Files

We often need to read data from files and write data to files within a Python program. The most common type of files you'll encounter in computational biology, are *text* files. Text files contain only unformatted text (e.g., no bold or italic types), typically encoded in the ASCII (https://en.wikipedia.org/wiki/ASCII) character set. However, for large datasets, *binary* files that contain space saving encodings are often used.

Reading from a file

To read or write a file in Python, we first open the file. When you open a file, you are asking the operating system to find where the file is and to give you an access point to its contents. This access point is called a *file handle* and is generated using Python's `open` command:

```
    file_handle = open('file_name')
```

Let's create a file on each of our computers with two lines containing DNA sequences. You can do this from the terminal or from right here within the notebook:

In [1]:
```
%%bash
echo ATGCATGCTAGCAT >seq.txt
echo AGCAGGCGAGCGAGATG >>seq.txt
echo AGCAGGCGAGCGAG >>seq.txt

less seq.txt
```

```
ATGCATGCTAGCAT
AGCAGGCGAGCGAGATG
AGCAGGCGAGCGAG
```

Now, let's open the file in Python:

In [2]:
```
file_handle = open('seq.txt')
file_handle
```

Out[2]:
```
<_io.TextIOWrapper name='seq.txt' mode='r' encoding='UTF-8'>
```

Notice that the file handle (file_handle) is not the actual data contained in the file, but rather an access point to the file for reading the data. The operation is successful if the requested file exists and you have the proper permissions to read the file.

The file handle has the *mode* in which it was opened. The default mode is *read* (`mode='r'`), which enables us to read the contents of a file.

Using the file handle, you can obtain the next line in the file using `readline` , a method associated with file handles. Each time you call `readline` , it returns the next line:

In [3]:
```
line = file_handle.readline()
print(line)
```

```
ATGCATGCTAGCAT
```

To access the contents of a file line by line, we can use a `for` loop:

In [4]:
```
file_handle = open('seq.txt')
for line in file_handle:
    print(line)
```

```
ATGCATGCTAGCAT

AGCAGGCGAGCGAGATG

AGCAGGCGAGCGAG
```

You may be surpised that there is an empty line between each line of sequence. Each line has `\n` at the end and the `print()` function adds a new line to the end of its output by default. We can strip off the new line characters from each line as we read them in using the `rstrip` method:

In [5]:
```
file_handle = open('seq.txt')
for line in file_handle:
    line = line.rstrip()
    print(line)
```

```
ATGCATGCTAGCAT
AGCAGGCGAGCGAGATG
AGCAGGCGAGCGAG
```

`rstrip` removes whitespace from the right side end of a string.

When the file is read using a `for` loop in this manner, Python reads in the contents of the file until it encounters a newline character (`\n` on a Mac or Linux; `\r\n` in Windows; `\n` is equivalent to the ASCII line feed (LF) character). After reading in the newline character it stops until readline is called again and then it picks up where it left off until the file is closed or it reaches the end of the file.

If the file does not exist where the program is invoked from or at the path provided, an exception is raised:

In [6]:
```python
file_handle = open('seqx.txt')
```

```
----------------------------------------------------------------------
-----
FileNotFoundError                         Traceback (most recent call
last)
<ipython-input-6-dd45e1fed2e1> in <module>()
----> 1 file_handle = open('seqx.txt')

FileNotFoundError: [Errno 2] No such file or directory: 'seqx.txt'
```

A common way to test if a file exists within a Python program, is to use `try` and `except` :

In [8]:
```python
def print_lines(file_name):
    try:
        file_handle = open(file_name)
    except:
        return -1
    for line in file_handle:
        print(line)

print_lines('seq.txt')
```

```
ATGCATGCTAGCAT

AGCAGGCGAGCGAGATG

AGCAGGCGAGCGAG
```

`try` handles all the different ways in which the operation can fail. The return value -1 tells the user that we weren't able to open the file.

A file will typically close once the file handle is no longer referenced but it's bad form to leave it open, as it continues to use resources until it's closed. To close a file, use the `close` method, `file_handle.close()` . Let's write a function that counts the number of lines in a file:

```
In [9]: def count_lines(file_handle):
            try:
                file_handle = open(file_handle)
            except:
                return -1
            count = 0
            for line in file_handle:
                count += 1
            return count
            file_handle.close()

        count_lines('seq.txt')
```

```
Out[9]: 3
```

Better yet, use a `with` statement, which will take care of the cleanup at the end, i.e. closing the file, even if an exception is raised:

```
In [11]: def count_lines(file_name):
             try:
                 file_handle = open(file_name)
             except:
                 return -1
             with file_handle:
                 count = 0
                 for line in file_handle:
                     count += 1
                 return count

         count_lines('seq.txt')
```

```
Out[11]: 3
```

```
In [ ]: with open(file_name) as file_handle: # alternative syntax
            do somthing
```

An entire file can be read at once using a file handle's `read` method:

```
In [12]: file_handle = open('seq.txt')
         file = file_handle.read()
         print(file)
```

```
ATGCATGCTAGCAT
AGCAGGCGAGCGAGATG
AGCAGGCGAGCGAG
```

Reading in an entire file at once is not typically recommended for large files as it can gobble up memory.

Exericise 7a

Write a function, `start_codon(file_name)`, that tests whether or not a start codon is found within the sequences in the `seq.txt` file and if so returns the first line it encounters with a start codon.

In [14]:
```
def start_codon(file_name):
    try:
        f = open(file_name)
    except:
        return 'File could not be opened'
    with f:
        for line in f:
            if 'ATG' in line:
                return line.rstrip()
        return -1

start_codon('seq.txt')
```

Out[14]:
```
'ATGCATGCTAGCAT'
```

Writing to a file

To write information to a file, open the file in write mode (`'w'`):

In [15]:
```
fout = open('output.txt', 'w')
fout
```

Out[15]:
```
<_io.TextIOWrapper name='output.txt' mode='w' encoding='UTF-8'>
```

This creates a new file, overwriting the contents if a file of that name already exists !!!

We can write to a file using the file handle's `write` method:

In [22]:
```
fout = open('output.txt', 'w')
fout.write('ATGACGTGACTACA\n')
fout.write('AGTGCATGCTAGTACAC\n')
fout.close()
```

```
In [23]: %%bash
         less output.txt
```

```
ATGACGTGACTACA
AGTGCATGCTAGTACAC
```

Note that we needed to explicitly put in a new-line character as `write` will not append it to the end automatically. When reading a file you can be a little sloppy about closing files, However, when writing to a file, it's important to explicitly close it.

Of course, you can also use `with` to automate closing of the file:

```
In [24]: infile = open('seq.txt')
         outfile = open('output.txt', 'w')
         with infile, outfile:
             for line in infile:
                 outfile.write(line)
```

```
In [25]: %%bash
         less output.txt
```

```
ATGCATGCTAGCAT
AGCAGGCGAGCGAGATG
AGCAGGCGAGCGAG
```

It's also common to see this syntax using `with` :

```
with open('output.txt', 'w') as outfile:
    outfile.write('ATGTAGTCGTACA\n')
    outfile.write('GGACGGGGCACAG\n')
```

You can use `with` on multiple files as well:

```
In [26]: %%bash
         echo ATACGATGTAGCTAGCTA >seq2.txt
         echo GATCGGGTCAACA >>seq2.txt
```

In [27]:
```
infile1 = open('seq.txt')
infile2 = open('seq2.txt')
outfile = open('output.txt', 'w')
with infile1, infile2, outfile:
    for line in infile1:
        outfile.write(line)
    for line in infile2:
        outfile.write(line)
```

In [28]:
```
%%bash
less output.txt
```

```
ATGCATGCTAGCAT
AGCAGGCGAGCGAGATG
AGCAGGCGAGCGAG
ATACGATGTAGCTAGCTA
GATCGGGTCAACA
```

Gotchas

Whitespace can be tricky because it's not always clear what it actually is when you print a string:

In [29]:
```
print('ATG\tCCC\nTGA T')
```

```
ATG     CCC
TGA T
```

The `repr()` function displays invisibles:

In [30]:
```
print(repr('ATG\tCCC\nTGA T'))
```

```
'ATG\tCCC\nTGA T'
```

As noted above, different operating systems use different symbols to represent new lines. In Python, the default behavior is for the `open` command to recognize *any* new line character and convert it to `\n`. Here's a wikipedia article (https://en.wikipedia.org/wiki/Newline) with more than you ever wanted to know about the newline character.

Additional Exercises

[Answer key (http://rna.colostate.edu/dokuwiki/doku.php?id=script:ex7)](http://rna.colostate.edu/dokuwiki/doku.php?id=script:ex7)

7b) Write a function called `max_from_file(file_name)` that returns the largest number stored in a file. The function should assume the numbers are floating point numbers, and that each line contains a single number.

7c) Write a function called `transcriber(input_file_name, ouput_file_name)` that reads in a fasta file - input_file_name - and converts the DNA sequences to RNA (i.e. replaces all Ts in sequence lines with Us). The function should write the results preserving fasta format to output_file_name.

7d) Write a function called `rev_comp(input_file_name, output_file_name)` that reads in a fasta file and writes each line to a new file. The function should add '_revcomp' to the end of the sequence id lines and write the reverse complements of the sequences to output_file_name. For example:

```
#input file
>seq1
ATG

#output file
>seq1_revcomp
CAT
```

7e) Write a function called `motif_finder_line_by_line(file_name, motif)` that computes the number of times a particular motif occurs in a sequence, such as a genome or chromosome sequence (a fasta file containing C. elegans chromosome 1 can be downloaded from the course website). The function should search for motifs by reading in the file one line at a time. Ignore instances of the motif that span two lines.

7f) Write a function called `motif_finder_all(file_name, motif)` that computes the number of times a particular motif occurs in a sequence, such as a genome or chromosome sequence (a fasta file containing C. elegans chromosome 1 can be downloaded from the course website). The function should read in the entire file and replace new line characters so as to find motifs that span two lines. Ignore any instances of the motif that might be found in header lines.

In [ ]:
```

```

Present Slides Themes Help