# Lab #3: Sequences

## Purpose

The purpose of this lab is to write Python programs to manipulate strings.

## Tasks

In this lab we will write and run Python programs that use string operations to process sequences.

## Success

To succeed in this lab, we will pay close attention to how sequences are set up as strings. Use as many aspects of regular expressions as possible.

## Introduction

In this lab we will write programs with string operations. There are a number of string operators (just like we have the arithmetic operators, +, -, /, and ** for numbers) in Python and they come in handy when we use sequences:

**+** Concatenation operator – sticks two strings together.
for example: `seq = "ACTTTC" + "TATA"`
If you were to print the **seq** variable, you would see: **ACTTTCTATA**

***** Repeats a sequence a number of times. for example:

```
trinucleotide = "CAG"
seq = trinucleotide * 4
```

If you were to print the **seq** variable, you would see the CAG repeated,

**in** Checks if the string to the left of the **in** operator is a substring that appears in the string to the right. e.g. `'bc' in 'abcd'` is **True**

**not in** Checks if the string to the left of the **not in** operator does not appear in the string to the right. e.g. `'bc' not in 'abcd'` is **False**

`str.find(s)` The Python string method "**find**" searches for the string `s` in the string `str`. If there is a match, **find** will give us a 0 or a positive number indicating where the first match was found; if no matches, **find** will give us a -1. We can use this in an **if** :

```
if ( seq.find("TATA") < 0 ):
    print ( "TATA subsequence not found" )
```

The above **if** statement checks if the string "TATA" appears anywhere in **seq**.

The next program asks the user to enter a string and checks if it has a certain pattern of characters:

```
# String input and matching:
# first, ask the users to enter a string
# store the users input in a string variable
userinput = input( "  Please enter a string: ")
# check if “abcd” is anywhere in the $userinput string
if ( 'abcd' in userinput ):
   print ( "   your string contained the pattern 'abcd'!" )
else:
   print ( "   your string did not have an 'abcd' in it." )
```

The **if** statement checks whether the pattern of an **a** followed by a **b**, a **c** and a **d** was present in the user's input string. Run the above program and enter the following input strings (only 1 string per run):

```
ab
```

```
abc
abcd
1 2 fabcd5 s
```

In the above program, use the **`-W all`** option in the first line of the program to turn on all "warnings" Python may have if it spots potential programming errors on our part. On my system it looks like:

```
#!/usr/bin/python3 -W all
```

It is a good idea to do this so that we can more easily detect and fix problems with our program.

A note on what constitutes "truth" in Python

The **`if`** and **`elsif`** statements and **`while`** and **`for`** loops check the value of an "expression" to see if it is true or false. Many things that are considered to be "true" by Python. It is actually easier to look at what is false:

| | |
|---|---|
| **`False`** | The constant "**`False`**" is false |
| **`0`** | The number 0 is false |
| **`""`** | The empty string is false |
| **`[]`** | The empty list is treated as false |
| **`None`** | The constant "**`None`**" represents an undefined value or absence of a value and is treated as false |

because everything else is true. i.e. **`1`** is true as is **`-1`** and any number except **`0`**. Any string with 1 or more characters is true.

Back to pattern matching: Suppose we wanted to see if a DNA coding sequence had any phenylalanines in it. We can consult the Universal Genetic Code and see that either a TTC or a TTT codon results in a phenylalanine and so checking a DNA sequence string has either one would be enough. This can be done with an `if` statement like:

```
if ( 'TTC' in seq or 'TTT' in seq ):
```

The above statement can be read as: "if the pattern TTC appears in **`seq`** or a TTT appears in it ..."
It turns out that we can use a more complex search technique that involves the use of "regular expressions":

```
import re                          # we need to import the re library
regs = 'TTC|TTT'                   # this is a regular expression
if ( re.search( regs, seq )): # search for TTC or TTT in seq
```

which checks if any of the alternatives, i.e. TTC or TTT separated by the "**`|`**" are present in the string **`seq`**. Since, in this case, the two alternatives have a common substring (TT), we can use an even more succinct match expresssion:

```
regs = 'TT[CT]'
```

which signifies two **`T`**s followed by either a **`T`** or a **`C`**. The **`[CT]`** means either a **`C`** or a **`T`**. In general, we can have many possibilities in square brackets; for example, **`[CTG]`** means either a **`C`** or a **`T`** or a **`G`**.

**1** Write a Python program that will read a DNA sequence from a file as a single string and checks if the sequence contains a codon for Leucine. You do not have to worry about codon frames and you can assume the entire sequence fits on a single line in the file and that the file only has 1 line in it. Look in Lab #2 if you do not remember how to read from a file.

**String substitution**

Besides searching for substrings, we can also modify a string. The `string.replace(` *something1*, *something2 )* function finds all substrings that match "*something1*" and replaces them with "*something2*". For example:

```
dna = "ATTTCAGCATTG"
rna = dna.replace( 'T', 'U' )
```

If you do this, you may notice that all the upper-case **T**s will be switched to upper-case **U**s. If we only want to replace the first 3 occurrences of **T**s with **U**s, we would use:

```
rna = dna.replace( 'T', 'U', 3 )
```

If we happened to have lowercase **t**s, the above replace will not work. To handle lower-case letters, we could simply do another substitution specifically for the lower-case letters:

```
rna = rna.replace( 't', 'u' )
```

Note that we did a **dna.replace** the first time and a **rna.replace** the second time. This is because Python strings are "immutable" - once we create a string, we cannot really change the internal characters inside it. The **dna.replace** function call gives or "returns" a new string that we store in the variable, **rna** – this function will not modify the variable, **dna**. The **rna.replace** function call returns a new string but we go ahead and "store" it in the variable, **rna**. This is a subtle difference but one that is very important that we recognize as it could lead to horrible bugs if we do not keep these things in mind.

Suppose we wanted to swap all guanines in a sequence with cytosines. i..e. convert "AGTCGC" to "ACTGCG". We might be tempted to use something like:

```
dna = "AGTCGC"
swapped = dna.replace( 'G', 'C' )
swapped = dna.replace( 'C', 'G' )
```

At first glance it looks like the first substitution will convert the Gs to Cs and the second will convert Cs to Gs. Instead of what we want, we will end up with the sequence: "AGTGGG". Why?

Hint: if we try the opposite order of replaces:

```
dna = "AGTCGC"
swapped = dna.replace( 'C', 'G' )
swapped = dna.replace( 'G', 'C' )
```

we end up with the sequence "ACTCCC".

We could try using the newly created **swapped** variable for the second **replace** function:

```
dna = "AGTCGC"
swapped = dna.replace( 'G', 'C' )
swapped = swapped.replace( 'C', 'G' )
```

A good try but this does not work either – make sure you understand why none of these work.

It turns out that there is no easy way to use the replace function for what we want to do. Instead we need to use the "**translate**" function. To convert all Xs to As, all Ys to Bs and all Zs to Cs in a string, we need to set up a translation table with "XYZ" as the *input* string and "ABC" as the *output* string.

```
s = ""   # used so we can call maketrans
tableinput  = 'XYZ'
tableoutput = 'ABC'
table = s.maketrans( tableinput, tableoutput )
```

In the case of the DNA sequence, we should use:

```
ttable = s.maketrans( "GC", "CG" )
```

and then we use this table in the translate method:

```
swapped = dna.translate( ttable )
```

which will convert all Gs and all Cs to Cs and Gs respectively without the problems of substitution.

**2** Write a program that will ask the user to enter a sequence and then compute and print the reverse complement of what the user entered. Use the technique of treating a string as a list or an array and using array indexing as in:

```
rev = dna[::-1]
```

to create a new string with the opposite direction of the original string.

## FASTA files

Remember from chapter 1 of the textbook (p. 3 of Baxevanis, "Bioinformatics" 4th ed., 2020, John Wiley) that the basic flatfile format for sequences is rather simple. The following text saved as a file would satisfy the basic requirements of a FASTA file format:

```
> part of NM_000238.2  Human KCNH2
CCATGGGCTCAGGATGCCGGTGCGGAGGGGCC
ACGTCGCGCCGCAGAACACCTTCCTGGACACCATCATC
CGCAAGTTTGAGGGCCAGAGCCGTAAGT
TCATCATCGCCAACGCTCGGGTGGAGAACTGC
GCCGTCATCT
ACTGCAACGACGGCTTCTGCGAGCTGTGCGGCT
ACTCGCGGGCCGAGGTGATGCAGCGACCCTGCACCTG
```

As long as the first line of the file is a "comment" and starts with a "**>**" and the rest of the file contains letters, the file satisfies the basic FASTA structure. Save the above text in a file named "KCNH2.fasta" in the same place as your script files. This fasta file is also provided in case you have trouble saving the sequence to a file. Then run the following program:

```
# Simple FASTA-type file reader
#  assumes there is only one sequence in the file.
# First ask the user to enter the name of the file to be read:
#
print ( "Welcome to the FASTA file reader program" )
filename = input( " Name of file? " )
#
# Now try to open the file
try:
    inpf = open( filename, "r" )
except IOError as e:
    print ( "Unable to open ", filename, ". Exiting program." )
    exit()
```

```
seq_info = inpf.readline()                  # Read the first line
print ( " Information about file:",seqinfo ) #  and print it out

data = inpf.readlines()                 # read in the remaining data
inpf.close()
print ( data )
```

The **try:** statement means that the indented lines after it may result in an error and if there is an error of the kind listed in the **except** part (we check for **IOError** in the above program), then the statements in the **except** part will get executed. If there were no errors, we skip past the **except** part. An error would occur if we run the program and, when asked for a "filename", we give it the name of a file that does not exist – try it and see what happens.

If everything is set up properly, this is the kind of thing you would see if you were to run the program:

```
tony@core2$ ./lab3n11string.py
Welcome to the FASTA file reader program
Filename? KCNH2.fasta
Information about file:  > part of NM_000238.2  Human KCNH2 DNA sequence

 And the actual sequence data:
['CCATGGGCTCAGGATGCCGGTGCGGAGGGGCC\n',
'ACGTCGCGCCGCAGAACACCTTCCTGGACACCATCATC\n', 'CGCAAGTTTGAGGGCCAGAGCCGTAAGT\
n', 'TCATCATCGCCAACGCTCGGGTGGAGAACTGC\n', 'GCCGTCATCT\n',
'ACTGCAACGACGGCTTCTGCGAGCTGTGCGGCT\n',
'ACTCGCGGGCCGAGGTGATGCAGCGACCCTGCACCTG\n']
Done.
tony@core2$
```

Note in the above run that the sequence is stored in a list and is broken up into the various lines from the input file. In addition, there is a "new line" character (made visible as the `'\n'` character in the program output. We can combine the different lines into a single sequence and get rid of the newlines using code like:

```
data = inpf.readlines()
seq = ''
seq = seq.join( data )
seq = seq.replace( '\n', '' )
```

**3** Extend the FASTA file reader program: after reading in the sequence from a FASTA file and putting the sequence information into a single string, ask the user for a short subsequence to search for. Then search big sequence for occurrences of the short sequence. Use a loop to ask the user repeatedly for things to search for. Provide a way for the user to stop searching as well.

## Regular expressions

Strings like "**TTC|TTT**" and "**TT[CT]**" are examples of a group of expressions called regular expressions. The simplest regular expressions are "normal" strings like 'A' or 'cat' – i.e. all normal strings can be used as regular expressions but they do not have the power or flexibility of regular expressions. Regular expressions are a good way to describe complex patterns in strings and since DNA and protein sequences can be treated as strings, using regular expressions has been a popular way to analyze biological sequence data. Usually, we will set up regular expressions with special characters like **|**, **[**, or **]** as we did above. Special characters or "metacharacters" change the meaning of characters around them.

One simple operation is searching:

```
if ( re.search( 'TATA', dnaseq ) ):
   print ( "Motif TATA occurs in sequence!" )
```

This could also be written as:

```
motif = "(TA){2}";
if ( re.search( motif, dnaseq ) ):
   print ( "Motif TATA occurs in sequence!" )
```

In the second version, we use a string to store the motif sequence we are looking for and this is generally a better idea than having the same motif appear as a string in different places in a Python program. To search for 2 or more occurrences of **TA**, i.e. **TATA** , or **TATATA**, or **TATATATA**, etc., we would use the regular expression: **(TA){2,*}**.
Think about how we could search a sequence for a particular trinucleotide repeat? A trinucleotide repeat is a pattern of 2 or more sets of 3 nucleotides.

## **Metacharacters in** Python

When using regular expressions for pattern matching, we use a number of special characters or "metacharacters". Some of the "metacharacters" we will use are given below (you do not have to memorize this list – keep it handy for reference):

**** the backslash or "escape" character changes the way the next character is interpreted. An `"n"` is just a string with the letter n but `"\n"` is the newline character; it is not a pair of characters.

**|** the pipe character means "or". To search for either `TTC` or `TTT`, use `'TTC|TTT'`

**()** A grouping metacharacter pair – used to group any part of a regular expression
`(TA){2}` is TATA
`((TA){2}GC)` is TATAGC

**[]** a character set. A string of the form **[ ... ]** is read "one of the characters in `[]`".
`[AT]` is the same as (A|T). We can also have a range of characters in square brackets:
`[A-Z]` means a single uppercase letter between A and Z.
`[0-9]` means any single digit
`[A-Z0-9]` means a single character that is either an uppercase letter or a single digit
`[0-2][0-9]` means a 2 digit number from 00 to 29
To search for an **A** followed by either a single **T** or a single **C** followed by a **G**, we would use: **`'A[TC]G'`**. Other special characters lose their meaning inside a `[]` except for the **^** character when it is the first one inside the `[]` when it means "not". For example `[^0]` means any character other than a 0.

**{}** Used to specify the number of times the previous "thing" occurs.
`A{2}` means AA
`A{2,5}` means any number between a minimum of 2 and a maximum of 5 As.
i.e. 2, 4, or 5 As: AA or AAA or AAAA or AAAAA.
`(TA){2}` means TATA
`(TA){3,}` means 3 or more TAs in a row.

`(TA){,3}` means 3 or fewer TAs in a row

**^** Position at the beginning of a string. The **^** is read as "starting with".
If we want to see if a sequence has an ATG anywhere in it, we can use the `search` function with the string **`'ATG'`**. On the other hand, if we want to see if the sequence starts with an ATG, we would use the `search` function with the string **`'^ATG'`** or the `match` function with the string **`'ATG'`**.

**$** Position at the end of a string
If we want to see if there is a "TTA" at the end of a sequence, we use **`'TTA$'`**.

***** "Zero or more" of the previous. **b*** Means **b**, **bb**, or **bbb**, etc. or the empty string

**+** "One or more" of the previous. **b+** Means **b**, **bb**, or **bbb**, etc.

**?** "None or one" of the previous. `A?` means nothing or one A
`(ATG)?` means either one or no ATGs.

**.** Any character except a newline
This character can be used to match any character except a newline and so we can use the regular expression **`A.G`** to search for an **A** followed by any single character (except a newline) followed by a **G**. To search for an **A** followed by 2 or more characters followed by a **G** we would use **`A.{2,}G`**.

To search for an **A** followed by zero or more **T**s followed by a **G**, we would use: **`'AT*G'`**.
The ***** metacharacter means "zero or more" here (it does not mean multiply here).

Combinations like **`*?`**, **`+?`**, **`??`**, **`.*?`**, and **`{m,n}?`** are used for "non-greedy" string matching. Other combinations like **`(?`** … **`)`** are not used here.

**Compiling regular expressions**
When using regular expressions, some functions require that we process or "compile" the regular expression first, before using it. All we have to do is to call the re.compile function with the regular expression. For example:

```
pattern = re.compile( '(TA){2}' )
```

Once we do this, we can use the `pattern` object to do things like search or replace.

**Substitution**
Just as the replace function (`s.replace( "…", "…")` does not change the string, `s`, the **`re.sub`** function finds all substrings where the regular expression matches and replaces them with another substring without modifying the original string:

```
motif1 = re.compile( "(TA){2}" )
motif2 = "TAAT"
newseq = motif1.sub( motif2, dna )
```

The above code will create a new string called `newseq` which will have a copy of the string `dna` but with all occurrences of "TATA" in the `dna` string replaced by "TAAT". In general, we can search for a subsequence using regular expressions like those above and then replace the matches with some other sequence.

**Translation**

Besides matching and substitution, another useful operation is the "translation" function, **`translate`**:

```
ttable = s.maketrans( "T", "U" )
seq = dna.translate( ttable )
```

The above operation will change all **T**s in dna to **U**s. This is useful for transcribing DNA to RNA. We can translate any set of characters to any other set (of the same size as the first set) of characters. For example: **`tr/AC/XY/`** will convert all **A**s to **X**s and all **C**s to **Y**s.

We can convert a DNA sequence to its complementary string using a table with input string **`'ACTG'`** and output string **`'TGAC'`**. Together with the technique to reverse strings, we can convert any DNA sequence to its reverse complement:

```
transcrtable = s.maketrans( 'ACTG', 'TGAC' )
transc = dna.translate( transcrtable )
revcom = transc[::-1]
```

The translate function returns a new string that is the complement and then we reverse it to get the reverse complement.

**Converting strings to arrays of characters**

While regular expressions are very powerful, they cannot do everything. Suppose we wanted to assemble exons from a genomic sequence, we would need access to parts of the sequence and then cut them out and put subsequences together. For such operations, we would want to convert a string into an array and then use parts of the array or loops (see Lab #2 – while and for loops) to go through the array of characters. To do this we could use something like:

```
dna_array = list( dnaseq )
for nuc in dna_array:
    print ( " ", nuc )
```

The list function method creates a list of individual characters from a string: every character in the string becomes a separate "piece" and each of these will be put into a separate array location. We can then process the array using a loop. The above example uses a loop to print each character on a separate line with a space before each one.

**4**

Write a Python program that will read in a FASTA file containing a DNA sequence and count the number of As, Ts, Cs, and Gs and print out this information.
Also print out the locations of any TATA and ATG sequences.
Required: This program should use Python regular expressions rather than external Python libraries
**Submit the 4 programs requested above – the numbers in the blue circles.**

**Some useful Python symbols**:

Alphanumeric metasymbols

| Symbol | Meaning |
|---|---|
| `\0` | Match the null character (ASCII NULL) |
| `\`*NNN* | If *NNN* is an octal number, match the character given in octal, i.e. up to 377 |
| `\`*n* | If *n* is a decimal number, match *n*th previously captured string |
| `\a` | Match the alarm character (BEL) |

| Symbol | Meaning |
|---|---|
| \A | true at the beginning of a string |
| \b | If used in a character range, match the backspace character (BS) |
| \b | True at word boundary |
| \B | True when not at word boundary |
| \cX | Match the control character Control-X |
| \d | Match any digit character |
| \D | Match any nondigit character |
| \e | Match the escape character (ASCII ESC, not backslash) |
| \E | End case (\L, \U) or metaquote (\Q) translation |
| \f | Match the formfeed character (FF) |
| \G | true at end-of-match position of prior m//g |
| \l | Lowercase the next character only |
| \L | Lowercase till \E |
| \n | Match the newline character (usually NL, but CR on Macs) |
| \Q | Quote (do-meta) metacharacters till \E |
| \r | Match the return character (usually CR, but NL on Macs) |
| \s | Match any whitespace character |
| \S | Match any nonwhitespace character |
| \t | Match the tab character (HT) |
| \u | Titlecase the next character only |
| \U | Uppercase (not titlecase) till \E |
| \w | Match any "word" character (alphanumerics plus _ ) |
| \W | Match any nonword character |
| \x{abcd} | Match the character given in hexadecimal |
| \z | true at end of string only |
| \Z | true at end of string or before optional newline |

Pattern Modifiers

| Modifier | Meaning |
|---|---|
| re.I | Ignore upper- or lowercase distinctions |
| re.s | Let . match newline |
| re.m | Let ^ and $ match next to embedded \n |

**Python raw strings**

Sometimes we don't want to metacharacters to have their special meanings and in such cases, we would want to use Python “raw” strings with a leading **r**. For example, **r'\n'** is a string with two characters, a **** followed by a **n**.