Introduction to Programming for BioInformatics
P. Takis Metaxas
Computer Science Department
Wellesley College
PMetaxas@wellesley.edu
Special Thanks to Joshua Kresh

Informatics for BioInformatics
  Programming skills (today)
  Python (today)
  Regular Expressions (today)
  Advance algorithms (Chapter 2)
  Download python: http://www.python.org

Where to go from here...
  If you are not familiar with programming:
  – Pasteur’s Intro to Programming using Python (long)
    http://www.pasteur.fr/formation/infobio/python/
  If you are familiar with programming but no Python:
  – Instant Python (short)
    http://www.hetland.org/python/instant-python.php
  – Beginning Python for BioInformatics (short)
  If you are familiar with Python but no regular expressions:
  – Regular Expressions howto (short)
    http://www.amk.ca/python/howto/regex/
  Else:
  – Pasteur’s Python course in BioInformatics (long)
    http://www.pasteur.fr/recherche/unites/sis/formation/python/

What is Programming
  It would be nice:
  – “HAL, can you solve this problem for me, dear?”
  Even this would be nice:
    – #include <my_knowledge.h>
    main ( ) {
      solve(this);
    }
    But instead we need:
    – An algorithm to solve “this”
      – A language to express our algorithm to a computer
Variables, Types, Operators

Variables store values of some type
- EcoRI = “GAATC”
- current_year = 2003
- GC_percentage = 0.40

Types have operators associated with them
- next_year = current_year + 1
- funny_strand = EcoRI + “AAAA” + EcoRI

Operators may look the same but behave differently depending on the type
- Operation + can be addition or concatenation...
- “Overloading”: CS types think it is a great idea

String: our MVT

Sequence of characters
- EcoRI = “GAATTC”

Index-able, but immutable
- EcoRI[0] is G; EcoRI[3] is T

Gazzillions of operations on strings
- upper( ), lower( ), join( ), split( ), replace( ), count( )….
- EcoRI.lower( ) returns “gaattc”
- EcoRI.count(‘A’) returns 2

How do you remember all the operations?
- pycrust: the flakiest python shell

pyCrust

http://www.orbtech.com/wiki/pyCrust

Write a piece of Python code to calculate the reverse complement of any given gene.
Gene is double helix strands composed of complementary nucleotides.
Complementary sequence: Nucleic acid sequence of bases that can form a double-stranded structure by matching base pairs.
E.g: complementary to GAATTC is CTTAAG and it reverse complementary is GAATTC

And now for something completely different…
The Algorithm
Reverse
Complement
replace("A", "T")
replace("T","A")
replace("G","C")
replace("C","G")
ACGGCAATTT
AAATTGCCGT
TGCCGTTAAA
But you cannot reverse a string
(“immutable”)
Enter the list!

List: Our next MVT
Ordered sequences of arbitrary python objects
Gazzilions of operations on lists
– reverse(), sort(), append(), pop(), remove(), ...
Created from a string by the list() operation
DNA= “AAATTGCCGT”
revcomp = list(DNA)
revcomp.reverse()
Glued back into a string by the join() operation
revcomp = “”.join(revcomp)

Let’s start coding!
# Four nucleotide are put into DNA
DNA= “AAATTGCCGT”
print "My original DNA: ", DNA , "\n"
#use function “reverse” to reverse the string
revcomp = list(DNA)
revcomp.reverse()
revcomp = “”.join(revcomp)
#substitute A by T, T by A, G by C and C by G
revcomp = revcomp.replace("A","T")
revcomp = revcomp.replace("T","A")
revcomp = revcomp.replace("G","C")
revcomp = revcomp.replace("C","G")
print "My reverse complementary DNA: ", revcomp, "\n"
Houston, We Have a Problem!
Complement
replace(revcomp, “A”, “T”)
replace(revcomp, “T”, “A”)
replace(revcomp, “G”, “C”)
replace(revcomp, “C”, “G”)
All the As are changed to T;
All the Ts are changed to A;
no T!
no C!
Welcome to debugging…
t = maketrans(“ATGC”, “TCGA”)
revcomp = revcomp.translate(t)
map to
A----------T
T----------A
G----------C
C----------G
There is a method known to strings…

Let’s make the correction!

from string import * # for maketrans
# Four nucleotide are put into DNA
DNA= “AAATTGCCGT”
print “My original DNA: ” , DNA , “\n”
#use function “reverse” to reverse the string
revcomp = list(DNA)
revcomp.reverse()
revcomp = “”.join(revcomp)
#substitute A by T, T by A, G by C and C by G
t = maketrans(“AGCT”, “TCGA”)
revcomp = revcomp.translate(t)
print “My reverse complementary DNA: ” , revcomp, “\n”

Function: Elegance and Generality

I have more DNA sequences, you know…
Code becomes quickly long and complicated
User-defined functions help code readability, generality
def reverseString(s):
```python

def reverseString(s):
    """Returns the reverse of a string""
    revcomp = list(s)
    revcomp.reverse()
    revcomp = "".join(revcomp)
    return revcomp

Calling a function with an argument:
reverseString('ATGC') returns 'GCAT'

s is a parameter. When calling the function s becomes 'ATGC'
revcomp is a local variable. Only visible inside reverseString()

def reverseComplement(s):
    """Returns the reverse complement of a DNA sequence""
    s = reverseString(s)
    s = complementString(s)
    return s

def complementString(s):
    """Substitutes A by T, T by A, G by C and C by G ""
    t = maketrans("AGCT","TCGA")
    s = s.translate(t)
    return s

Do a dir('string') to see that these functions are part of Python's
To use them, save them in a file 'dna2.py' and use execfile('dna2.py')

12

Decisions, Decisions, Decisions...

Often you want to decide
between two alternatives
ETs_dna = "AGGXATG"
if 'X' in ETs_dna:
    print "What the heck is X?"
else:
    print "He looks normal"

Decisions can be nested
if 'A' not in ETs_dna:
    print "He got no adenine"
elif 'G' not in ETs_dna:
    print "Got adenine but no guanine"
else:
    print "He got his As, he got his Gs..."

Normal execution flow: SEQUENTIAL

Play it again, Sam...

    counter = 5
while counter > 0:
    print "I love this class"
    counter = counter -1

for counter in range(5):  
    print "I love this class"
```
Repeating code is very powerful

```python
ETs_dna = "AGGXYZATG"
my_bases = "ACGT"
for aa in ETs_dna:
    if aa not in my_bases:
        print "What the heck is ", aa
    else:
        print "He looks normal"
```

Dictionary: Yet another MVT

Like a regular dictionary
Associates keys with values
Gazzilion of operations:
- keys(), values(), remove(), update(), clear(), ...
  Allows easy access through keys
- basecomplement = {'A': 'T', 'C': 'G', 'T': 'A', 'G': 'C'}
- basecomplement.keys() returns ['A', 'C', 'T', 'G']
- basecomplement.values() returns ['T', 'G', 'A', 'C']
- basecomplement['A'] returns 'T'

Searching for Patterns
python's strings have some searching abilities
```python
>>> execfile('my_dna_data.py')  # defines dna, EcoRI, BamHI, HindIII
>>> dna.find(EcoRI)
186
>>> dna.find(BamHI)
86
>>> dna.find(HindIII)
-1
>>> dna.index(BamHI)
86
```
But how can you tell Python
“Find me a string that starts with BamHI and ends with EcoRI”

Enter Regular Expressions
REs are a set of functions that will search for general substrings
```python
# create a mini-program to search fast for 'gaattc'
>>> import re  # you need this module
>>> p = re.compile('gaattc')
# now perform a search on dna for this pattern
>>> m = p.search(dna)
# what did you find?
>>> m.group()
'gaattc'
# where exactly did you find it?
>>> m.start()
```
Specify the substring

"Find me a string that starts with BamHI and ends with EcoRI"
(and in between can have any number of g, a, t and c symbols)

```python
>>> q=re.compile('ggatcc[\w]*gaattc')
>>> n=q.search(dna)
>>> n.start()
86
>>> n.end()
1314
```

You greedy pig. Couldn’t you find any shorter than this?

```python
>>> q=re.compile('ggatcc[\w]*?gaattc')
>>> n=q.search(dna)
>>> n.start()
86
>>> n.end()
192
```

Creating the Patterns

- `cat` matches exactly the sequence cat. Nothing else.
- `ca*t` matches 0 or more a’s: ct, cat, caat, ...
- `ca+t` matches 1 or more a’s: cat, caat, caaat, ...
- `ca?` matches 0 or 1 a’s: ct, cat. Nothing else.
- `ca{1,3}t` matches 1-3 a’s: cat, caat, caaat. Nothing else.

All of the above are greedy matches (find the longest pattern). For the shortest, add an ? as in "*, +?, ?, {1-3}?

- `c[ag]t` matches either a or g: cat, cgt.
- `c[^ag]t` matches anything but a or g: cct, ctt, c t, ...
- `\d` = [0-9] (any digit)
- `\s` =[^\n\r\f\v\t] (any whitespace character)

What else can you do?

- `p.search(dna)` = scan through the string dna for any matches of pattern p
- `p.match(dna)` = check if pattern p matches the BEGINNING of the string dna
- `p.split(dna)` = split dna into a LIST, whenever pattern p matches
- `p.sub(q,dna)` = find all substrings of dna where pattern p matches and replace them with a string q
- `p.subn(q,dna,5)` = no more than 5 replacements
Homework Assignment:
Protein Expression on Silico
1) Reading a Fasta file and reformat it
2) Transcription on Silico
3) Translation

1 ggcgcacata gcgacttggt gggcgcgtcc agtgatgact gggggatccc ggcaagtaac
   atgactaaaa agaagcggga gaatctgggc gtcgctctag agatcgatgg gttagaggag
   aagctgtccc agtgtcggag agacctggag gccgtgaact ccagactcca cagccgggag
   ctgagcccag aggccaggag gtccctggag aaggagaaaa acagcctaat gaacaaagcc
   tccaactacg agaaggaact gaagtttctt cggcaagaga accggaagaa catgctgctg
   ctgctggcca tctttatcct cctgacgctc gtctatgcct actggaccat gtgagcctgg
   cacttcccca caaccagcac aggcttccac ttggcccctt ggtcaggatc aagcaggcac
   ttcaagcctc aataggacca aggtgctggg gtgttcccct cccaacctag tgttcaagca
   tggcttcctg gcgcccagcc ttgcctccct ggcctgctgg ggggttccgg gtctccagaa
   ggacatggtg ctggtccctc ccttagccca agggagaggc aataaagaac acaaagctgt
   tcccgtaaaa aaaaaaaaaa aaaaaaaaaa aaa

Can you answer a few questions with your code:
1) Sequence length;
2) Base content;
3) Print the reverse complementary strand;
4) Think about how to translate this gene;

Code Hints:
1) Copy sequence to a file called seq.dat;
2) seq = open("seq.dat")
3) Set up your control flow
   line = seq.readline()
   while line:
      #1) substitute all the digits with null
      line = re.sub("\d","",line)
      #2) substitute white space with null
      line = re.sub("\w","",line)
      #3) Reverse Complement your DNA
      line = revcomp(line)
      DNA = DNA+line
      line = seq.readline()
#Base content
print "Adenine: ", newDNA.count("a")
print "Thymine: ", newDNA.count("t")
print "Guanine: ", newDNA.count("g")
print "Cytosine: ", newDNA.count("c")

#Translation
standard = { 'ttt': 'F', 'tct': 'S', 'tat': 'Y', 'tgt': 'C', 'ttc': 'F', 'tcc': 'S', 'tac': 'Y', 'tgc': 'C',
            'tta': 'L', 'tca': 'S', 'taa': '*', 'tca': '*', 'ltg': 'L', 'tgg': 'S', 'tag': '*', 'tcg': 'W', 'ctf': 'L',
            'caq': 'Q', 'cqa': 'R', 'cql': 'L', 'ccq': 'P', 'cag': 'Q', 'cgg': 'R', 'att': 'I', 'act': 'T', 'aat': 'N',
            'agt': 'S', 'atc': 'I', 'acc': 'T', 'aac': 'N', 'agc': 'S', 'ata': 'I', 'aca': 'T', 'aaa': 'K', 'aga': 'R',
            'atg': 'M', 'aac': 'I', 'aag': 'K', 'agg': 'R', 'gtt': 'V', 'gtc': 'A', 'gat': 'D', 'gtg': 'G',
            'gct': 'A', 'gac': 'D', 'gcc': 'A', 'gag': 'E', 'ggc': 'A', 'gaa': 'E', 'gga': 'G', 'ggg': 'V',
            'gag': 'A', 'gag': 'E', 'ggg': 'G' }  #a dictionary
#function definition
def dnatoprotein (dna, code):
   *** translate a DNA sequence to a protein ***
prot = ""
for i in xrange(0,len(dna),3):
    prot += code.get(dna[i:i+3], "?")
return prot