Tiffany Gassmann

A10684434

BIMM 185

March 25, 2017

Week 3 Lab Report

Introduction

The focus of this week was to continue on with the Codon usage and translatability as well as an introduction to Biopython and rsync. Our coding challenges included calculating the G/C content in the ecoli genome, making different plots of the CUI we calculated last week, and parsing a genomic file downloaded from NCBI using rsync.

Github: https://github.com/tiffanygassmann/BIMM-185-Week-3

Part 1: G/C Content E coli

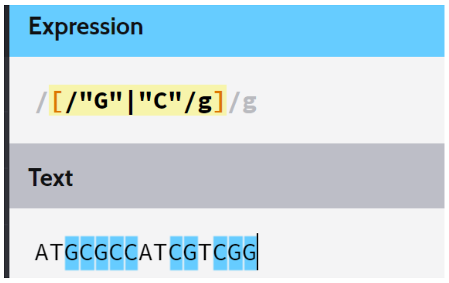
Our clicker question for this week was to calculate the G/C content of the ecoli genome we downloaded from NCBI. I wrote a simple python script which took in the sequence without the /n characters and calculated the G/C content using simple for loop and counters.

```
def GC_content(seq):

    length = len(seq)

    C = 0.0
    G = 0.0

    #check each nucleotide G/C and in sequence and increment count
    for nuc in seq:
        if nuc == "C" or nuc:
            C += 1
        if nuc == "G":
            G += 1

    #totals and content calculations
    G_C_total = C + G
    gc_content = G_C_total / length

    return float(gc_content)
```

GC Content: 0.507907098593

This can also be achieved using a regular expression:

Which searches for the "G"'s and "C"'s globally, excluding all other characters.



Expression

```
/[/"G"|"C"/g]/g
```

Text

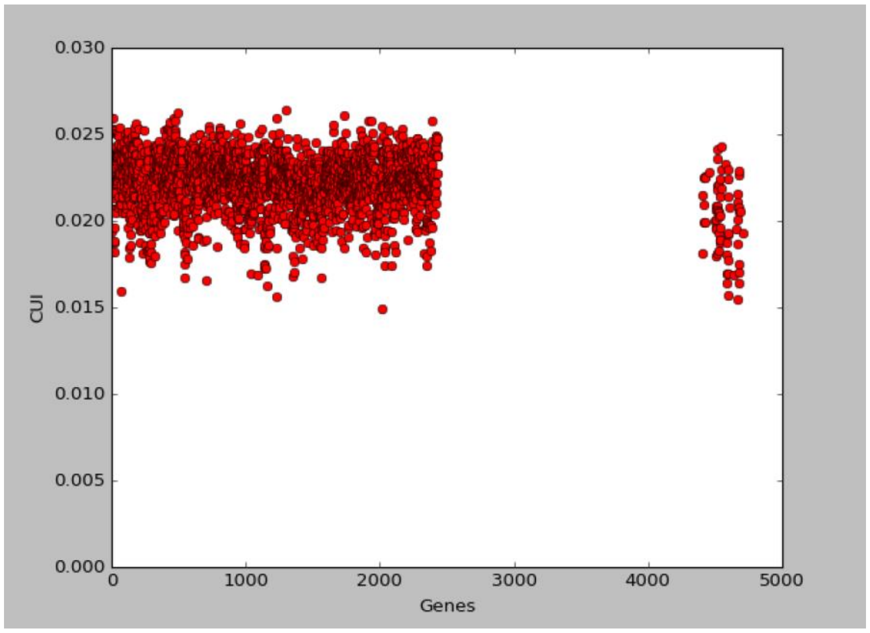
```
ATGCGCCATCGTCGG
```

Part 2: CUI and Translatability

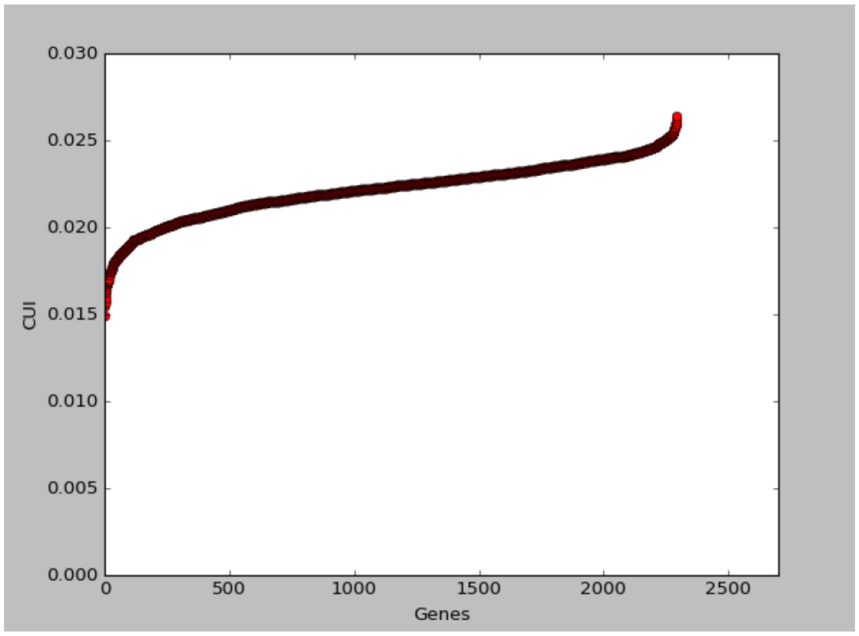
The codon usage index uses the genomic frequencies of a coon to quantify the extent to which the CU of a gene is adapted to its genome. The Hypothesis is that the more adapted a CU of a gene is to genome, the more compatible it will be with the translational machinery – tRNA – of the cell.

I then created plots to show the personality and distribution of the CUI.

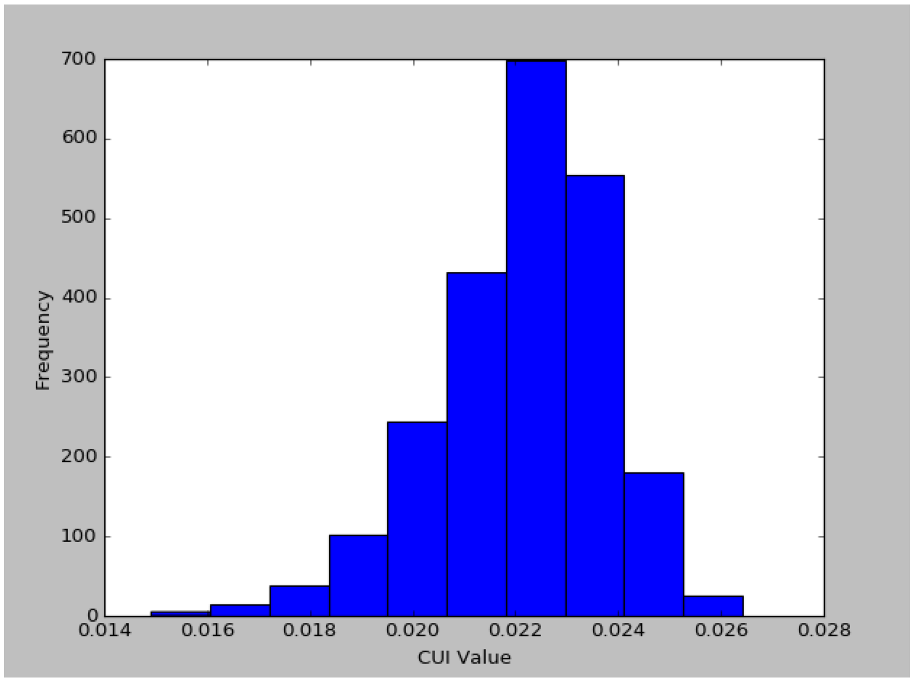
Genes in Chromosomal Order vs CUI



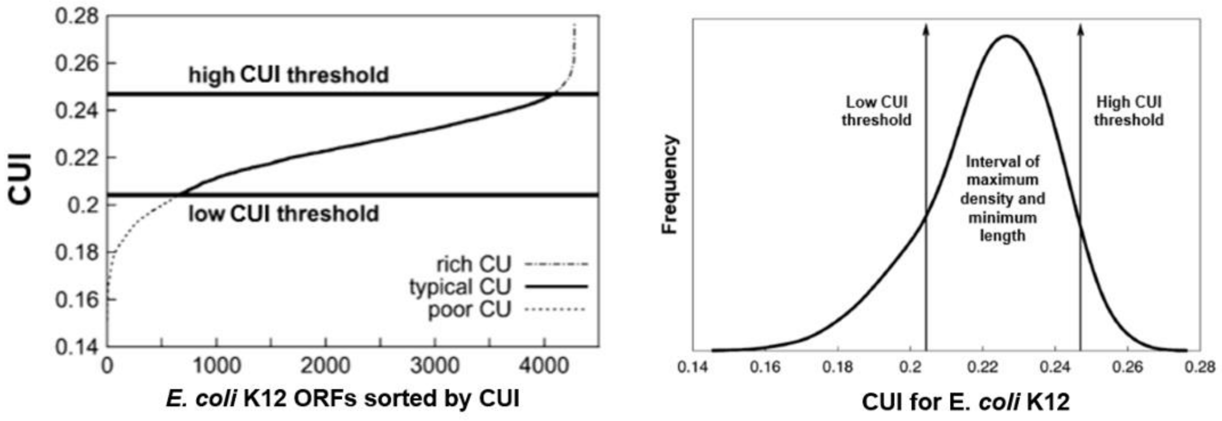
Genes in Ascending Order Based on CUI vs CUI



Histogram of CUI Values



The main topic of our first class this week surrounded the following figure:



The question asked was how to estimate the thresholds automatically and how to analyze the index to codon richness index. In the reproduced data above there is a clear area where the CUI lies upon, and it is useful for biologists to understand the threshold of the CUI in order to quantify the correctness of the CUI hypothesis.

The code for creating these plots can be found in the file: “codon_usage.py” in my Github repository.

Part 3: Rsync and Biopython

In the second portion of the week our assignment was to explore the multitude of functions included in Biopython and to understand rsync.

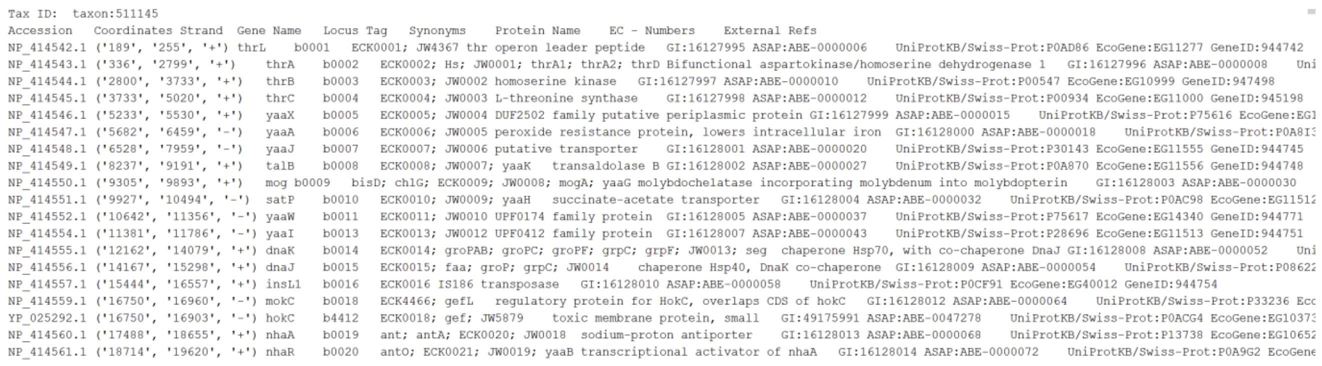
The following script was run which allowed us to download all files included in NCBI RefSeq:

```
rsync –avzL rsync://rsync.ncbi.nlm.nih.gov/genomes/all/GCF/000/005/845/GCF_000005845.2_ASM584v2/  E_coli_K12_MG1655
```

Our next assignment was to write a script which parsed the GenBank file for E. coli:

1. Tax ID (Check 'source', and /db_xref for taxon)
2. the accession (/protein_id)
3. coordinates (CDS line)
4. strand (see word 'complement' in CDS line)
5. gene name (/gene)
6. locus tag (/locus_tag)
7. synonyms (/gene_synonym)
8. protein name (/product)
9. EC-number(s) (/EC_number)
10. external references (/db_xref)

Here is a Sample of the output file:

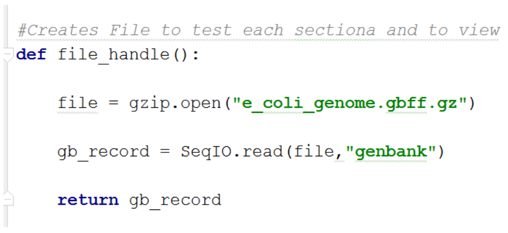


```
Tax ID:  taxon:511145
Accession   Coordinates Strand  Gene Name   Locus Tag   Synonyms    Protein Name    EC - Numbers    External Refs
NP_414542.1 ('189', '255', '+') thrL    b0001   ECK0001; JW4367 thr operon leader peptide   GI:16127995 ASAP:ABE-0000006    UniProtKB/Swiss-Prot:P0AD86 EcoGene:EG11277 GeneID:944742
NP_414543.1 ('336', '2799', '+')    thrA    b0002   ECK0002; Hs; JW0001; thrA1; thrA2; thrD Bifunctional aspartokinase/homoserine dehydrogenase 1   GI:16127996 ASAP:ABE-0000008    Uni
NP_414544.1 ('2800', '3733', '+')   thrB    b0003   ECK0003; JW0002 homoserine kinase   GI:16127997 ASAP:ABE-0000010    UniProtKB/Swiss-Prot:P00547 EcoGene:EG10999 GeneID:947498
NP_414545.1 ('3733', '5020', '+')   thrC    b0004   ECK0004; JW0003 L-threonine synthase    GI:16127998 ASAP:ABE-0000012    UniProtKB/Swiss-Prot:P00934 EcoGene:EG11000 GeneID:945198
NP_414546.1 ('5233', '5530', '+')   yaaX    b0005   ECK0005; JW0004 DUF2502 family putative periplasmic protein GI:16127999 ASAP:ABE-0000015    UniProtKB/Swiss-Prot:P75616 EcoGene:EG1
NP_414547.1 ('5682', '6459', '-')   yaaA    b0006   ECK0006; JW0005 peroxide resistance protein, lowers intracellular iron  GI:16128000 ASAP:ABE-0000018    UniProtKB/Swiss-Prot:P0A8I3
NP_414548.1 ('6528', '7959', '-')   yaaJ    b0007   ECK0007; JW0006 putative transporter    GI:16128001 ASAP:ABE-0000020    UniProtKB/Swiss-Prot:P30143 EcoGene:EG11555 GeneID:944745
NP_414549.1 ('8237', '9191', '+')   talB    b0008   ECK0008; JW0007; yaaK   transaldolase B GI:16128002 ASAP:ABE-0000027    UniProtKB/Swiss-Prot:P0A870 EcoGene:EG11556 GeneID:944748
NP_414550.1 ('9305', '9893', '+')   mog b0009   bisD; chlG; ECK0009; JW0008; mogA; yaaG molybdochelatase incorporating molybdenum into molybdopterin    GI:16128003 ASAP:ABE-0000030
NP_414551.1 ('9927', '10494', '-')  satP    b0010   ECK0010; JW0009; yaaH   succinate-acetate transporter   GI:16128004 ASAP:ABE-0000032    UniProtKB/Swiss-Prot:P0AC98 EcoGene:EG1151
NP_414552.1 ('10642', '11356', '-') yaaW    b0011   ECK0011; JW0010 UPF0174 family protein  GI:16128005 ASAP:ABE-0000037    UniProtKB/Swiss-Prot:P75617 EcoGene:EG14340 GeneID:944771
NP_414554.1 ('11381', '11786', '-') yaaI    b0013   ECK0013; JW0012 UPF0412 family protein  GI:16128007 ASAP:ABE-0000043    UniProtKB/Swiss-Prot:P28696 EcoGene:EG11513 GeneID:944751
NP_414555.1 ('12162', '14079', '+') dnaK    b0014   ECK0014; groPAB; groPC; groPF; grpC; grpF; JW0013; seg  chaperone Hsp70, with co-chaperone DnaJ GI:16128008 ASAP:ABE-0000052    Uni
NP_414556.1 ('14167', '15298', '+') dnaJ    b0015   ECK0015; faa; groP; grpC; JW0014    chaperone Hsp40, DnaK co-chaperone  GI:16128009 ASAP:ABE-0000054    UniProtKB/Swiss-Prot:P0862
NP_414557.1 ('15444', '16557', '+') insL1   b0016   ECK0016 IS186 transposase   GI:16128010 ASAP:ABE-0000058    UniProtKB/Swiss-Prot:P0CF91 EcoGene:EG40012 GeneID:944754
NP_414559.1 ('16750', '16960', '-') mokC    b0018   ECK4466; gefL   regulatory protein for HokC, overlaps CDS of hokC   GI:16128012 ASAP:ABE-0000064    UniProtKB/Swiss-Prot:P33236 Ecc
YP_025292.1 ('16750', '16903', '-') hokC    b4412   ECK0018; gef; JW5879    toxic membrane protein, small   GI:49175991 ASAP:ABE-0047278    UniProtKB/Swiss-Prot:P0ACG4 EcoGene:EG1037
NP_414560.1 ('17488', '18655', '+') nhaA    b0019   ant; antA; ECK0020; JW0018  sodium-proton antiporter    GI:16128013 ASAP:ABE-0000068    UniProtKB/Swiss-Prot:P13738 EcoGene:EG1065
NP_414561.1 ('18714', '19620', '+') nhaR    b0020   antO; ECK0021; JW0019; yaaB transcriptional activator of nhaA   GI:16128014 ASAP:ABE-0000072    UniProtKB/Swiss-Prot:P0A9G2 EcoGene
```

In order to parse the information I separated the work into two different functions; one that handled the Source information and another to handle the CDS.

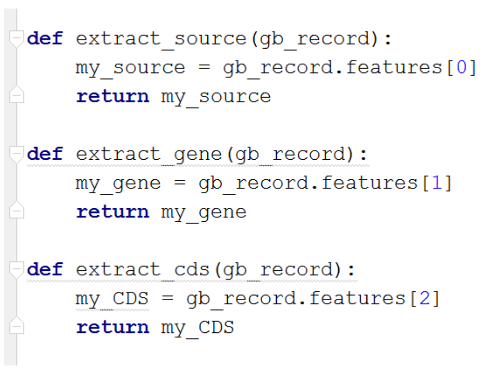
The first step was to first view the different areas of the file so I could determine which piece of info came from which section and to see the identifier names.

This first function opens the file and reads it.

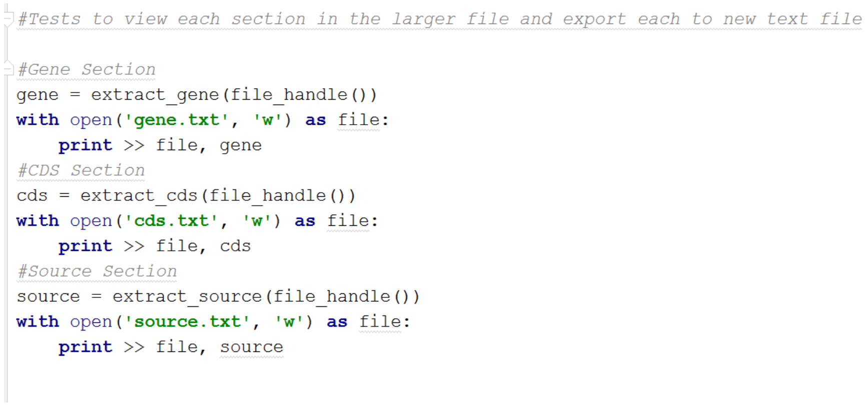


```python
#Creates File to test each sectiona and to view
def file_handle():

    file = gzip.open("e_coli_genome.gbff.gz")

    gb_record = SeqIO.read(file,"genbank")

    return gb_record
```

These next scripts output the different sections to new text files for observations.

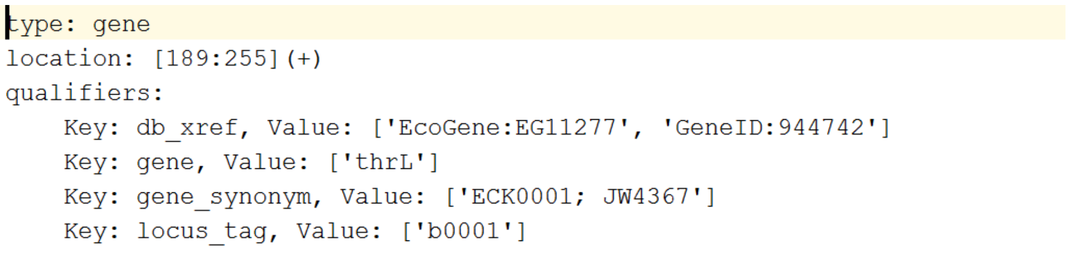


```
def extract_source(gb_record):
    my_source = gb_record.features[0]
    return my_source

def extract_gene(gb_record):
    my_gene = gb_record.features[1]
    return my_gene

def extract_cds(gb_record):
    my_CDS = gb_record.features[2]
    return my_CDS
```



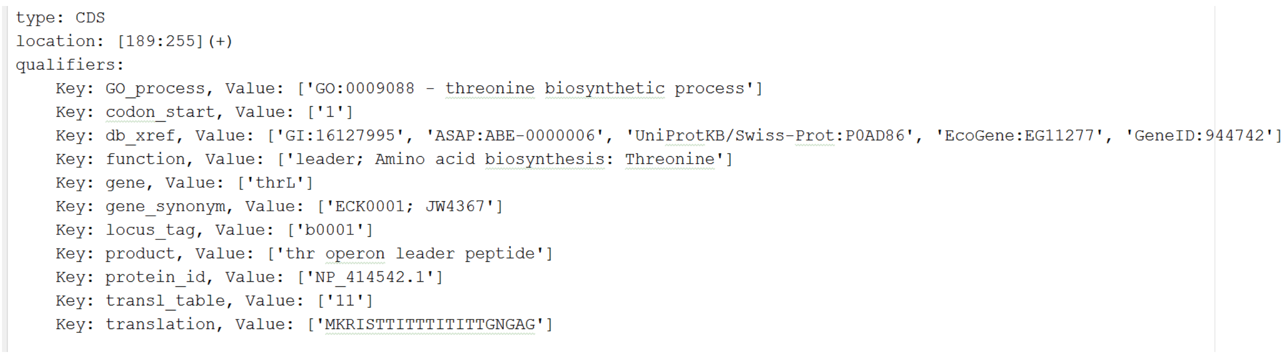
```
#Tests to view each section in the larger file and export each to new text file

#Gene Section
gene = extract_gene(file_handle())
with open('gene.txt', 'w') as file:
    print >> file, gene
#CDS Section
cds = extract_cds(file_handle())
with open('cds.txt', 'w') as file:
    print >> file, cds
#Source Section
source = extract_source(file_handle())
with open('source.txt', 'w') as file:
    print >> file, source
```

>>gene.txt



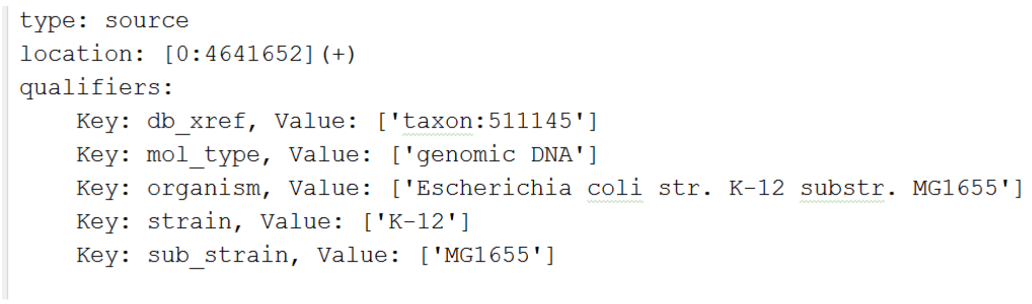
```
type: gene
location: [189:255](+)
qualifiers:
    Key: db_xref, Value: ['EcoGene:EG11277', 'GeneID:944742']
    Key: gene, Value: ['thrL']
    Key: gene_synonym, Value: ['ECK0001; JW4367']
    Key: locus_tag, Value: ['b0001']
```

>>cds.txt



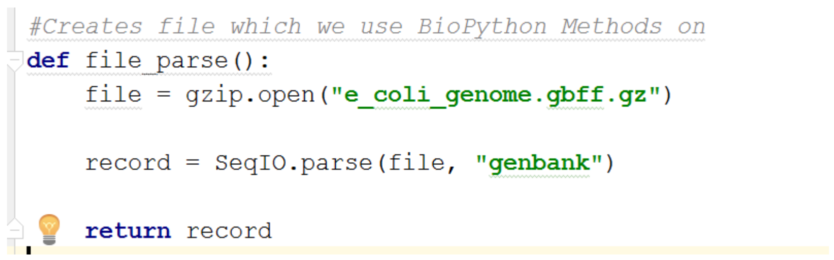
```
type: CDS
location: [189:255](+)
qualifiers:
    Key: GO_process, Value: ['GO:0009088 - threonine biosynthetic process']
    Key: codon_start, Value: ['1']
    Key: db_xref, Value: ['GI:16127995', 'ASAP:ABE-0000006', 'UniProtKB/Swiss-Prot:P0AD86', 'EcoGene:EG11277', 'GeneID:944742']
    Key: function, Value: ['leader; Amino acid biosynthesis: Threonine']
    Key: gene, Value: ['thrL']
    Key: gene_synonym, Value: ['ECK0001; JW4367']
    Key: locus_tag, Value: ['b0001']
    Key: product, Value: ['thr operon leader peptide']
    Key: protein_id, Value: ['NP_414542.1']
    Key: transl_table, Value: ['11']
    Key: translation, Value: ['MKRISTTITTTITITTGNGAG']
```

>>source.txt



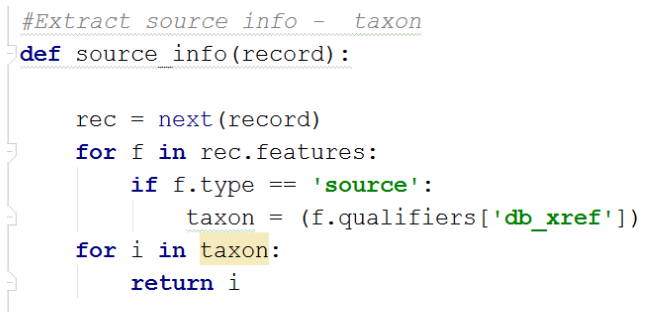
```
type: source
location: [0:4641652](+)
qualifiers:
    Key: db_xref, Value: ['taxon:511145']
    Key: mol_type, Value: ['genomic DNA']
    Key: organism, Value: ['Escherichia coli str. K-12 substr. MG1655']
    Key: strain, Value: ['K-12']
    Key: sub_strain, Value: ['MG1655']
```

The next step in this assignment was the parse the file using biopython and to scan over each section individully.



```
#Creates file which we use BioPython Methods on
def file_parse():
    file = gzip.open("e_coli_genome.gbff.gz")

    record = SeqIO.parse(file, "genbank")

    return record
```

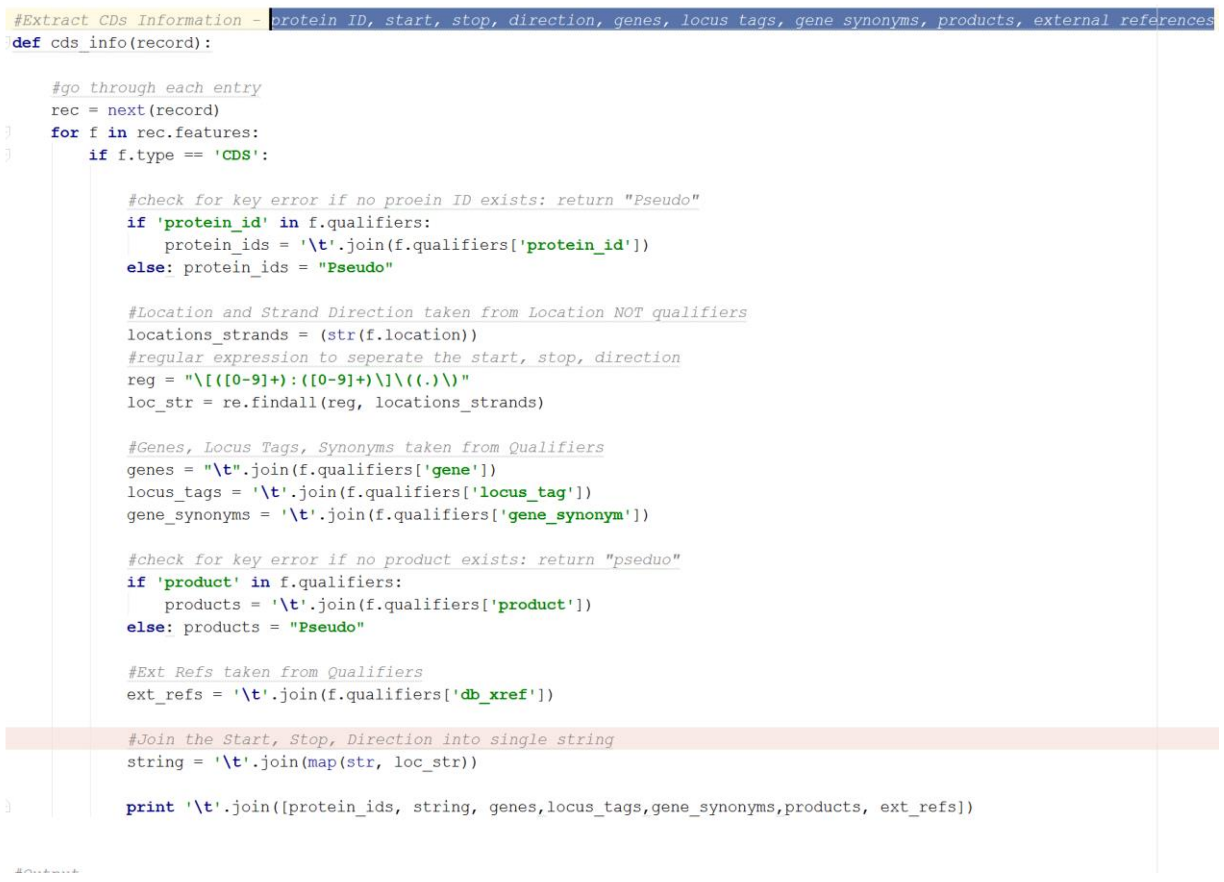
From the previous methods which allowed me to view each section (source, CDS, Gene) I was able to determine which piece of info came with each feature. The first one tackled is source:



```
#Extract source info -  taxon
def source_info(record):

    rec = next(record)
    for f in rec.features:
        if f.type == 'source':
            taxon = (f.qualifiers['db_xref'])
    for i in taxon:
        return i
```

The only piece of info we needed from here was the taxon. Each genome file only contains one source section.

The remaining pieces of info we needed to extract for each gene were all found in the CDS: *protein ID, start, stop, direction, genes, locus tags, gene synonyms, products, external references.*



```
#Extract CDs Information - protein ID, start, stop, direction, genes, locus tags, gene synonyms, products, external references
def cds_info(record):

    #go through each entry
    rec = next(record)
    for f in rec.features:
        if f.type == 'CDS':

            #check for key error if no proein ID exists: return "Pseudo"
            if 'protein_id' in f.qualifiers:
                protein_ids = '\t'.join(f.qualifiers['protein_id'])
            else: protein_ids = "Pseudo"

            #Location and Strand Direction taken from Location NOT qualifiers
            locations_strands = (str(f.location))
            #regular expression to seperate the start, stop, direction
            reg = "\[([0-9]+):([0-9]+)\]\((.)\)"
            loc_str = re.findall(reg, locations_strands)

            #Genes, Locus Tags, Synonyms taken from Qualifiers
            genes = "\t".join(f.qualifiers['gene'])
            locus_tags = '\t'.join(f.qualifiers['locus_tag'])
            gene_synonyms = '\t'.join(f.qualifiers['gene_synonym'])

            #check for key error if no product exists: return "pseduo"
            if 'product' in f.qualifiers:
                products = '\t'.join(f.qualifiers['product'])
            else: products = "Pseudo"

            #Ext Refs taken from Qualifiers
            ext_refs = '\t'.join(f.qualifiers['db_xref'])

            #Join the Start, Stop, Direction into single string
            string = '\t'.join(map(str, loc_str))

            print '\t'.join([protein_ids, string, genes,locus_tags,gene_synonyms,products, ext_refs])
```

The piece of info besides the Location/Strand direction was found in the qualifiers sections. Where the location/Strand Direction was found in the location section. This was accessed by f.location rather than f.qualifiers[accession_name].

The important BioPython methods we used are as follows:

.parse

The workhorse function Bio.SeqIO.parse() is used to read in sequence data as SeqRecord objects. The Bio.SeqIO.parse() function returns an *iterator* which gives SeqRecord objects. Iterators are typically used in a for loop as shown below.

.location

– The location of the SeqFeature on the sequence that you are dealing with.
The SeqFeature delegates much of its functionality to the location object, and includes a number of shortcut attributes for properties of the location:

.ref

.qualifiers

– This is a Python dictionary of additional information about the feature. The key is some kind of terse one-word description of what the information contained in the value is about, and the value is the actual information. For example, a common key for a qualifier might be “evidence” and the value might be “computational (non-experimental).” This is just a way to let the person who is looking at the feature know that it has not be experimentally (i. e. in a wet lab) confirmed. Note that other the value will be a list of strings (even when there is only one string). This is a reflection of the feature tables in GenBank/EMBL files.

The code and output files for this project can be found in the githun file “biopython_test.py”

Conclusion

Biopython is an important set of libraries that provide the ability to deal with a multitude of biological files – sequences, FASTA, FASTQ, GenBank….

A central object in bioinformatics is sequence analysis. Biopython Seq has two central attributes:

1. data -- as the name implies, this is the actual sequence data string of the sequence.

2. alphabet -- an object describing what the individual characters making up the string ``mean'' and how they should be interpreted.

Some interesting functions from Seq include:

```
>>> from Bio.Seq import Seq
>>> my_seq = Seq("AGTACACTGGT")
>>> my_seq
Seq('AGTACACTGGT', Alphabet())
>>> aStringSeq = str(my_seq)
>>> aStringSeq
```

```
'AGTACACTGGT'
>>> my_seq_complement = my_seq.complement()
>>> my_seq_complement
Seq('TCATGTGACCA', Alphabet())
>>> my_seq_reverse = my_seq.reverse()
>>> my_seq_rc = my_seq.reverse_complement()
>>> my_seq_rc
Seq('ACCAGTGTACT', Alphabet())
```

When dealing with FASTA files some important functions are:

```
>>> for seq_record in SeqIO.parse(os.path.join("data","ls_orchid.fasta"), "fasta"):
...     print seq_record.id
...     print repr(seq_record.seq)
...     print len(seq_record)
...
gi|2765658|emb|Z78533.1|CIZ78533
Seq('CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGG...CGC',
SingleLetterAlphabet())
740
```

Besides just dealing with certain types of files, sequences, Biopython can also do some interesting functions such as:

1. Producing Randomized Genomes

```
>>> nuc_list = list(original_rec.seq)
>>> random.shuffle(nuc_list) #acts in situ!
```

In short, the options are endless when it comes to using Biopython. The interesting thing with these libraries are that there are a multitude of ways to do the same thing, making it both useful and frustrating.