Strings

A string is simply a sequence of characters. From a biological perspective, this is quite useful, as a DNA sequence is simply a string composed of only 4 letters, and thus easily manipulated in Python. In this tutorial, we'll review many of the python features we covered in previous tutorials, and introduce some additional functionality for working with strings.

In [1]:
```
dna = 'ATGTAGC'
print(dna)
```

```
ATGTAGC
```

Looping

You can loop over the characters of a string using a for loop, as we saw on Tuesday:

In [2]:
```
for nt in dna:
    print(nt)
```

```
A
T
G
T
A
G
C
```

Let's write a function, `reverse()` , that returns the reverse of a DNA or RNA sequence:

In [9]:
```
def reverse(seq):
    rev = ''
    for nt in seq:
        rev = nt + rev
        print(rev)
    return rev

reverse('ATGCT')
```

```
A
TA
GTA
CGTA
TCGTA
```

Out[9]:
```
'TCGTA'
```

Let's write a function, `nt_counter()`, that prompts the user for a DNA sequence and returns the number of As, Cs, Ts, and Gs it contains:

In [10]:
```
def nt_counter():
    seq = input('Enter a sequence: ')
    a = 0
    c = 0
    g = 0
    t = 0
    for nt in seq:
        if nt == 'A' or nt == 'a':
            a += 1
        elif nt == 'T':
            t += 1
        elif nt == 'C':
            c += 1
        elif nt == 'G':
            g += 1
        else:
            return 'Invalid sequence'
    return 'A:', a, 'T:', t, 'C:', c, 'G:', g

nt_counter()
```

```
Enter a sequence: AATTTCCCCGGGGG
```

Out[10]:
```
('A:', 2, 'T:', 3, 'C:', 4, 'G:', 5)
```

Python has a lot of built-in functionality for working with strings. We've already seen the length function (`len()`):

In [11]:
```
len('ATGC')
```

Out[11]:
```
4
```

Slicing strings

We can easily capture substrings within a string using the *bracket* operator:

In [13]:
```
dna = 'ATGTTT'
dna[0:3]
```

Out[13]:
```
'ATG'
```

The number in brackets is called the *index*.

Note that in most programming languages indexing of strings and other objects starts at 0. A nice explanation why is given here (http://www.cs.utexas.edu/~EWD/ewd08xx/EWD831.PDF). (The writer is Edsger W. Dijkstra, one of the founding fathers of computer science). Thus, the largest value an index can be is `len(string) - 1`:

In [21]:
```
dna = 'ATGTACA'
#print(len(dna))
print(dna[6:])
```

```
A
```

If we want to count from the end of a string, we can use a negative value as our index:

In [22]:
```
dna[-1]
```

Out[22]:
```
'A'
```

It may be a little bit confusing that the first position is 0 but the last position is -1.

Using the bracket operator we can extract longer substrings as well:

In [23]:
```
seq = "ATG_TAC_TA"
      #0123456789
print(seq[4:7])
print(seq[8:len(seq)])
```

```
TAC
TA
```

These are called *slices*. Notice that the slice does not include the character at the end position, similar to the end value in the `range(start, stop[,step])` function. Similar to `range()`, slicing can include a step parameter:

In [29]:
```
seq = 'ATATATATATAT'
print(seq[0:len(seq):2])
```

```
AAAAAA
```

Let's write a function, `kmer(sequence, k)`, that prints every possible sequence of length k (k-mer) from a given DNA sequence:

In [32]:
```
def kmer(sequence, k):
    for i in range(len(sequence)-k+1):
        print(sequence[i:i+k])

kmer('ATGTCACGG', 3)
```

```
ATG
TGT
GTC
TCA
CAC
ACG
CGG
```

There are of course special values used when parameters are excluded from a slice that provide some nice shortcuts:

In [ ]:
```
seq = 'ATG_TAC_TA'
print (seq[:])   # the entire string
print (seq[4:])  # a suffix of the string
print (seq[:4])  # a prefix of the string
```

Python doesn't have a built-in function for reversing a string, however, we can do so using slices:

In [33]:
```
seq = 'ATG'
seq[::-1]
```

Out[33]:
```
'GTA'
```

String concatenation

We've already seen that you can concatenate strings together:

In [34]:
```
'TGA' + 'ATG'
```

Out[34]:
```
'TGAATG'
```

Exercise 6a

In the cell below, write a function `first_and_last()`, that concatenates the first and last nt of a sequence.

In [37]:
```
def first_and_last(seq):
    return seq[0]+seq[-1]

print(first_and_last('ATTTTTG'))
```

```
AG
```

Let's write a function, `every_other_codon()`, that concatenates every other codon in an RNA sequence:

In [1]:
```
rna = 'AUGGGGCUGAGAUUUAAAUUU'
      # 1  2  3  4  5  6  7
def every_other_codon(seq):
    codons = ''
    i = 0
    while i < len(seq):
        codons += seq[i:i+3]
        i += 6
    return codons

print(every_other_codon(rna))
```

```
AUGCUGUUUUUU
```

Stings are immutable

Existing strings cannot be changed. So while, it may be tempting to reassign a character in a string to a different value, it is not permissable:

In [2]:
```
seq = 'ATG'
seq[1] = 'U'
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-2-4674fdf3829c> in <module>()
      1 seq = 'ATG'
----> 2 seq[1] = 'U'

TypeError: 'str' object does not support item assignment
```

Of course, you can always create a new string:

In [3]:
```
seq = 'ATG'
new_seq = seq[0] + 'U' + seq[-1]
print(new_seq)
```

```
AUG
```

The `in` operator

As we saw before, the `in` operator tests if one string is a substring of another:

In [4]:
```
seq = 'ATGTGATTA'

'ATG' in seq
```

Out[4]:
```
True
```

A string is also a substring of itself:

In [5]:
```
seq in seq
```

Out[5]:
```
True
```

And of course we can test if two sequences are identical using `==` :

In [8]:
```
seq == 'ATGTGATTA'
```

Out[8]:
```
True
```

Exercise 6b

In the cell below, write a function, `coding_region()` , that returns the sequence downstream of the first start codon encountered (ATG). We've covered a lot of material now, but one approach you could use is a `for` loop to iterate through a sequence until a start codon is found in a 3 nt slice. You can return the downstream sequence using slices as well.

In [10]:
```
def coding_region(seq):
    for i in range(len(seq)):
        if 'ATG' in seq[i:i+3]:
            return seq[i+3:]

coding_region('TATGCGACAGCACATGA')
```

Out[10]:
```
'A'
```

Using the `not` operator, we can also test if a substring is not in a string:

In [11]:
```
seq = 'ATGAGAGA'
'ATG' not in seq
```

Out[11]:
```
False
```

We can use some of the comparison operators with strings, for example to test if a single character is lowercase, we can use `<=` :

In [27]:
```
char = 'a'

'a' <= 'A'
```

Out[27]:
```
False
```

Or to test the alphabetical order of some strings:

In [28]:
```
'cat' <= 'dog' <= 'zebra'
```

Out[28]:
```
True
```

WE'LL PICK UP HERE ON TUESDAY Oct. 9

Exercise 6c

Write a function, `reverse()` , that returns the reverse of a DNA sequence excluding any non-DNA characters.

In [ ]:
```

```

The `in` operator only determines if a string is a substring of another string. Let's write a function, `find()` that computes where in a string a substring occurs:

In [ ]:
```

```

Let's modify the find function with optional parameters that allow us to search within a substring of the sequence:

In [ ]:
```

```

As we saw previously, we can assign default arguments in a function.

String methods

Almost everything in Python is an object. Strings are objects. Objects often have functionality built into them in special functions called **methods**. Methods are like other functions we've seen but are associated with specific objects. For example, string objects have a `find()` method. The sytax is `str.find(str, start=0, end=len(str))`. For example:

In [ ]:

According to the syntax noted aboved, we can search a substring of the string:

In [ ]:

A return value of -1 indicates that the string was not found.

Note that `str` is the name of the *class* that represents string objects in Python.

If you would like to know what other methods a string has, you can type `str.` followed by tab for autocomplete to see all the possible options:

In [ ]:

Python also has a function, `dir(object)`, that lists methods available for that object:

In [ ]:

Alternatively, you can get all the methods associated with strings using:

In [ ]:

In case you're unsure what a given method does, you can use Python itself to get some help on it:

In [ ]:

But more likely you'll do a google search and find a list of methods like the one here (https://docs.python.org/3/library/stdtypes.html#string-methods).

Another very useful string method is `str.replace(old, new[, count)` for replacing one string with another. For example, let's replace all Ts in a DNA sequence with Us:

In [ ]:

If we want to replace only the first instance, we can use the optional count parameter:

In [ ]:

We can use the string method `str.translate` to do more complex substitutions. Let's replace A, C, G, and T with 1, 2, 3, and 4, respectively, within a string. We first have to have a translation table:

In [ ]:

String parsing

In many programming applications we need to process strings in various ways. This is called *parsing*. For example, let's extract the domain name associated with an email address such as `tai@mail.colostate.edu (mailto:`tai@mail.colostate.edu) `, using a slice and the` str.find` method:

In [ ]:

To return an uppercase string you can use `str.upper()`, where `str` is your string or a variable containing your string:

In [ ]:

Notice, however, that the variable itself is unchanged.

Alternatively, you can use the syntax `str.upper('string')`:

In [ ]:

How would you return a string in all lowercase letters?

In [ ]:

We can import the string module to facilitate string parsing:

In [ ]:
```
import string
print(string.ascii_lowercase)
print(string.ascii_uppercase)
string.whitespace
```

Let's use a `for` loop to remove all whitespace from a sequence:

In [ ]:
```

```

Debugging

Debugging is the process of making sure your program is free of errors ("bugs"). The first step in debugging your code is to know exactly what it's supposed to do. When writing your code you also need to anticipate errors that might occur if something other than what you intended is contained in the input. For example, a user inputs something wrong from the command line or a file contains unexpected header lines.

Exercise 6d

Consider for example, a fasta file which has the format:

```
# SEQUENCE IDS CORRESPOND TO FASTQ DATASET
>id_1
ATGAGATAG

>id_2
TGATGATGT
```

The following snippet of code is intended to concatenate the sequences into a one line fasta file. It works fine until it encounters an empty line or commented line.

Fix the code so that it ignores commented and empty lines. The code uses the `input()` function to input the file line by line, including the empty line.

In [ ]:
```
seq = ''
while True:
    line = input('line:  ')
    if line[0] == '>':
        continue
    elif line == 'done':
        break
    else:
        seq += line
print(">Concatenated sequence\n" + seq)
```

Additional Exercises

6e) Write a function, `underscore()`, that replaces all whitespace characters in a string with underscores and returns the modified string.

6f) Write a function, `findevery(string, substring)` that finds every occurance of a substring within a string and prints the positions, without using `str.find()`.

In [ ]: