Python 7

Regular Expressions

Regular Expressions is a language for pattern matching. Many different computer languages incorporate regular expressions, as do some unix commands, like grep and sed. So far we have seen a few functions for finding exact matches in strings, but this is not always sufficient.

Functions that utilize regular expressions allow for non-exact pattern matching.

These specialized functions are not included in the core of Python. We need to import them by typing
`import re`
at the top of your script

```
#!/usr/bin/env python3

import re
```

First we will go over a few examples then go into the mechanics in more detail.

Let's start simple and find an exact match for the EcoRI restriction site in a string.

```
>>> dna =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAA
AGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAAGACAGTCTTCTAATTCCT
CATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> if re.search(r"GAATTC",dna):
...   print("Found an EcoRI site!")
...
Found an EcoRI site!
>>>
```

> Since we can search for control characters like a tab (\t), it is good to get in the habit of using the raw string function
> `r`
> when defining patterns.

> Here we used the `search()` function with two arguments, 1) our pattern and 2) the string we want to search.

Let's find out what is returned by the `search()` function.

```
>>> dna =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAA
AGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAAGACAGTCTTCTAATTCCT
CATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> found=re.search(r"GAATTC",dna)
>>> print(found)
<_sre.SRE_Match object; span=(70, 76), match='GAATTC'>
```

> Information about the first match is returned

How about a non-exact match. Let's search for a pattern in our sequence that has to match the following criteria:

- G or A
- followed by a C
- followed by one of anything or nothing
- followed by a G

This could match any of these:

- GCAG
- GCTG
- GCGG
- GCCG
- GCG
- ACAG
- ACTG
- ACGG
- ACCG
- ACG

We could test for each of these, or use regular expressions. This is exactly what regular expressions can do for us.

```
>>> dna =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAA
AGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAAGACAGTCTTCTAATTCCT
CATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> found=re.search(r"[GA]C.?G",dna)
>>> print(found)
<_sre.SRE_Match object; span=(7, 10), match='ACG'>
```

> Here you can see in the returned information that ACG starts at string postion 7 (nt 8).
>
> The first position following the end of the match is at string postion 10 (nt 11).

What about other potential matches in our DNA string? We can use `findall()` function to find all matches.

```
>>> dna =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAA
AGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAAGACAGTCTTCTAATTCCT
CATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> found=re.findall(r"[GA]C.?G",dna)
>>> print(found)
['ACG', 'GCTG', 'ACTG', 'ACCG', 'ACAG', 'ACCG', 'ACAG']
```

> `findall()` returns a list of all the pieces of the string that match the regex.

A quick count of all the matching sites can be done by counting the length of the returned list.

```
>>> len (re.findall(r"[GA]C.?G",dna))
7
```

> There are 7 sites that match our pattern.
>
> Here we have another example of nesting.
>
> We call the `findall()` function, searching for all the matches.
>
> This function returns a list, the list is past to the `len()` function, which in turn returns the number of elements in the list.

Let' Try It

1. If you want to find just the first occurrence of a pattern, what method do you use?
2. If you want to find all the occurrences of a pattern, what method do you use?
3. What operator have we seen that will report if an exact match is in a sequence (string, list, etc)?
4. What string method have we seen that will count the number of occurrences of an exact match in a string?

Let's talk a bit more about all the new characters we see in the pattern.

The pattern in made up of atoms. Each atom represents **ONE** character.

Individual Characters

Atom	Description
a-z, A-Z, 0-9 and some punctuation	These are ordinary characters that match themselves
"."	The dot, or period. This matches any single character except for the newline.

Character Classes

A group of characters that are allowed to be matched one time. There are a few predefined classes, which are symbols that means a series of characters.

Atom	Description
`[ ]`	A bracketed list of characters, like `[GA]`. This indicates a single character can match any character in the bracketed list.
`\d`	Digits. Also can be written `[0-9]`
`\D`	Not digits. Also can be written `[^0-9]`
`\w`	Word character. Also can be written `[A-Za-z0-9_]` Note underscore is part of this class
`\W`	Not a word character, or `[^A-Za-z0-9_]`
`\s`	White space character. Also can be written `[ \r\t\n]`. Note the space character after the first `[`
`\S`	Not whitespace. Also `[^ \r\t\n]`
`[^]`	a carat within a bracketed list of characters indicates anything but the characters that follows

Anchors

A pattern can be anchored to a region in the string:

Atom	Description
`^`	Matches the beginning of the string
`$`	Matches the end of the string
`\b`	Matches a word boundary between `\w` and `\W`

Examples:

```
g..t
```

> matches "gaat", "goat", and "gotta get a goat" (twice)

```
g[gatc][gatc]t
```

> matches "gaat", "gttt", "gatt", and "gotta get an agatt" (once)

```
\d\d\d-\d\d\d\d
```

> matches 867-5309, and 5867-5309 but not 8-67-5309.

```
^\d\d\d-\d\d\d\d
```

> matches 867-5309 and 867-53091 but not 5867-5309.

```
^\d\d\d-\d\d\d\d$
```

> only matche 3 digits followed by a dash followed by 4 digits, not extra characters anywhere are allowed

Quantifiers

Quantifiers quantify how many atoms are to be found. By default an atom matches only once. This behaviour can be modified following an atom with a quantifier.

Quantifier	Description
`?`	atom matches zero or exactly once
`*`	atom matches zero or more times
`+`	atom matches one or more times
`{3}`	atom matches exactly 3 times
`{2,4}`	atom matches between 2 and 4 times, inclusive
`{4,}`	atom matches at least 4 times

Examples:

```
goa?t
```

> matches "goat" and "got". Also any text that contains these words.

```
g.+t
```

> matches "goat", "goot", and "grant", among others.

```
g.*t
```

> matches "gt", "goat", "goot", and "grant", among others.

```
^\d{3}-\d{4}$
```

> matches US telephone numbers (no extra text allowed).

Let' Try It

1. What would be a pattern to recognize an email address?
2. What would be a pattern to recognize the ID portion of a sequence record in a FASTA file?

Variables and Patterns

Variables can be used to store patterns.

```
>>> pattern = r"C[ATC]G"
>>> len (re.findall(pattern,dna))
7
```

> In this example, we stored our pattern for a CHG methylation site (where H correspond to A, T or C) in the variable named 'pattern' and used it as the first argument to `findall`.

Either Or

A pipe '|' can be used to indicated that either the pattern before or after the '|' can match. Enclose the two options in parenthesis.

```
big bad (wolf|sheep)
```

> This pattern must match a string that contains:
>
> - "big" followed by a space followed by

> - "bad" followed by
> - a space followed by
> - *either* "wolf" or "sheep"
>
> This would match:
>
> - "big bad wolf"
> - "big bad sheep"

Let' Try It

1. What would a pattern to match 'ATG' followed by a C or a T look like?

Subpatterns

Subpatterns, or parts of the pattern enclosed in parenthesis can be extracted and stored for later use.

```
Who's afraid of the big bad w(.+)f
```

> This pattern has only one subpattern (.+)

You can combine parenthesis and quantifiers to quantify entire subpatterns.

```
Who's afraid of the big (bad )?wolf\?
```

> This matches:
>
> - "Who's afraid of the big bad wolf?"
> - As well as "Who's afraid of the big wolf?".
>
> The 'bad ' is optional, it can be present 0 or 1 times in our string.
>
> This also shows how to literally match special characters. Use a '\' in to escape them.

Let' Try It

1. What pattern could you use to capture the ID in a sequence record of a FASTA file in a subpattern.

Example FASTA sequence record.

```
>ID Optional Descrption
SEQUENCE
SEQUENCE
SEQUENCE
```

Using Subpatterns Inside the Regular Expression Match

This is helpful when you want to find a subpattern and then match the contents again. They can be used within the function call and used after the function call.

Subpatterns within the function call

Once a subpattern matches, you can refer to it within the same regular expression. The first subpattern becomes \1, the second \2, the third \3, and so on.

```
Who's afraid of the big bad w(.)\1f
```

> This would match:
>
> - "Who's afraid of the big bad woof"
> - "Who's afraid of the big bad weef"
> - "Who's afraid of the big bad waaf"
>
> But Not:
>
> - "Who's afraid of the big bad wolf"
> - "Who's afraid of the big bad wife"

In a similar vein,

```
\b(\w+)s love \1 food\b
```

> This pattern will match
>
> - "dogs love dog food"
> - But not "dogs love monkey food".
>
> We were able to use the subpattern within the regular expression by using `\1`
>
> If there were more subpatterns they would be `\2`, `\3` , `\4`, etc

Using Subpatterns Outside the Regular Expression

Subpatterns can be retrieved after the `search()` function call, or outside the regular expression, by using the `group()` method. This is a method and it belongs to the object that is returned by the `search()` function.

The subpatterns are retrieved by a number. This will be the same number that could be used within the regular expression, i.e.,

- `\1` within the subpattern can be used outside with `search_found_obj.group(1)`
- `\2` within the subpattern can be used outside with `search_found_obj.group(2)`
- `\3` within the subpattern can be used outside with `search_found_obj.group(3)`
- and so on

Example:

```
>>> dna =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAA
AGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTA
ATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> found=re.search( r"(.{50})TATTAT(.{25})"  , dna )
>>> upstream = found.group(1)
>>> print(upstream)
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
>>> downstream = found.group(2)
>>  print(downstream)
CCGGTTTCCAAAGACAGTCTTCTAA
```

> 1. This pattern will recognize a consensus transcription start site (TATTAT)
> 2. And store the 50 base pairs upstream of the site
> 3. And the 25 base pairs downstream of the site

If you want to find the upstream and downstream sequence of ALL 'TATTAT' sites, use the `findall()` function.

```
>>>
dna="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTT
CCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCT
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCA
AATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTT
TATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCT
CTGG"
>>> found = re.findall( r"(.{50})TATTAT(.{25})"  , dna )
>>> print(found)
[('TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA', 'CCGGTTTCCAAAGACAGTCTTCTAA'),
('TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA', 'CCGGTTTCCAAAGACAGTCTTCTAA')]
```

> The subpatterns are stored in tuples within a list. More about this type of data structure later.

Another option for retrieving the upstream and downstream subpatterns is to put the `findall()` in a for loop

```
>>>
dna="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTT
CCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCT
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCA
AATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTT
TATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCT
CTGG"
>>> for (upstream, downstream) in re.findall( r"(.{50})TATTAT(.{25})"  , dna ):
...   print("upstream:" , upstream)
...   print("downstream:" , downstream)
...
upstream: TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
downstream: CCGGTTTCCAAAGACAGTCTTCTAA
upstream: TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
downstream: CCGGTTTCCAAAGACAGTCTTCTAA
```

1. This code executes the `findall()` function once
2. The subpatterns are returned in a tuple
3. The subpatterns are stored in the variables upstream and downstream
4. The for block of code is executed
5. The `findall()` searches again
6. A match is found
7. New subpatterns are returned and stored in the variables upstream and downstream
8. The for block of code gets executed again
9. The `findall()` searches again, but no match is found
10. The for loop ends

Another way to get this done is with an iterator, use the `finditer()` function in a for loop. This allows you to not store all the matches in memory. `finditer()` also allows you to retrieve the postion of the match.

```
>>>
dna="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTT
CCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCT
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCA
AATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTT
TATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCT
CTGG"
>>> for match in re.finditer(r"(.{50})TATTAT(.{25})"  , dna):
...   print("upstream:" , match.group(1))
...   print("downstream:" , match.group(2))
...
upstream: TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
downstream: CCGGTTTCCAAAGACAGTCTTCTAA
upstream: TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA
downstream: CCGGTTTCCAAAGACAGTCTTCTAA
```

1. This code executes `finditer()` function once.
2. The match object is returned. A match object will have all the information about the match.
3. In the for block we call the `group()` method on the first match object returned
4. We print out the first and second subpattern using the `group()` method
5. The `finditer()` function is executed a second time and a match is found
6. The second match object is returned
7. The second subpatterns are retrieved from the match object using the `group()` method
8. The `finditer()` function is executed again, but no matches found, so the loop ends

Get position of the subpattern with `finditer()`

The match object contains information about the match that can be retrieved with match methods like `start()` and `end()`

```
#!/usr/bin/env python3

import re

dna="ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTT
CCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCT
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCA
AATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTT
TATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCT
CTGG"

for found in re.finditer(r"(.{50})TATTAT(.{25})"  , dna):
  whole    = found.group(0)
```

```
  up       = found.group(1)
  down     = found.group(2)
  up_start = found.start(1) + 1   # need to convert from 0 to 1 notation
  up_end   = found.end(1)
  dn_start = found.start(2) + 1
  dn_end   = found.end(2)

  print( whole , up , up_start, up_end , down , dn_start , dn_end , sep="\t" )
```

> we can use these match object methods `group()`, `start()`, `end()` to get the string, start position, and end position of each subpattern.

```
$ python3 re.finditer.pos.py
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAA
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA  98  148 CCGGTTTCCAAAGACAGTCTTCTAA 154
179
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAA
TCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGA  320 370 CCGGTTTCCAAAGACAGTCTTCTAA 376
401
```

FYI: `match()` function is another regular expression function that looks for patterns. It is similar to search but it only looks at the beginning of the string for the pattern while `search()` looks in the entire string. Usually `finditer()` , `search()`, and `findall()` will be more useful.

Subpatterns and Greediness

By default, regular expressions are "greedy". They try to match as much as they can. Use the quantifier '?' to make the match not greedy. The not greedy match is called 'lazy'

```
>>> phrase = 'The fox ate my box of doughnuts'
>>> found = re.search(r"(f.+x)",phrase)
>>> print(found.group(1))
fox ate my box
```

> The pattern f.+x does not match what you might expect, it matches past 'fox' all the way out to 'fox ate my box'. The '.+' id greedy As many characters as possible are found that are between the 'f' and the 'x'.

Let's make this match lazy by using '?'

```
>>> found = re.search(r"(f.+?x)",phrase)
>>> print(found.group(1))
fox
```

> The match is now lazy and will only match 'fox'

Practical Example: Codons

Extracting codons from a string of DNA can be accomplished by using a subpattern in a `findall()` function. Remember the `findall()` function will return a list of the matches.

```
>>> dna = 'GTTGCCTGAAATGGCGGAACCTTGAA'
>>> codons = re.findall(r"(.{3})",dna)
>>> print(codons)
['GTT', 'GCC', 'TGA', 'AAT', 'GGC', 'GGA', 'ACC', 'TTG']
```

Or you can use a for loop to do something to each match.

```
>>> for codon in re.findall(r"(.{3})",dna):
...   print(codon)
...
GTT
GCC
TGA
AAT
GGC
GGA
ACC
TTG
>>>
```

> `finditer()` would also work in this for loop.
> Each codon can be accessed by using the `group()` method.

Truth and Regular Expression Matches

The `search()`, `match()`, `findall()`, and `finditer()` can be used in conditional tests. If a match is not found an empty list or 'None' is returned. These are both False.

```
>>> found=re.search( r"(.{50})TATTATZ(.{25})"  , dna )
>>> if found:
...    print("found it")
... else:
...    print("not found")
...
not found
>>> print(found)
None
```

> None is False so the else block is executed and "not found" is printed

Nest it!

```
>>> if re.search( r"(.{50})TATTATZ(.{25})"  , dna ):
...     print("found it")
... else:
...     print("not found")
...
not found
```

Using Regular expressions in substitutions

Earlier we went over how to find an **exact pattern** and replace it using the `replace()` method. To find a pattern, or inexact match, and make a replacement the regular expression `sub()` function is used. This function takes the pattern, the replacement, the string to be searched, the number of times to do the replacement, and flags.

```
>>> phrase = "Who's afraid of the big bad wolf?"
>>> re.sub(r'w.+f' , 'goat', phrase)
"Who's afraid of the big bad goat?"
>>> print(phrase)
Who's afraid of the big bad wolf?
```

> The `sub()` function returns "Who's afraid of the big bad goat?"
> The value of variable phrase has not been altered
> The new string can be stored in a new variable for later use.

Let's save the new string that is returned in a variable

```
>>> phrase = "He had a wife."
>>> new_phrase = re.sub(r'w.+f' , 'goat', phrase)
>>> print(new_phrase)
He had a goate.
>>> print(phrase)
He had a wife.
```

> The characters between 'w' and 'f' have been replaced with 'goat'.
> The new string is saved in new_phrase

Using subpatterns in the replacement

Sometimes you want to find a pattern and use it in the replacement.

```
>>> phrase = "Who's afraid of the big bad wolf?"
>>> new_phrase = re.sub(r"(\w+) (\w+) wolf" , r"\2 \1 wolf" , phrase)
>>> print(new_phrase)
Who's afraid of the bad big wolf?
```

> We found two words before 'wolf' and swapped the order.
> \2 refers to the second subpattern
> \1 refers to the first subpattern

Let' Try It

1. How would you use regular expressions to find all occurrences of 'ATG' and replace with '-M-' in this sequence 'GCAGAGGTGATGGACTCCGTAATGGCCAAATGACACGT'?

Regular Expression Option Modifiers

Modifier	Description
`re.I` `re.IGNORECASE`	Performs case-insensitive matching.
`re.M` `re.MULTILINE`	Makes $ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).
`re.S` `re.DOTALL`	Makes a period (dot) match any character, including a newline.
`re.U`	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
`re.X` `VERBOSE`	This flag allows you to write regular expressions that look nicer and are more readable by allowing you to visually separate logical sections of the pattern and add comments. Whitespace within the pattern is ignored, except when in a character class or when preceded by an unescaped backslash. When a line contains a # that is not in a character class and is not preceded by an unescaped backslash, all characters from the leftmost such # through the end of the line are ignored.

```
>>> dna = "atgcgtaatggc"
>>> re.search(r"ATG",dna)
>>>
>>> re.search(r"ATG",dna , re.I)
<_sre.SRE_Match object; span=(0, 3), match='atg'>
>>>
```

> We can make our search case insensitive by using the `re.I` or `re.IGNORECASE` flag.

You can use more than one flag by concatenating them with `|`. `re.search(r"ATG",dna , re.I|re.M)`

Helpful Regex tools

There are a lot of online tools for actually seeing what is happening in your regular expression. Search for `Python Regular Expression Tester`

- regex101
- pyregex
- pythex