
B bad
5 bad
```

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":

```
for state in ['XXX...OOO', 'XXX...OOA']:
    #print([(x, x in 'XO.') for x in state])
```

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

```
XXX...OOO OK
XXX...OOA No
```

Can we make this simpler? Well, 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. Those were regular expressions we were using.

The regex 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}`.)

To use regular expressions:

```
import re
```

Now let's describe our pattern using a character class `[XO]` and the length `{1}`:

```
for player in ['X', 'O', 'A']:
    print(player, re.match('[XO]{1}', player))
```

```
X <_sre.SRE_Match object; span=(0, 1), match='X'>
O <_sre.SRE_Match object; span=(0, 1), match='O'>
A None
```

We can extend this to our state problem:

```
state = 'XXX...OOO'
print(state, re.match('[XO.]{9}', state))
```

```
XXX...OOO <_sre.SRE_Match object; span=(0, 9), match='XXX...OOO'>
```

```
state = 'XXX...OOA'
print(state, re.match('[XO.]{9}', state))
```

```
XXX...OOA None
```

Building regular expressions

How do we match a number?

```
print(re.match('1', '1'))
```

```
<_sre.SRE_Match object; span=(0, 1), match='1'>
```

But that only works for just "1"

```
print(re.match('2', '1'))
```

```
None
```

How do we match all the numbers from 0 to 9? We can create a character class that contains that range:

```
print(re.match('[0-9]', '1'))
```

```
<_sre.SRE_Match object; span=(0, 1), match='1'>
```

There is a short-hand for the character class `[0-9]` that is `\d` (digit)

```
re.match('\d', '1')
```

```
<_sre.SRE_Match object; span=(0, 1), match='1'>
```

But this only matches the first number we see:

```
re.match('\d', '123')
```

```
<_sre.SRE_Match object; span=(0, 1), match='1'>
```

We can use `{}` to indicate `{min,max}`, `{min,}`, `{,max}`, or `{exactly}`:

```
print(re.match('\d{1,4}', '8005551212'))
```

```
<_sre.SRE_Match object; span=(0, 4), match='8005'>
```

```
print(re.match('\d{1,}', '8005551212'))
```

```
<_sre.SRE_Match object; span=(0, 10), match='8005551212'>
```

```
print(re.match('\d{,5}', '8005551212'))
```

```
<_sre.SRE_Match object; span=(0, 5), match='80055'>
```

```
print(re.match('\d{8}', '8005551212'))
```

```
<_sre.SRE_Match object; span=(0, 8), match='80055512'>
```

match vs search

Note that we are using `re.match` which requires the regex to match **at the beginning of the string**:

```
print(re.match('\d{10}', 'That number to call is 8005551212!'))
```

```
None
```

If you want to match anywhere in the string, use `re.search`:

```
for s in ['123', 'abc456', '789def']:
    print(s, re.search('\d{3}', s))
```

```
123 <_sre.SRE_Match object; span=(0, 3), match='123'>
abc456 <_sre.SRE_Match object; span=(3, 6), match='456'>
789def <_sre.SRE_Match object; span=(0, 3), match='789'>
```

To anchor your match to the beginning of the string, use the `^`:

```
for s in ['123', 'abc456', '789def']:
    print(s, re.search('^\d{3}', s))
```

```
123 <_sre.SRE_Match object; span=(0, 3), match='123'>
abc456 None
789def <_sre.SRE_Match object; span=(0, 3), match='789'>
```

Use `$` for the end of the string:

```
for s in ['123', 'abc456', '789def']:
    print(s, re.search('\d{3}$', s))
```

```
123 <_sre.SRE_Match object; span=(0, 3), match='123'>
abc456 <_sre.SRE_Match object; span=(3, 6), match='456'>
789def None
```

And use both to say that the entire string from beginning to end must match:

```
for s in ['123', 'abc456', '789def']:
    print(s, re.search('^\d{3}$', s))
```

```
123 <_sre.SRE_Match object; span=(0, 3), match='123'>
abc123 None
123def None
```

Returning to our previous problem of trying to see if we got *exactly* one "X" or "O" for our tic-tac-toe player:

```
for player in ['X', 'O', 'XX', 'OO']:
    print(player, re.match('[XO]{1}', player))
```

```
X <_sre.SRE_Match object; span=(0, 1), match='X'>
O <_sre.SRE_Match object; span=(0, 1), match='O'>
XX <_sre.SRE_Match object; span=(0, 1), match='X'>
```

```
OO <_sre.SRE_Match object; span=(0, 1), match='O'>
```

The problem is that there is a match of `[XO]{1}` in the strings "XX" and "OO" – there *is* exactly one X or O at the beginning of those strings. Since `re.match` already anchors the match to the beginning of the string, we could just add `$` to the end of our pattern:

```
for player in ['X', 'O', 'XX', 'OO']:
    print(player, re.match('[XO]{1}$', player))
```

```
X <_sre.SRE_Match object; span=(0, 1), match='X'>
O <_sre.SRE_Match object; span=(0, 1), match='O'>
XX None
OO None
```

Or use `re.search` with `^$` to indicate a match over the entire string:

```
for player in ['X', 'O', 'XX', 'OO']:
    print(player, re.search('^[XO]{1}$', player))
```

```
X <_sre.SRE_Match object; span=(0, 1), match='X'>
O <_sre.SRE_Match object; span=(0, 1), match='O'>
XX None
OO None
```

Matching SSNs and Dates

What if we wanted to recognize a US SSN (social security number)? We will use `re.compile` to create the regex and use it in a `for` loop:

```
ssn_re = re.compile('\d{3}-\d{2}-\d{4}')
for s in ['123456789', '123-456-789', '123-45-6789']:
    print('{}: {}'.format(s, ssn_re.match(s)))
```

```
123456789: None
123-456-789: None
123-45-6789: <_sre.SRE_Match object; span=(0, 11), match='123-45-6789'>
```

SSNs always use a dash (`-`) as a number separator, but dates do not.

```
date_re = re.compile('\d{4}-\d{2}-\d{2}')
dates = ['1999-01-01', '1999/01/01']
for d in dates:
    print('{}: {}'.format(d, date_re.match(d)))
```

```
1999-01-01: <_sre.SRE_Match object; span=(0, 10), match='1999-01-01'>
1999/01/01: None
```

Just as we created a character class with `[0-9]` to represent all the numbers from 0 to 9, we can create a class to represent the separators "/" and "-" with `[/-]`. As regular expressions get longer, it makes sense to break each unit onto

a different line and use Python's literal string expression to join them into a single string. As a bonus, we can comment on each unit of the regex.

```
date_re = re.compile('\d{4}'  # year
                     '[/-]'   # separator
                     '\d{2}'  # month
                     '[/-]'   # separator
                     '\d{2}') # day

sep = '[/-]'
date_re = re.compile('\d{4}' + # year
                     sep     + # separator
                     '\d{1,2}' + # month
                     sep     + # separator
                     '\d{1,2}')  # day

dates = ['1999-01-01', '1999/01/01', '1999/1/1']
for d in dates:
    print('{}: {}'.format(d, date_re.match(d)))
```

```
1999-01-01: <_sre.SRE_Match object; span=(0, 10), match='1999-01-01'>
1999/01/01: <_sre.SRE_Match object; span=(0, 10), match='1999/01/01'>
1999/1/1: <_sre.SRE_Match object; span=(0, 8), match='1999/1/1'>
```

If we wanted to extract each part of the date (year, month, day), we can use parentheses `()` around the parts we want to capture into `groups`. The group "0" is the whole string that was match, and they are numbered sequentially after that for each group.

Can you change the regex to match all three strings?

```
date_re = re.compile('(\d{4})'  # capture year (group 1)
                     '[/-]'     # separator
                     '(\d{2})'  # capture month (group 2)
                     '[/-]'     # separator
                     '(\d{2})') # capture day (group 3)

dates = ['1999-01-01', '1999/01/01', '1999.01.01']
for d in dates:
    match = date_re.match(d)
    print('{}: {}'.format(d, 'match' if match else 'miss'))
    if match:
        print(match.groups())
        print('year:', match.group(1))
    print()
```

```
1999-01-01: match
('1999', '01', '01')
year: 1999
```

```
1999/01/01: match
('1999', '01', '01')
year: 1999

1999.01.01: miss
```

As we add more groups, it can be confusing to remember them by their positions, so we can name them with `?P<name>` just inside the opening paren.

```
date_re = re.compile('(?P<year>\d{4})'
                     '[/-]'
                     '(?P<month>\d{2})'
                     '[/-]'
                     '(?P<day>\d{2})')

dates = ['1999-01-01', '1999/01/01', '1999.01.01']

for d in dates:
    match = date_re.match(d)
    print('{}: {}'.format(d, 'match' if match else 'miss'))
    if match:
        print('{} = year "{}" month "{}" day "{}"'.format(d,
                                                         match.group('year'),
                                                         match.group('month'),
                                                         match.group('day')))
    print()
```

```
1999-01-01: match
1999-01-01 = year "1999" month "01" day "01"

1999/01/01: match
1999/01/01 = year "1999" month "01" day "01"

1999.01.01: miss
```

Matching US Phone Numbers

What if we wanted to match a US phone number?

```
phone_re = re.compile('(\d{3})'  # area code
                      ' '        # a space
                      '\d{3}'    # prefix
                      '-'        # dash
                      '\d{4}')   # line number
print(phone_re.match('(800) 555-1212'))
```

```
None
```

Why didn't that work?

What do those parentheses do again? They group!

So we need to indicate that the parens are literal things to match by using backslashes `\` to escape them.

```
phone_re = re.compile('\('     # left paren
                      '\d{3}'  # area code
                      '\)'     # right paren
                      ' '      # space
                      '\d{3}'  # prefix
                      '-'      # dash
                      '\d{4}') # line number
print(phone_re.match('(800) 555-1212'))
```

```
<_sre.SRE_Match object; span=(0, 14), match='(800) 555-1212'>
```

We could also use character classes to make this more readable:

```
phone_re = re.compile('[(]'    # left paren
                      '\d{3}'  # area code
                      '[)]'    # right paren
                      ' '      # space
                      '\d{3}'  # prefix
                      '-'      # dash
                      '\d{4}') # line number


print(phone_re.match('(800) 555-1212'))
```

```
<_sre.SRE_Match object; span=(0, 14), match='(800) 555-1212'>
```

There is not always a space after the area code, and it may sometimes it may be more than one space (or a tab?). We can use the `\s` to indicate any type of whitespace and `*` to indicate zero or more:

```
phone_re = re.compile('[(]'    # left paren
                      '\d{3}'  # area code
                      '[)]'    # right paren
                      '\s*'    # zero or more spaces
                      '\d{3}'  # prefix
                      '-'      # dash
                      '\d{4}') # line number
phones = ['(800)555-1212', '(800) 555-1212', '(800)  555-1212']
for phone in phones:
    print('{}\t{}'.format(phone, phone_re.match(phone)))
```

```
(800)555-1212   <_sre.SRE_Match object; span=(0, 13), match='(800)555-1212'>
(800) 555-1212  <_sre.SRE_Match object; span=(0, 14), match='(800) 555-1212'>
```

```
(800)  555-1212 <_sre.SRE_Match object; span=(0, 15), match='(800)  555-1212'>
```

When the parens around the area code are optional, usually there is a dash to separate the area code:

```
phone_re = re.compile('[(]?'   # optional left paren
                      '\d{3}'  # area code
                      '[)]?'   # optional right paren
                      '[-]?'   # optional dash
                      '\s*'    # zero or more whitespace
                      '\d{3}'  # prefix
                      '-'      # dash
                      '\d{4}') # line number

phones = ['(800)555-1212', '(800) 555-1212', '800-555-1212']
for phone in phones:
    print('{}\t{}'.format(phone, phone_re.match(phone)))
```

```
(800)555-1212   <_sre.SRE_Match object; span=(0, 13), match='(800)555-1212'>
(800) 555-1212  <_sre.SRE_Match object; span=(0, 14), match='(800) 555-1212'>
800-555-1212    <_sre.SRE_Match object; span=(0, 12), match='800-555-1212'>
```

This has the affect of matching a dash after parens which is generally not a valid format:

```
phone_re = re.compile('[(]?'
                      '\d{3}'
                      '[)]?'
                      '[-]?'
                      '\s*'
                      '\d{3}'
                      '-'
                      '\d{4}')

phone_re.match('(800)-555-1212')
```

```
<_sre.SRE_Match object; span=(0, 14), match='(800)-555-1212'>
```

We really have to create two regexes to handle these cases:

```
phone_re1 = re.compile('[(]'
                       '\d{3}'
                       '[)]'
                       '\s*'
                       '\d{3}'
                       '-'
                       '\d{4}')

phone_re2 = re.compile('\d{3}'
                       '-'
```

```
                        '\d{3}'
                        '-'
                        '\d{4}')

phones = ['(800)555-1212', '(800) 555-1212', '800-555-1212', '(800)-555-1212']
for phone in phones:
    match1 = phone_re1.match(phone)
    match2 = phone_re2.match(phone)
    print('{}\t{}'.format(phone, 'match' if match1 or match2 else 'miss'))
```

```
(800)555-1212   match
(800) 555-1212  match
800-555-1212    match
(800)-555-1212  miss
```

I worked with a graphic artist who always insisted on using dots as the number separator, and sometimes there are no separators at all. The combination of these two regexes find the valid formats and skip the invalid one.

```
phone_re1 = re.compile('[(]'
                        '\d{3}'
                        '[)]'
                        '\s*'
                        '\d{3}'
                        '[.-]'
                        '\d{4}')

phone_re2 = re.compile('\d{3}'
                        '[.-]?'
                        '\d{3}'
                        '[.-]?'
                        '\d{4}')

phones = ['8005551212', '(800)555-1212', '(800) 555-1212',
          '800-555-1212', '(800)-555-1212', '800.555.1212']

for phone in phones:
    match = phone_re1.match(phone) or phone_re2.match(phone)
    print('{}\t{}'.format(phone, 'match' if match else 'miss'))
```

```
8005551212  match
(800)555-1212   match
(800) 555-1212  match
800-555-1212    match
(800)-555-1212  miss
800.555.1212    match
```

OK, now let's normalize the numbers by using parens to capture the area code,

prefix, and line number and then create a standard representation.

```
phone_re1 = re.compile('[(]'
                       '(\d{3})'  # group 1
                       '[)]'
                       '\s*'
                       '(\d{3})'  # group 2
                       '[.-]'
                       '(\d{4})') # group 3

phone_re2 = re.compile('(\d{3})'  # group 1
                       '[.-]?'
                       '(\d{3})'  # group 2
                       '[.-]?'
                       '(\d{4})') # group 3

phones = ['8005551212', '(800)555-1212', '(800) 555-1212',
          '800-555-1212', '(800)-555-1212', '800.555.1212']

for phone in phones:
    match = phone_re1.match(phone) or phone_re2.match(phone)
    standard = '{}-{}-{}'.format(match.group(1),
                                 match.group(2),
                                 match.group(3)) if match else 'miss'
    print('{}\t{}'.format(phone, standard))
```

```
8005551212  800-555-1212
(800)555-1212   800-555-1212
(800) 555-1212  800-555-1212
800-555-1212    800-555-1212
(800)-555-1212  miss
800.555.1212    800-555-1212
```

And if we add named capture groups...

```
phone_re1 = re.compile('[(]'
                       '(?P<area_code>\d{3})'
                       '[)]'
                       '\s*'
                       '(?P<prefix>\d{3})'
                       '[.-]'
                       '(?P<line_num>\d{4})')

phone_re2 = re.compile('(?P<area_code>\d{3})'
                       '[.-]?'
                       '(?P<prefix>\d{3})'
                       '[.-]?'
                       '(?P<line_num>\d{4})')
```

```
phones = ['8005551212', '(800)555-1212', '(800) 555-1212',
          '800-555-1212', '(800)-555-1212', '800.555.1212']

for phone in phones:
    match = phone_re1.match(phone) or phone_re2.match(phone)
    standard = '{}-{}-{}'.format(match.group('area_code'),
                                 match.group('prefix'),
                                 match.group('line_num')) if match else 'miss'
    print('{}\t{}'.format(phone, standard))
```

```
8005551212  800-555-1212
(800)555-1212   800-555-1212
(800) 555-1212  800-555-1212
800-555-1212    800-555-1212
(800)-555-1212  miss
800.555.1212    800-555-1212
```

And if we add named capture groups and named groups in `format`:

```
phone_re1 = re.compile('[(]'
                       '(?P<area_code>\d{3})'
                       '[)]'
                       '\s*(?P<prefix>\d{3})'
                       '[.-]'
                       '(?P<line_num>\d{4})')
phone_re2 = re.compile('(?P<area_code>\d{3})'
                       '[.-]?'
                       '(?P<prefix>\d{3})'
                       '[.-]?'
                       '(?P<line_num>\d{4})')
phones = ['8005551212', '(800)555-1212', '(800) 555-1212',
          '800-555-1212', '(800)-555-1212', '800.555.1212']
for phone in phones:
    match = phone_re1.match(phone) or phone_re2.match(phone)
    tmpl = '{area_code}-{prefix}-{line_num}'
    standard = tmpl.format(prefix=match.group('prefix'),
                           area_code=match.group('area_code'),
                           line_num=match.group('line_num')) if match else 'miss'
    print('{}\t{}'.format(phone, standard))
```

```
8005551212  800-555-1212
(800)555-1212   800-555-1212
(800) 555-1212  800-555-1212
800-555-1212    800-555-1212
(800)-555-1212  miss
800.555.1212    800-555-1212
```

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":

```
print(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 might 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):

```
print(re.search('\d+\.', '27.83387'))
print(re.search('\d+\.', '27x83387'))
```

```
<_sre.SRE_Match object; span=(0, 3), match='27.'>
None
```

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:

```
print(re.search('\d+\.\d+', '27'))
print(re.search('\d+(\.\d+)?', '27'))
print(re.search('\d+(\.\d+)?', '27.83387'))
```

```
None
<_sre.SRE_Match object; span=(0, 2), match='27'>
<_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:

```
print(re.search('-?\d+(\.\d+)?', '-27.83387'))
print(re.search('-?\d+(\.\d+)?', '27.83387'))
print(re.search('-?\d+(\.\d+)?', '-27'))
print(re.search('-?\d+(\.\d+)?', '27'))
```

```
<_sre.SRE_Match object; span=(0, 9), match='-27.83387'>
<_sre.SRE_Match object; span=(0, 8), match='27.83387'>
<_sre.SRE_Match object; span=(0, 3), match='-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 `[+-]?`:

```
print(re.search('[+-]?\d+(\.\d+)?', '-27.83387'))
print(re.search('[+-]?\d+(\.\d+)?', '+27.83387'))
print(re.search('[+-]?\d+(\.\d+)?', '27.83387'))
```

```
<_sre.SRE_Match object; span=(0, 9), match='-27.83387'>
<_sre.SRE_Match object; span=(0, 9), match='+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.

Usually the data we want to find it part of a larger string, however, and the above fails to capture more than one thing, e.g.:

```
print(re.search('[+-]?\d+(\.\d+)?', 'Lat is "-27.83387" and lon is "+132.43."'))
```

```
<_sre.SRE_Match object; span=(8, 17), match='-27.83387'>
```

We really need to match more than once using our pattern matching to extract data. We saw earlier that we can use parens to group optional patterns, but the parens also end up creating a **capture group** that we can refer to by position:

```
re.findall('([+-]?\d+(\.\d+)?)','Lat is "-27.83387" and lon is "+132.43."')
```

```
[('-27.83387', '.83387'), ('+132.43', '.43')]
```

OK, it was a bit unexpected that we have matches for both the whole float and the decimal part. This is because of the dual nature of the parens, and in the case of using them to group the optional part we are also creating another capture. If we change `()` to `(?:)`, we make this a non-capturing group:

```
re.findall('([+-]?\d+(?:\.\d+)?)', 'Lat is "-27.83387" and lon is "+132.43."')
```

```
['-27.83387', '+132.43']
```

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. This code uses parens to create capture groups which it then uses `match.group(n)` to extract:

```
$ cat -n parse_lat_lon.py
     1  #!/usr/bin/env python3
     2
     3  import os
     4  import re
     5  import sys
     6
     7  args = sys.argv[1:]
     8
     9  if len(args) != 1:
    10      print('Usage: {} FILE'.format(os.path.basename(sys.argv[0])))
    11      sys.exit(1)
    12
    13  file = args[0]
    14
    15  float_ = r'[+-]?\d+\.*\d*'
    16  ll1 = re.compile('(' + float_ + ')\s*[,_]\s*(' + float_ + ')')
    17  ll2 = re.compile('(' + float_ + ')(?:\s*([NS]))?(?:\s*,)?\s+(' + float_ +
    18                   ')(?:\s*([EW]))?)')
    19  loc_hms = r"""
    20  \d+\.\d+'\d+\.\d+"
    21  """.strip()
```

```
    ll3 = re.compile('(' + loc_hms + ')\s+(' + loc_hms + ')')

    for line in open(file):
        line = line.rstrip()
        ll_match1 = ll1.search(line)
        ll_match2 = ll2.search(line)
        ll_match3 = ll3.search(line)

        if ll_match1:
            lat, lon = ll_match1.group(1), ll_match1.group(2)
            lat = float(lat)
            lon = float(lon)
            print('lat = {}, lon = {}'.format(lat, lon))
        elif ll_match2:
            lat, lat_dir, lon, lon_dir = ll_match2.group(
                1), ll_match2.group(2), ll_match2.group(
                    3), ll_match2.group(4)
            lat = float(lat)
            lon = float(lon)

            if lat_dir == 'S':
                lat *= -1

            if lon_dir == 'W':
                lon *= -1
            print('lat = {}, lon = {}'.format(lat, lon))
        elif ll_match3:
            lat, lon = ll_match3.group(1), ll_match3.group(2)
            print('lat = {}, lon = {}'.format(lat, lon))
        else:
            print('No match: "{}"'.format(line))
$ cat lat_lon.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"
$ ./parse_lat_lon.py lat_lon.txt
lat = 27.83387, lon = -65.4906
lat = 29.3, lon = 122.08
lat = 28.56, lon = -88.70377
No match: "This line will not be included"
```

```
lat = 39.283, lon = -76.611
lat = 78.0, lon = 5.0
No match: "attr.lat_lon             : missing"
lat = 0.0, lon = -170.0
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 the code below, you will need to install the `dateparser` module:

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

```
import dateparser
for date in ['March 24, 2014',
             '2013-08-15',
             '20100910',
             '02-May-2012',
             'Jul-2009',
             '5/04/2012']:

    print('{:15}\t{}'.format(date, dateparser.parse(date)))
```

```
March 24, 2014  2014-03-24 00:00:00
2013-08-15      2013-08-15 00:00:00
20100910        2000-02-01 09:01:00
02-May-2012     2012-05-02 00:00:00
Jul-2009        2009-07-26 00:00:00
5/04/2012       2012-05-04 00:00:00
```

You can see it's not perfect, e.g., "20100910" should be "2010-09-10" and "Jul-2009" should not resolve to the 26th 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!

Exercises

Write the regular expressions to parse the year, month, and day from the following date formats found in SRA metadata. When no day is present, e.g., "2/14," use "01" for the day.

```
d1 = "2012-03-09T08:59"
print(d1, re.match('', d1))
```

```
2012-03-09T08:59 <_sre.SRE_Match object; span=(0, 0), match=''>
```

```
d2 = "2012-03-09T08:59:03"
```

```
d3 = "2017-06-16Z"
```

```
d4 = "2015-01"
```

```
d5 = "2015-01/2015-02"
```

```
d6 = "2015-01-03/2015-02-14"
```

```
d7 = "20100910"
```

```
d8 = "12/06"
```

```
d9 = "2/14"
```

```
d10 = "2/14-12/15"
```

```
d11 = "2017-06-16Z"
```

```
# "Excel" format! What is that?! Look it up.
d12 = "34210"
```

```
d13 = "Dec-2015"
```

```
d14 = "March-2017"
```

```
d15 = "May, 2017"
```

```
d16 = "March-April 2017"
```

```
d17 = "July of 2011"
```

```
d18 = "2008 August"
```

Now combine all your code from the previous cell to normalize all the dates into the same format.

```
dates = ["2012-03-09T08:59", "2012-03-09T08:59:03", "2017-06-16Z",
         "2015-01", "2015-01/2015-02", "2015-01-03/2015-02-14",
         "20100910", "12/06", "2/14", "2/14-12/15", "2017-06-16Z",
         "34210", "Dec-2015", "March-2017", "May, 2017",
         "March-April 2017", "July of 2011", "2008 August"]

for date in dates:
```

```
    year = '1999'
    month = '01'
    day = '01'
    print('{}-{}-{}\t{}'.format(year, month, day, date))
```

```
1999-01-01  2012-03-09T08:59
1999-01-01  2012-03-09T08:59:03
1999-01-01  2017-06-16Z
1999-01-01  2015-01
1999-01-01  2015-01/2015-02
1999-01-01  2015-01-03/2015-02-14
1999-01-01  20100910
1999-01-01  12/06
1999-01-01  2/14
1999-01-01  2/14-12/15
1999-01-01  2017-06-16Z
1999-01-01  34210
1999-01-01  Dec-2015
1999-01-01  March-2017
1999-01-01  May, 2017
1999-01-01  March-April 2017
1999-01-01  July of 2011
1999-01-01  2008 August
```