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 =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGAC
AGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTA
ATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> 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 =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGAC
AGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTA
ATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> 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 =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGAC
AGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTA
ATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> 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 position 7 (nt 8).
>
> The first position following the end of the match is at string position 10 (nt 11).

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

```
>>> dna =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGAC
AGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTA
ATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> 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 match 3 digits followed by a dash followed by 4 digits, not extra characters anywhere are allowed

Find out about 867-5309 and even more 🎵

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 Description
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 =
'ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGAC
AGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCA
TTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG'
>>> 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 =
"ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGAC
AGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCA
TTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCA
AACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATA
TTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG"
>>> 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 =
"ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGAC
AGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCA
TTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCA
AACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATA
TTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG"
>>> 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 re.findall() function once
2. In the first iteration of the `for` loop, the first set of matched subpatterns are returned as a tuple
3. The subpatterns are stored in the variables upstream and downstream
4. The `for` block of code is executed, and the matches are printed
5. In the second iteration of the loop, the next set of matched subpatterns are returned as a tuple
6. New subpatterns are returned and stored in the variables upstream and downstream
7. The `for` block of code gets executed again
8. There is not a 3rd iteration because there are no more matches
9. The `for` loop ends

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

```
>>> dna =
"ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGAC
AGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCA
TTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCA
AACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATA
TTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG"
>>> 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 =
"ACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCAAACACTTCCAAAAGGAATTCACCGGTTTCCAAAGAC
AGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATATTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCA
TTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGACAAAATACGTTTTGTAAATGTTGTGCTGTTAACACTGCAAATAAACTTGGTAGCA
AACACTTCCAAAAGGAATTCACCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGGATA
TTATCCGGTTTCCAAAGACAGTCTTCTAATTCCTCATTAGTAATAAGTAAAATGTTTATTGTTGTAGCTCTGG"

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 is 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.

You could use this to find all the frame-shifted codons:

```
>>> for start in range(3):
...   print(re.findall(r"(.{,3})", dna[start:]))
...
['GTT', 'GCC', 'TGA', 'AAT', 'GGC', 'GGA', 'ACC', 'TTG', 'AA', '']
['TTG', 'CCT', 'GAA', 'ATG', 'GCG', 'GAA', 'CCT', 'TGA', 'A', '']
['TGC', 'CTG', 'AAA', 'TGG', 'CGG', 'AAC', 'CTT', 'GAA', '']
```

Truth and Regular Expression Matches

The `re.search()`, `re.match()`, `re.findall()`, and `re.finditer()` can be used in conditional tests. When a match is not found:

- `re.search()` and `re.match()` return the special value None.
- `re.findall()` function returns an empty list
- `re.finditer()` returns an *iterator* that will return no values upon evaluation

All of the preceding values will evaluate to `False` in a Boolean context.

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

`None` evaluates to `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 str.replace() method.
To find a pattern, or inexact match, and make a replacement the regular expression re.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 `re.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 combine multiple flags using `|`, e.g., `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

Link to Python 7 Problem Set

Python 8

Data Structures

Sometimes a *simple* list or dictionary just doesn't do what you want. Sometimes you need to organize data in a more *complex* way. You can nest any data type inside any other type. This lets you build multidimensional data tables easily.

List of lists

List of lists, often called a matrix are important for organizing and accessing data

Here's a way to make a 3 x 3 table of values.

```
>>> M = [[1,2,3], [4,5,6], [7,8,9]]
>>> M[1] # second row (starts with index 0)
[4,5,6]
>>> M[1][2] # second row, third element
6
```

Here's a way to store sequence alignment data:

Four sequences aligned:

```
AT-TG
AATAG
T-TTG
AA-TA
```

The alignment in a list of lists.

```
aln = [
['A', 'T', '-', 'T', 'G'],
['A', 'A', 'T', 'A', 'G'],
['T', '-', 'T', 'T', 'G'],
['A', 'A', '-', 'T', 'A']
]
```

Get the full length of one sequence:

```
>>> seq = aln[2]
>>> seq
['T', '-', 'T', 'T', 'G']
```

> Use the outermost index to access each sequence

Retrieve the nucleotide at a particular position in a sequence.

```
>>> nt = aln[2][3]
>>> nt
'T'
```

> Use the outermost index to access the sequence of interest and the inner most index to access the position

Get every nucleotide in a single column:

```
>>> col = [seq[3] for seq in aln]
>>> col
['T', 'A', 'T', 'T']
```

> Retrieve each sequence from the aln list then the 4th column for each sequence.

Lists of dictionaries

You can nest dictionaries in lists as well:

```
>>> records = [
... {'seq' : 'actgctagt', 'accession' : 'ABC123', 'genetic_code' : 1},
... {'seq' : 'ttaggttta', 'accession' : 'XYZ456', 'genetic_code' : 1},
... {'seq' : 'cgcgatcgt', 'accession' : 'HIJ789', 'genetic_code' : 5}
... ]
>>> records[0]['seq']
'actgctagt'
>>> records[0]['accession']
'ABC123'
>>> records[0]['genetic_code']
1
```

> Here you can retrieve the accession of one record at a time by using a combination of the outer index and the key 'accession'

Dictionaries of lists

And, if you haven't guessed, you can nest lists in dictionaries

Here is a dictionary of kmers. The key is the kmer and its values is a list of positions

```
>>> kmers = {'ggaa': [4, 10], 'aatt': [0, 6, 12], 'gaat': [5, 11], 'tgga':
... [3, 9], 'attg': [1, 7, 13], 'ttgg': [2, 8]}
>>> kmers
{'tgga': [3, 9], 'ttgg': [2, 8], 'aatt': [0, 6, 12], 'attg': [1, 7, 13], 'ggaa': [4, 10],
'gaat': [5, 11]}
>>>
>>> kmers['ggaa']
[4, 10]
>>> len(kmers['ggaa'])
2
```

> Here we can get a list of the positions of a kmer by using the kmer as the key. We can also do things to the returned list, like determining its length. The length will be the total count of this kmers.

You can also use the `get()` method to retrieve records.

```
>>> kmers['ggaa']
[4, 10]
>>> kmers.get('ggaa')
[4, 10]
```

> These two statements return the same results, but if the key does not exist you will get nothing and not an error.

Dictionaries of dictionaries

Dictionaries of dictionaries is my favorite!! You can do so many useful things with this data structure. Here we are storing a gene name and some different types of information about that gene, such as its, sequence, length, description, nucleotide composition and length.

```
>>> genes = {
... 'gene1' : {
...     'seq' : "TATGCC",
...    'desc' : 'something',
...     'len' : 6,
... 'nt_comp' : {
...             'A' : 1,
...             'T' : 2,
...             'G' : 1,
...             'C' : 2,
...            }
...   },
...
... 'gene2' : {
...     'seq' : "CAAATG",
...    'desc' : 'something',
...     'len' : 6,
```

```
... 'nt_comp' : {
...           'A' : 3,
...           'T' : 1,
...           'G' : 1,
...           'C' : 1,
...           }
...       }
... }
>>> genes
{'gene1': {'nt_comp': {'C': 2, 'G': 1, 'A': 1, 'T': 2}, 'desc': 'something', 'len': 6, 'seq':
'TATGCC'}, 'gene2': {'nt_comp': {'C': 1, 'G': 1, 'A': 3, 'T': 1}, 'desc': 'something', 'len':
6, 'seq': 'CAAATG'}}
>>> genes['gene2']['nt_comp']
{'C': 1, 'G': 1, 'A': 3, 'T': 1}
```

> Here we store a gene name as the outermost key, with a second level of keys for qualities of the gene, like sequence, length, nucleotide composition. We can retrieve a quality by using the gene name and quality in conjunction.

To retrieve just one gene's nucleotide composition

```
>>> genes['gene1']['nt_comp']
{'C': 2, 'G': 1, 'A': 1, 'T': 2}
```

Alter one gene's nucleotide count with `=` assignment operator:

```
>>> genes['gene1']['nt_comp']
{'C': 2, 'G': 1, 'A': 1, 'T': 2}
>>>
>>> genes['gene1']['nt_comp']['T']=6
>>> genes['gene1']['nt_comp']
{'C': 2, 'G': 1, 'A': 1, 'T': 6}
```

Alter one gene's nucleotide count with `+=` assignment operator:

```
>>> genes['gene1']['nt_comp']
{'C': 2, 'G': 1, 'A': 1, 'T': 6}
>>>
>>> genes['gene1']['nt_comp']['A']+=1
>>>
>>> genes['gene1']['nt_comp']
{'C': 2, 'G': 1, 'A': 2, 'T': 6}
```

To retrieve the A composition of every gene use a for loop.

```
>>> for gene in sorted(genes):
...   A_comp = genes[gene]['nt_comp']['A']
...   print(f"{gene}: As= {A_comp}")
...
gene1: As= 2
gene2: As= 3
```

Building Complex Datastructures

Below is an example of building a list with a mixed collection of value types. Remember that all elements inside a list or dictionary should be the same type. In other words, the values in a list should all be lists or dictionaries or scalar values. This allows you to loop over the data structure.

This is a list with lists and a dictionary. The dictionary has a key with a value that is a dictionary.

```
[
   [1, 2, 3],
   [4, 5, 6],
   {
       'key': 'value',
       'key2':
            {
                'something_new': 'Yay'
            }
    }
 ]
```

Building this data structure in the interpreter:

```
>>> new_data = []
>>> new_data
[]
>>> new_data.append([1,2,3])
>>> new_data
[[1, 2, 3]]
>>> new_data[0]
[1, 2, 3]
>>> new_data.append([4,5,6])
>>> new_data
[[1, 2, 3], [4, 5, 6]]
>>> new_data[1]
[4, 5, 6]
>>> new_data[1][2]
6
>>> new_data.append({})
```

```
>>> new_data
[[1, 2, 3], [4, 5, 6], {}]
>>> new_data[2]['key']='value'
>>> new_data
[[1, 2, 3], [4, 5, 6], {'key': 'value'}]
>>> new_data[2]['key2']={}
>>> new_data
[[1, 2, 3], [4, 5, 6], {'key2': {}, 'key': 'value'}]
>>> new_data[2]['key2']['something_new']='Yay'
>>> new_data
[[1, 2, 3], [4, 5, 6], {'key2': {'something_new': 'Yay'}, 'key': 'value'}]
>>>
```

Same example in a script file: Building Complex Datastructures

Course T-shirt Organization and Counting

We have a spreadsheet of everyone's style, size, color. We want to know how many of each unique combination of style-size-color we need to order

shirts.txt

```
mens  small heather seafoam
womens  medium  Heather Purple
womens  medium  berry
mens  medium  heather coral silk
womens  Small Kiwi
Mens  large Graphite Heather
mens  large sport grey
mens  small Carolina Blue
```

We want something like this:

```
womens  small antique heliconia   2
womens  xs      heather orange      1
womens  medium  kiwi                2
womens  medium  royal heather       1
```

shirts.py

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

shirts = {}
with open("shirts.txt","r") as file_object:
  for line in file_object:
    line = line.rstrip()
    [style, size, color] = line.split("\t")
```

```
    style = style.lower()
    size = size.lower()
    color = color.lower()
    if style not in shirts:
        shirts[style] = {}
    if size not in shirts[style]:
        shirts[style][size] = {}
    if color not in shirts[style][size]:
        shirts[style][size][color] = 0

    shirts[style][size][color] += 1

for style in shirts:
  for size in shirts[style]:
    for color in shirts[style][size]:
      count = shirts[style][size][color]
      print(style,size,color,count,sep="\t")
```

Output:

```
sro$ python3 shirts.py
mens  small heather maroon      1
mens  small royal blue          1
mens  small olive               1
mens  large graphite heather    1
womens  medium  heather purple      3
womens  medium  berry               2
womens  medium  royal heather       1
womens  medium  kiwi                2
...
```

This is what the data structure we just built looks likes

```
{
  'mens':
    {
      'small':
        {
           'heather seafoam': 1,
           'carolina blue': 1,
           'cornsilk': 1,
           'dark heather': 1,
           'heather maroon': 1,
           'royal blue': 1,
           'olive': 1
        },
```

```
      'large':
        {
          'graphite heather': 1,
          'sport grey': 1,
          'heather purple': 1,
          'heather coral silk': 1,
          'heather irish': 1,
          'heather royal': 1,
          'carolina blue': 1
        },
      'medium':
        {
          'heather coral silk': 1,
          'heather royal': 2,
          'heather galapagos blue': 1,
          'heather forest': 1,
          'gold': 1,
          'heather military green': 1,
          'dark heather': 1,
          'carolina blue': 1,
          'iris': 1
        },
      'xs':
        {
          'white': 1
        },
      'xl':
        {
          'heather cardinal': 1,
          'indigo blue': 1
        }
          },
  'womens':
    {
      'medium':
        {
          'heather purple': 3,
          'berry': 2,
          'royal heather': 1,
          'kiwi': 2,
          'carolina blue': 1
        },
      'small':
        {
          'kiwi': 1,
          'berry': 1,
          'antique heliconia': 2
        },
      'large':
```

```
        {
          'kiwi': 1
        },
      'xs':
        {
          'heather orange': 1
        }
    },
  'child':
    {
      '4t':
        {
          'green': 2
        },
      '3t':
        {
          'pink': 1
        },
      '2t':
        {
          'orange': 1
        },
      '6t':
        {
          'pink': 1
        }
    }
}
```

There are also specific data table and frame handling libraries like Pandas.

Here is a intro to data structures in Panda.

Here is a very nice interactive tutorial

Link to Python 8 Problem Set

Python 9

Exceptions

There are a few different types of errors when coding. Syntax errors, logic errors, and exceptions. You have probably encountered all three. Syntax and logic errors are issues you need to deal with while coding. An exception is a special type of error that can be informative and used to write code to respond to this type of error. This is especially relevant when dealing with user input. What if they don't give you any, or it is the wrong kind of input? We want our code to be able to detect these types of errors and respond accordingly.

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

import sys
file = sys.argv[1]

print("User provided file:" , file)
```

> This code takes user provided input and prints it

Run it.

```
$ python scripts/exceptions.py test.txt
User provided file: test.txt
```

What happens if the user does not provide any input and we try to print it?

```
$ python scripts/exceptions.py
Traceback (most recent call last):
  File "scripts/exceptions.py", line 4, in <module>
    file = sys.argv[1]
IndexError: list index out of range
```

> We get an **IndexError** exception, which is raised when an index is not found in a sequence.

We have already seen quite a few exceptions throughout the lecture notes, here are some:

- ValueError: math domain error
- AttributeError: 'list' object has no attribute 'rstrip'
- SyntaxError: EOL while scanning string literal
- NameError: name 'GGTCTAC' is not defined
- SyntaxError: Missing parentheses in call to 'print'
- AttributeError: 'int' object has no attribute 'lower'
- IndexError: list assignment index out of range

- NameError: name 'HDAC' is not defined

Link to Python Documentation of built in types of exceptions

We can use the exception to our advantage to help the people who are running the script. We can use a try/except condition like an if/else block to look for exceptions and to execute specific code if we **do not have** an exception and do something different if we **do have** an exception.

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

file = ''
try:
  file = sys.argv[1]
  print("User provided file:" , file)
except:
  print("Please provide a file name")
```

> We need to "try" to get a user provided argument. If we are successful then we can print it out. If we try and fail, we execute the code in the except portion of our try/except and print that we need a file name.

Let's run it WITH user input

```
$ python3 scripts/exceptions_try.py test.txt
User provided file: test.txt
```

> It runs as expected

Let's run it WITHOUT user input

```
$ python scripts/exceptions_try.py
Please provide a file name
```

> Yeah, the user is informed that they need to provide a file name to the script

What if the user provides input but it is not a valid file or the path is incorrect? Or if you want to check to see if the user provided input as well as if it can open the input.

We can add multiple exception tests, like if/elif block. Each except statement can specify what kind of exception it is waiting to receive. If that kind of exception occurs, that block of code will be executed.

```
import sys

file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
```

```
    FASTA = open(file, "r")
    for line in FASTA:
      line = line.rstrip()
      print(line)
except IndexError:
    print("Please provide a file name")
except IOError:
    print("Can't find file:" , file)
```

> Here we test for an IndexError: Raised when an index is not found in a sequence.
> The IndexError occurs when we try to access a list element that does not exists.
> And we test for a IOError: Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.
> The IOError happens when we try to access a file that does not exist.

Let's run it with a file that does not exist.

```
$ python scripts/exceptions_try_files.py test.txt
User provided file name: test.txt
Can't find file: test.txt
```

> This informs the user that they did provide input but that the file listed can not be found.

Let's run it with no input

```
$ python scripts/exceptions_try_files.py
Please provide a file name
```

> This informs the user that they need to provide a file.

try/except/else/finally

Lets summarize what we have covered and add on `else` and `finally`.

```
try:
  # try block is executed until an exception is raised
except _ExceptionType_:
  # if there is an exception of "ExceptionType" this block will be executed
  # there can be more than one except block, just like an elif
except:
  # if there are any exceptions that are not of "ExceptionType" this except block will be executed
else:
  # the else block is executed after the try block has been completed, which means there were no exceptions raised
finally:
  # the finally block is executed if exceptions are or are not raised (no matter what happens)
```

Getting more information about an exception

Some exceptions can be thrown for multiple reasons, for example, ErrorIO will occur if the file does not exist as well as if you don't have permissions to read it. We can get more information by viewing the contents of our Exception Object. Yes, an exception is an object too! The system errors get stored in the exception object. To access the object use `as` and supply a variable name, like 'ex'

```
file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  FASTA = open(file, "r")
  for line in FASTA:
    line = line.rstrip()
    print(line)
except IndexError:
  print("Please provide a file name")
except IOError as ex:
  print("Can't find file:" , file , ': ' , ex.strerror  )
```

> Here we added `except IOError as ex` and now we can get the 'strerror' message from ex.

Run it.

```
$ python scripts/exceptions_try_files_as.py  test.txt
User provided file name: test.txt
Can't find file: test.txt :  No such file or directory
```

> Now we know that this file name or path is not valid

Raising an Exception

We can call or raise exceptions too!! This is accomplished by using a `raise` statement.

1. First, create a new Exception Object, i.e., `ValueError()`
2. Use the Exception Object in a Raise statement `raise ValueError('your message')`

Let's raise an exception if the file name does not end in 'fa'

```
import sys

file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  if not file.endswith('.fa'):
    raise ValueError("Not a FASTA file")
  FASTA = open(file, "r")
  for line in FASTA:
    print(line)
except IndexError:
  print("Please provide a file name")
except IOError as ex:
  print("Can't find file:" , file , ': ' , ex.strerror  )
```

> Here we raise a known exception, 'ValueError', if the file does not end with (uses `endswith()` method).

Let's run it.

```
$ python scripts/exceptions_try_files_raise.py test.txt
User provided file name: test.txt
Traceback (most recent call last):
  File "scripts/exceptions_try_files_raise.py", line 10, in <module>
    raise ValueError("Not a FASTA file")
ValueError: Not a FASTA file
```

> Our exception get's raised, now lets do something with it.

```
import sys

file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  if not file.endswith('.fa'):
    raise ValueError("Not a FASTA file")
  FASTA = open(file, "r")
  for line in FASTA:
    print(line)
```

```
except IndexError:
  print("Please provide a file name")
except ValueError:
  print("File needs to be a FASTA file and end with .fa")
except IOError as ex:
  print("Can't find file:" , file , ': ' , ex.strerror  )
```

> Here we created an exception to catch any ValueError

Let's Run it.

```
$ python scripts/exceptions_try_files_raise_value.py test.txt
User provided file name: test.txt
File needs to be a FASTA file and end with .fa
```

> We get a great error message now.

But what if there is another ValueError, how can we tell if has anything to do with the FASTA file extension or not? Answer: the message will be different.

Creating Custom Exceptions

We can create our own custom exception. We will need to create a new class of exception. Below is the syntax to do this.

```
import sys

class NotFASTAError(Exception):
  pass


file = ''
try:
  file = sys.argv[1]
  print("User provided file name:" , file)
  if not file.endswith('.fa'):
    raise NotFASTAError("Not a FASTA file")
  FASTA = open(file, "r")
  for line in FASTA:
    print(line)
except IndexError:
  print("Please provide a file name")
except NotFASTAError:
  print("File needs to be a FASTA file and end with .fa")
except IOError as ex:
  print("Can't find file:" , file , ': ' , ex.strerror  )
```

> Here we created a new class of exception called 'NotFASTAError'. Then we raised this new exception.

Let's Run it.

```
$ python scripts/exceptions_try_files_raise_try.py test.txt
User provided file name: test.txt
File needs to be a FASTA file and end with .fa
```

> Our new class of exception, NotFASTAError, works just like the built in exceptions.

Link to Python 9 Problem Set