Practical Python programming by example

Converting a nucleotide sequence into an amino acid sequence

Decisions, decisions, decisions...

Topics to be covered

- Programming Models
  - Structured vs Object oriented
  - Self Contained vs Library based
- Command line arguments
- Program logic
- Make executable

The Task

Write a "simple" program to translate a DNA sequence into its protein equivalent

- Input - DNA sequence file
- Process - convert 3 letter bases to appropriate AA code (one letter or 3 letter)
- Output - Protein sequence file

The Solution

Three different programs

1) Brute force "dumb" program
2) Modular program that uses language features
3) Program built on BioPython library

What is your input

- RAW nucleotide data
  - all one line
  - multi lines
    - separated by CR (Unix/Linux)
    - separated by LF (Mac)
    - separated by LF+ CR (Windows)
- Fasta formated data (has a header line ">name description"
  - all one line
  - multi lines
    - separated by CR (Unix/Linux)
    - separated by LF (Mac)
    - separated by LF+ CR (Windows)
  - Could be multiple records in the one file

What is your Output

- File Format (Raw, Fasta, multi record)
- One or three letter codes (ARG vs R)
- Just the protein sequence or the DNA sequence on one line with the three letter code beneath it
- Do we just want the best protein (start to stop code) or a full translation
- Do we want the standard frame (starting at base 1) or an alternate frame or all three
- What about reverse compliment?
- Lets not even think about sequences (genomic) with introns/exons

Process

- DNA -> Protein or amino acids
  - but in biology DNA->RNA->protein
  - who cares - translation table is often in RNA format. So do we convert the Us in the matrix to Ts or do we convert the DNA to RNA.

RNA Codons

Second Letter

1st letter	U		C		A		G		3rd letter
U	UUU	Phe	UCU	Ser	UAU	Tyr	UGU	Cys	U
	UUC		UCC		UAC		UGC		C
	UUA	Leu	UCA		UAA	Stop	UGA	Stop	A
	UUG		UCG		UAG	Stop	UGG	Trp	G
C	CUU	Leu	CCU	Pro	CAU	His	CGU	Arg	U
	CUC		CCC		CAC		CGC		C
	CUA		CCA		CAA	Gln	CGA		A
	CUG		CCG		CAG		CGG		G
A	AUU	Ile	ACU	Thr	AAU	Asn	AGU	Ser	U
	AUC		ACC		AAC		AGC		C
	AUA		ACA		AAA	Lys	AGA	Arg	A
	AUG	Met	ACG		AAG		AGG		G
G	GUU	Val	GCU	Ala	GAU	Asp	GGU	Gly	U
	GUC		GCC		GAC		GGC		C
	GUA		GCA		GAA	Glu	GGA		A
	GUG		GCG		GAG		GGG		G

DNA Codons

1	2								3
	T		C		A		G		
T	TTT	Phe	TCT	Ser	TAT	Tyr	TGT	Cys	T
	TTC	Phe	TCC	Ser	TAC	Tyr	TGC	Cys	C
	TTA	Leu	TCA	Ser	TAA	stop	TGA	stop	A
	TTG	Leu	TCG	Ser	TAG	stop	TGG	Trp	G
C	CTT	Leu	CCT	Pro	CAT	His	CGT	Arg	T
	CTC	Leu	CCC	Pro	CAC	His	CGC	Arg	C
	CTA	Leu	CCA	Pro	CAA	Gln	CGA	Arg	A
	CTG	Leu	CCG	Pro	CAG	Gln	CGG	Arg	G
A	ATT	Ile	ACT	Thr	AAT	Asn	AGT	Ser	T
	ATC	Ile	ACC	Thr	AAC	Asn	AGC	Ser	C
	ATA	Ile	ACA	Thr	AAA	Asn	AGA	Arg	A
	ATG	Met	ACG	Thr	AAG	Lys	AGG	Arg	G
G	GTT	Val	GCT	Ala	GAT	Lys	GGT	Gly	T
	GTC	Val	GCC	Ala	GAC	Asp	GGC	Gly	C
	GTA	Val	GCA	Ala	GAA	Glu	GGA	Gly	A
	GTG	Val	GCG	Ala	GAG	Glu	GGG	Gly	G

Practically the choice is moot, UNLESS you were going to translate ALOT of sequences - then having to "transcribe" all the DNA sequences into RNA before translation would be a big waste

Versatility

- "Hard coding" file names or data makes life easy, but very limiting
- Learn to parse the command line for file names and parameters
- Streaming data in/out is also an option

Python

- Structure programming
  - or
- Object oriented programming
- Self contained
  - or
- Use a Library (BioPython)

Python

Write if all yourself

- You are in TOTAL control
- No dependencies
- Self contained
- Must do it all the work yourself and must test and validate (reinvent the wheel)
- Non-standard

Using Libraries

- Prewritten code - simpler to implement
- Standard (validate) code/function - tried and true
- Must understand exactly what the library code does and you must trust it
- May not have enough control or granularity
- Dependencies
- Need to track the dependencies
- Licensing/distirbution issues

```
#!/usr/bin/env python

debug=1;

codon=[
"ATA","ATC","ATT","ATG","ACA","ACC","ACG","ACT","AAC","AAT","AAA","AAG","AGC","AGT","AGA","AGG","CTA","CTC","CTG","CTT","CCA","C
CC","CCG","CCT","CAC","CAT","CAA","CAG","CGA","CGC","CGG","CGT","GTA","GTC","GTG","GTT","GCA","GCC","GCG","GCT","GAC","GAT","GAA
","GAG","GGA","GGC","GGG","GGT","TCA","TCC","TCG","TCT","TTC","TTT","TTA","TTG","TAC","TAT","TAA","TAG","TGC","TGT","TGA","TGG"]

aminoacid=[
"I","I","I","M","T","T","T","T","N","N","K","K","S","S","R","R","L","L","L","L","P","P","P","P","H","H","Q","Q","R","R","R","R",
"V","V","V","V","A","A","A","A","D","D","E","E","G","G","G","G","S","S","S","S","F","F","L","L","Y","Y","*","*","C","C","*","W"]

line=""
dna=""
dna_strip=""
header=""
protein=""

with open("short.fa") as read_file:
       for line in read_file:
           if line[0] in ['>']:
               header=line
           else:
               dna=dna+line
seqlength=len(dna)

if  (debug):
    print header
    print (dna)
    print seqlength

for i in range(0,seqlength,1):
    if (dna[i]!='\n' and dna[i]!='\r'):
        dna_strip=dna_strip+dna[i]
seq_length_strip=len(dna_strip)

if  (debug):
    print dna_strip
    print seq_length_strip

for i in range(0, seq_length_strip,3):
    for j in range (0,len(codon),1):
        if (dna_strip[i:i+3] == codon[j]):
           protein=protein+aminoacid[j]

print header + protein
```

dumb_trans.py

Features:

- Hardcoded values
  - Debug
  - input file name
- Manual stripping of CR\LF
- Output to terminal (not file)
- Codons in separate lists
- Double loop
- No comments or usage info

```
#!/usr/bin/env python

# Python program to convert DNA to protein
# input and output are fasta files


import argparse


# Get program arguements
def get_args():
	"""*get_args* - parses program's arg values.
	:returns: (*dict*) Contains user provided variables.
	"""

	parser = argparse.ArgumentParser()

## Required Arguements
	parser.add_argument("--input", "-i", help="Required data input fasta file. ", required=True,dest="input")
	parser.add_argument("--output", "-o", help="Required data output fasta file. ", required=True,dest="output")

	parser.add_argument("--debug", "-d", help="Optional debug flag",action='store_true')

	return parser.parse_args()



# routine to tranlate the sequence
def translate(seq):
	protein =""
#table contains codon info as a dictionary
	table = {
		'ATA':'I', 'ATC':'I', 'ATT':'I', 'ATG':'M',
		'ACA':'T', 'ACC':'T', 'ACG':'T', 'ACT':'T',
		'AAC':'N', 'AAT':'N', 'AAA':'K', 'AAG':'K',
		'AGC':'S', 'AGT':'S', 'AGA':'R', 'AGG':'R',
		'CTA':'L', 'CTC':'L', 'CTG':'L', 'CTT':'L',
		'CCA':'P', 'CCC':'P', 'CCG':'P', 'CCT':'P',
		'CAC':'H', 'CAT':'H', 'CAA':'Q', 'CAG':'Q',
		'CGA':'R', 'CGC':'R', 'CGG':'R', 'CGT':'R',
		'GTA':'V', 'GTC':'V', 'GTG':'V', 'GTT':'V',
		'GCA':'A', 'GCC':'A', 'GCG':'A', 'GCT':'A',
		'GAC':'D', 'GAT':'D', 'GAA':'E', 'GAG':'E',
		'GGA':'G', 'GGC':'G', 'GGG':'G', 'GGT':'G',
		'TCA':'S', 'TCC':'S', 'TCG':'S', 'TCT':'S',
		'TTC':'F', 'TTT':'F', 'TTA':'L', 'TTG':'L',
		'TAC':'Y', 'TAT':'Y', 'TAA':'*', 'TAG':'*',
		'TGC':'C', 'TGT':'C', 'TGA':'*', 'TGG':'W',
	}

# lookup requires that the sequence is a multiple of 3
	seqlength=len(seq)
	if (debug):
		print seqlength
	end = (int(seqlength/3))*3
# lookup the AA for each codon in the DNA sequence
	for i in range(0, end, 3):
		codon = seq[i:i + 3]
		protein+= table[codon]
	return protein

# rountine to read in the fasta file
def  read_fasta(input_file):
	dna=""
	with open(input_file) as read_file:
		for line in read_file:
			if line[0] in ['>']:
				header=line.rstrip()
			else:
				dna+= line.rstrip()
	return [header,dna]

# rountine to write out the fasta file
def write_fasta(name, sequence, output_file):

	write_file = open(output_file, 'w')
	write_file.write(name + ' translated\n')
	seq_length = len(sequence)

	for i in range(0, seq_length, 60):
		write_file.write(sequence[i:i + 60] + '\n')
	write_file.close()



############################## Start main () ##############################

## Parse arguements.
args = get_args()
infile = args.input.rstrip("")
outfile = args.output.rstrip("")
debug=args.debug
if (debug):
	print (infile + "\t" + outfile)


#get the DNA sequence
seq =  read_fasta(infile)

#translate the DNA sequence
prot = translate(seq[1])

if (debug):
	print seq[1]
	print seq[0] + "translated protein"
	print prot

#write out the protein sequence
write_fasta(seq[0],prot,outfile)
```

better_trans.py

Features:

- Command line arguments
  - Debug
  - input/output file name
- Built in usage help
- Single codon dictionary
- Output to file
- Use of rstrip to clean lines
- Use of dictionary lookup
- Comments and help

Biopyton Library

Biopython is a set of freely available tools for biological computation written in Python by an international team of developers.

It is a distributed collaborative effort to develop Python libraries and applications which address the needs of current and future work in bioinformatics.

The source code is made available under the Biopython License, which is extremely liberal and compatible with almost every license in the world.

Other Libraries of Note

NAME	Description	URL
NumPy	NumPy offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more.	https://numpy.org
SciPy	SciPy (pronounced “Sigh Pie”) is a Python-based ecosystem of open-source software for mathematics, science, and engineering. In particular, these are some of the core packages:	https://www.scipy.org
Pandas	**pandas** is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.	https://pandas.pydata.org
Jupyter	Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages.	https://jupyter.org

Code using Biopython is only 9 lines long

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

#import libraries
from Bio import SeqIO , Seq
from Bio.SeqRecord import SeqRecord

#set file names
Infile="short.fa"
Outfile="protein.fa"
#read in the file
item=SeqIO.read(Infile,"fasta")

#get and set length (should be a multiple of 3)
seqlength=len(item.seq)
end = (int(seqlength/3))*3

#print some debugging
print (item.id)
print (item.seq)
print (seqlength)
print (end)

#do the translation
protein=SeqRecord(item.seq[0:end].translate(),id=item.id, description="translated protein")

#write out the fasta file
SeqIO.write(protein,Outfile,"fasta")
```

bio_trans.py

Features:

- Command line arguments
  - Debug
  - input/output file name
- Built in usage help
- Use of std functions to read/write fast files
- Single line translation from built in codon tables
- Comments

better_trans.py vs bio_trans.py

```
def get_args():
def read_fasta(input_file):
def translate(seq):
def write_fasta(name, sequence, output_file):
```

```
def get_args():
```

```
#read in the file
item=SeqIO.read(infile,"fasta")

#do the translation
protein=SeqRecord(item.seq[0:end].translate(),id=item.id, description="
translated")

#write out the fasta file
SeqIO.write(protein,outfile,"fasta")
```

Notes

Make scripts executable

```
% which python
```

First Line of script:

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

Command:

```
chmod a+x scriptname
```

Watch out for indentation

Consistency with Tabs and Spaces

Parse arguments from command line

input/output and flags

Algorithms

A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

Using indexes to speed up processing

Find all the restriction sites in a DNA sequence

Simple brute force vs "smarter search"

In the following slides it is assumed that all restriction sites are 4bp long and that bases in the target sequence are equally distributed (25% A,G,C,T)

Brute Force (sliding window)

```
ATGGTAAGCTGCTGATGCTGCATCC
AGCT
 AGCT
  AGCT
   AGCT
    AGCT
     AGCT
      AGCT
       AGCT
        AGCT
```

Smarter Search
(key off first base)

AGCT AGCT AGCT AGCT

ATGGTAAGCTGCTGATGCTGCATCC

BRUTE FORCE

100,000

1000 4-LETTER COMPARISONS * 100 ENZYMES=100,000

INDEX ON EACH BASE (4)

1000 1-LETTER COMPARISONS=1000
4 1-LETTER COMPARISONS * 100 =400#
250 4-LETTER COMPARISONS * 100 ENZYMES=25,000

26,500

INDEX ON EACH 2-MER (16)

1000 2-LETTER COMPARISONS=1000
16 2 LETTER COMPARISONS * 100 =1600#
63 4-LETTER COMPARISONS * 100 ENZYMES=6,300

8,900

(#this could be recalculated in the enzyme file)

11X SPEEDUP

Optimization Considerations

Writing good, clean efficient code is always a good goal, but when is it worth optimizing the process

- Something that takes a *long time* Is it worth it to get a 10 fold speedup if the programs takes seconds - probably not
- Something that is run *many many times*