Regular Expressions (regex)

Regular expressions are character combinations that represent a particular pattern, for example, as we've seen before `\n` represents a new line. For more detials, see Wiki (https://en.wikipedia.org/wiki/Regular_expression), Python.org (https://docs.python.org/3/library/re.html)

Common regular expressions and character classes

Regular Expressions

^	Matches beginning of line
$	Matches end of line
.	Matches any single character except newline
a|b	Matches either a or b
(re)	Groups regular expressions and remembers matched text
\w	Matches word characters
\W	Matches nonword characters
\s	Matches whitespace. Equivalent to [\t\n\r\f]
\S	Matches nonwhitespace
\d	Matches digits
\D	Matches nondigits
\A	Matches beginning of string
\Z	Matches end of string
\b	Matches word boundaries when outside brackets (see note below)
\B	Matches nonword boundaries
\n	Matches newline
\t	Matches tab
\\t	Negates tab

NOTE: when using \b, you may need to treat string as raw (r"string").
See example below.

Repitition

+	1 or more occurrences of the pattern to its left, e.g. 'i+' = one or more i's
*	0 or more occurrences of the pattern to its left
?	0 or 1 occurrences of the pattern to its left

Character Classes

[Aa]t	Matches "At" or "at"
A[Tt]	Matches "AT" or "At"
[TU]	Matches "T" or "U"
[0-9]	Matches any digit
[a-z]	Matches any lowercase ASCII letter
[A-Z]	Matches any uppercase ASCII letter
[^UT]	Matches anything other than "T" or "U"
[^0-9]	Match anything other than a digit

See [Tutorials Point (https://www.tutorialspoint.com/python/python_reg_expressions.htm)](https://www.tutorialspoint.com/python/python_reg_expressions.htm) for additional expressions.

The `re` module

Before we can use many regular expressions in python we have to import the **`re`** module:

In [1]:
```
import re
```

The `re.search()` method

The **`re.search(pattern, subject)`** method is used to search for patterns. Its similar to the unix command `grep` :

In [22]:
```
seq = 'TGATGATTATA'
re.search('ATG.*', seq)
```

Out[22]:
```
<_sre.SRE_Match object; span=(2, 11), match='ATGATTATA'>
```

`.` is a wild card matching any single character except newline. **`*`** means to match the previous pattern 0 or more times so **`.*`** will match essentially any pattern that doesn't contain a new line.

In [7]:
```
seq = 'TGATGATTATA'
match = re.search('ATG.*', seq).group()
match
```

Out[7]:
```
'ATGATTATA'
```

To get the matching pattern, we need to append **`.group()`** to `re.search()` :

In [ ]:
```
```

To get the index at which the position starts, we need to append **`.start()`** :

In [8]:
```
seq = 'TGATGATTATA'
re.search('ATG.*', seq).start()
```

Out[8]:
```
2
```

To get the index for the end of the match, use **`.end()`** :

```
In [9]: seq = 'TGATGATTATA'
        re.search('ATG.*', seq).end()
```

```
Out[9]: 11
```

Note that the end index is actually the index just after the match.

Let's look at another approach:

```
In [10]: seq = 'ATAGCTGATGATTCTATCA'
         match = re.search('ATG.*', seq)
         sequence = match.group()
         start = match.start()
         end = match.end()
         print(sequence + ': ' + str(start) + '-' + str(end))
```

```
ATGATTCTATCA: 7-19
```

If a pattern is found, `re.search()` equates to **`True`** , otherwise it equates to **`False`** :

```
In [13]: seq = 'ATGATCGATCA'
         if re.search('ATG.*', seq):
             print('The sequence contains a start codon.')
```

```
The sequence contains a start codon.
```

Let's create a file that contains several miRNAs, some of which belong to the same family, let-7, based on positions 2-8 being identicial (miRNA1, miRNA2, and miRNA5).

```
In [14]: %%bash
         printf '>miRNA1\nTGAGGTAGTAGGTTGTATAGTT\n' >miRNAs.fa
         printf '>miRNA2\nAGAGGTAGTACCATGTATCGTT\n' >>miRNAs.fa
         printf '>miRNA3\nTCCCTGAGACCTCAAGTGTGA\n' >>miRNAs.fa
         printf '>miRNA4\nTACAAAAGAGTCGCTCTCTTCA\n' >>miRNAs.fa
         printf '>miRNA5\nTGAGGTAGTAGGATGTACAGTA\n' >>miRNAs.fa
         cat miRNAs.fa
```

```
>miRNA1
TGAGGTAGTAGGTTGTATAGTT
>miRNA2
AGAGGTAGTACCATGTATCGTT
>miRNA3
TCCCTGAGACCTCAAGTGTGA
>miRNA4
TACAAAAGAGTCGCTCTCTTCA
>miRNA5
TGAGGTAGTAGGATGTACAGTA
```

Next, let's use `re.search` to which miRNAs are in the let-7 family:

```
In [20]: try:
             file_handle = open('miRNAs.fa')
         except:
             print('error in open')
         with file_handle:
             for line in file_handle:
                 line = line.rstrip()
                 if re.search(r'\b.GAGGTAG', line):
                     print(line)
```

```
TGAGGTAGTAGGTTGTATAGTT
AGAGGTAGTACCATGTATCGTT
TGAGGTAGTAGGATGTACAGTA
```

`\b` marks word boundaries but `r` is needed out front of the pattern otherwise ptyhon interpret the backslash as something that needs to be negated.

Of course we could have done the same thing using slicing.

To identify if one of several possible patterns is matched, use `pattern1|pattern2` :

```
In [23]: seq = 'AATGACACACA'
         if re.search('ATG|AUG', seq):
             print('Sequence contains a start codon')
```

```
Sequence contains a start codon
```

Sometimes we need to group patterns. Consider the following example:

```
In [ ]: seq = 'TATGCATTGA'
        if re.search('ATG|AUG.*UGA|TGA', seq):
            print('Sequence contains start and stop codons')
```

```
In [24]: seq = 'TATGCATTGA'
         if re.search('(ATG)|(AUG).*(UGA)|(TGA)', seq):
             print('Sequence contains start and stop codons')
```

```
Sequence contains start and stop codons
```

Patterns can be delimited with `()` . Unless escaped, `()` are ignored in pattern matching.

Let's write a function that identifies the longest open reading frame in the sequence below:

```
In [27]: seq = 'TTGCCCTGAAGTAATCATGCCCTGAGCTTACACTATCACTACACTATGATCCCC'
         def longest_orf(sequence):
             return re.search('ATG([ACTG][ATCG][ATCG])*(TAA|TAG|TGA)', sequence).g
         longest_orf(seq)
```

```
Out[27]: 'ATGCCCTGAGCTTACACTATCACTACACTATGA'
```

By default, patern matching greedy, meaning it will match as much of the subject as possible. If we want to make a pattern match as conservative as possible, such that the shortes match is found, we can append `?` to `*` .

Let's find the shortest open reading frame in the sequence:

```
In [28]: seq = 'TTGCCCTGAAGTAATCATGCCCTGAGCTTACACTATCACTACACTATGATCCCC'
         def longest_orf(sequence):
             return re.search('ATG([ACTG][ATCG][ATCG])*?(TAA|TAG|TGA)', sequence).
         longest_orf(seq)
```

```
Out[28]: 'ATGCCCTGA'
```

Exercise 11a

Write a function, `intron_finder(sequence)` , that identifies the longest possible intron in a sequence, such as the one below. Assume any sequence between 'GT' and 'AG' is an intron.

```
In [34]: seq = 'GCTTCCATTCGAAGCCGTTCAAGCTATGGATGTCATTCTTCGTCATCTTCCAAGCTTGAAATACA'
         def intron_finder(sequence):
             return re.search('GT.*AG', sequence).group()

         intron_finder(seq)
```

```
Out[34]: 'GTTCAAGCTATGGATGTCATTCTTCGTCATCTTCCAAG'
```

The `re.findall()` method

The **`re.findall(pattern, subject)`** method finds all non-overlapping pattern matches and returns them in list form:

In [35]:
```
try:
    file_handle = open('miRNAs.fa')
except:
    print('error opening file')
with file_handle:
    file = file_handle.read()
    let7 = re.findall(r'\b.GAGGTAG.*', file)
    print(let7)
```

```
['TGAGGTAGTAGGTTGTATAGTT', 'AGAGGTAGTACCATGTATCGTT', 'TGAGGTAGTAGGATGT
ACAGTA']
```

If `()` are included in the pattern, only the sub pattern of the pattern enclosed in `()` will be returned (and sometimes thats all you want):

In [36]:
```
try:
    file_handle = open('miRNAs.fa')
except:
    print('error opening file')
with file_handle:
    file = file_handle.read()
    let7 = re.findall(r'\b.GAGGTAG(.*)', file)
    print(let7)
```

```
['TAGGTTGTATAGTT', 'TACCATGTATCGTT', 'TAGGATGTACAGTA']
```

Exercise 11b

Write a function, `blpI_finder(sequence)`, that finds all the posible BlpI restriction sites (GCTNAGC) in a sequences, such as the one below.

In [40]:
```
eq = 'ATGTCCATGCTGAGCCGTTCAAGCTGCTCAGCTGTCATTCTTCGTCATCGCTAAGCGCTTGAAATAC
ef blpI_finder(sequence):
   return re.findall(r'GCT.AGC', sequence)

en(blpI_finder(seq))
```

Out[40]:
```
3
```

The `re.sub()` method

For doing substiutions using regular expressions, there's the `re.sub(regex, replacement, subject)` method.

Let's convert a fasta file, such as `miRNAs.fa` to tab separated format:

In [41]:
```
%%bash
cat miRNAs.fa
```

```
>miRNA1
TGAGGTAGTAGGTTGTATAGTT
>miRNA2
AGAGGTAGTACCATGTATCGTT
>miRNA3
TCCCTGAGACCTCAAGTGTGA
>miRNA4
TACAAAAGAGTCGCTCTCTTCA
>miRNA5
TGAGGTAGTAGGATGTACAGTA
```

What pattern do we need to search for?

In [42]:
```
try:
    file_handle = open('miRNAs.fa')
except:
    print('errer opening file')
with file_handle:
    file = file_handle.read()
    tab_separated = re.sub('[0-9]\n', '\t', file)
    print(tab_separated)
```

```
>miRNA	TGAGGTAGTAGGTTGTATAGTT
>miRNA	AGAGGTAGTACCATGTATCGTT
>miRNA	TCCCTGAGACCTCAAGTGTGA
>miRNA	TACAAAAGAGTCGCTCTCTTCA
>miRNA	TGAGGTAGTAGGATGTACAGTA
```

We lost some important information in that substitution. It's common to need to keep track of part of the pattern that was matched and include it in the subsitution as part of the new pattern. To do this, we can store part or all of a pattern match as follows: `(sub_pattern)` . You can have multiple sub_patterns enclosed within `()` and each will get a separate reference - `\1`, `\2`, ... The part fo the pattern, sub_pattern enclosed within `()` can be referenced with `\1` . Let's fix our code from above:

In [43]:
```
try:
    file_handle = open('miRNAs.fa')
except:
    print('errer opening file')
with file_handle:
    file = file_handle.read()
    tab_separated = re.sub('>(.*)\n', r'\1\t', file)
    print(tab_separated)
```

```
miRNA1	TGAGGTAGTAGGTTGTATAGTT
miRNA2	AGAGGTAGTACCATGTATCGTT
miRNA3	TCCCTGAGACCTCAAGTGTGA
miRNA4	TACAAAAGAGTCGCTCTCTTCA
miRNA5	TGAGGTAGTAGGATGTACAGTA
```

Exercise 11c

Write a function, `rna_splicer(sequence)` , that removes potential introns from a sequence. As before, assume introns are bounded by GT and AG. This time, use the non-greedy quantifier (`?`) to remove only the shortest sequences bounded by GT and AG. Does `re.sub()` change the original sequence?

In [ ]:
```
q = 'TCCATCAGTCGGTTCGCCCATCTCAGTGGAAAATGATGCTTAACATTGATGTCTCTGCGACTGCTTTCT
```

To escape special characters that might be part of the pattern we're interested in, we use `\` . For example:

In [ ]:
```
hiseq_bill = 'Reagents, 2 flowcells, technical support, $5,500'
```

In [ ]:
```
hiseq_bill = 'Reagents, 2 flowcells, technical support, $5,500'
```

Within brackets, most characters except `^` are not treated as special.

Just for fun - the `re.finditer()` method

Another method that we won't spend much time on is `re.finditer(pattern, subject)` that produces an iterater that you can loop over in a for loop. For example:

In [ ]:
```
try:
    file_handle = open('miRNAs.fa')
except:
    print('error in open')
with file_handle:
    file = file_handle.read()
    let7 = re.finditer(r'\b.GAGGTAG.*', file)
    for match in let7:
        print(match)
        print(match.group())
        print(match.start())
        print(match.end())
```

Sneap peak: Command line arguments

Up until now, to pass arguments to a function from the command line, we've relied on the `input()` function. We'll look at a couple different ways of passing arguments from the command line starting with `sys.arg`. `sys.argv` is a list that contains the argument passed to Python via the command line.

Copy the following code into a file:

```
import sys # we first have to import the sys module

def command_line_arguments(arg1):
    """Print argument passed from the command line"""
    print("Command line argument: " + arg1)

if __name__ == '__main__' :
    argument = sys.argv[1] # sys.argv[1] is the second argument o
n the command line
    command_line_arguments(argument)
```

We can pass multiple arguments from the command line and each will be stored in the `sys.arvg` list.

Copy the following code into a file:

```
import sys

def command_line_arguments(arg1, arg2, arg3):
    """Print three arguments passed from the command line"""
    print("Command line arguments:", arg1, arg2, arg3)

if __name__ == '__main__' :
    argument1, argument2, argument3 = sys.argv[1:] # a list slice
    command_line_arguments(argument1, argument2, argument3)
```

Additional Exercises

Answer to exercise 11e below.

11e) Write a function, `pygrep(pattern, file)` , that replicates what the unix commmand grep does - search for a pattern in a file and return each line of the file that contains the pattern. The function should be contained within a script that prompts the user for a pattern and file name within an infinite loop that exits when the user hits return. Remember to import re.

```
In [ ]:
```

11f) As the course comes to an end, take some time to go back over the notebooks, exercises, and assignments. Make yourself a cheat sheet with useful functions, methods, and approaches we used for doing things such that if you don't find yourself using Python regularly, you won't have to go searching through your Notebooks to figure out how to do things. For example, what's the proper way of opening files? Reading files? Writing to files? Searching dictionaries? Creating lists? Tuples? etc.

11g) Make yourself a generic template for opening a file, looping through the file line by line, and writing to a new file. Arguments should be passed to the file via the command line.

```python
"""
Solution to exercise 11e
"""
def pygrep(pattern, file):
    """
    Searches for a pattern within a file.
    Usage: python3 pygrep.py
    Input file name and pattern at prompt.
    """
    import re
    try:
        file_handle = open(file)
    except:
        print('file not found')
        return
    with file_handle:
        for line in file_handle:
            if re.search(pattern, line):
                print(line, end = '')
            else:
                continue

if __name__ == '__main__':
    while True:
        pattern = input('Pattern: ')
        file = input('File: ')
        if pattern == '' or file == '': break
        pygrep(pattern, file)
```