Essential Computing for Bioinformatics

Lecture 5

High-level Programming with Python

Container Objects

MARC: Developing Bioinformatics Programs
July 2009

Alex Ropelewski
PSC-NRBSC

Bienvenido Vélez
UPR Mayaguez
The following material is the result of a curriculum development effort to provide a set of courses to support bioinformatics efforts involving students from the biological sciences, computer science, and mathematics departments. They have been developed as a part of the NIH funded project “Assisting Bioinformatics Efforts at Minority Schools” (2T36 GM008789). The people involved with the curriculum development effort include:

- Dr. Hugh B. Nicholas, Dr. Troy Wymore, Mr. Alexander Ropelewski and Dr. David Deerfield II, National Resource for Biomedical Supercomputing, Pittsburgh Supercomputing Center, Carnegie Mellon University.
- Dr. Ricardo González Méndez, University of Puerto Rico Medical Sciences Campus.
- Dr. Alade Tokuta, North Carolina Central University.
- Dr. Jaime Seguel and Dr. Bienvenido Vélez, University of Puerto Rico at Mayagüez.
- Dr. Satish Bhalla, Johnson C. Smith University.

Unless otherwise specified, all the information contained within is Copyrighted © by Carnegie Mellon University. Permission is granted for use, modify, and reproduce these materials for teaching purposes.

Most recent versions of these presentations can be found at [http://marc.psc.edu/](http://marc.psc.edu/)
Outline

• Lecture 2 Homework
• Lists and Other Sequences
• Dictionaries and Sequence Translation
• Finding ORF’s in sequences

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center.
from string import *
def searchPattern(dna, pattern):
    'print all start positions of a pattern string inside a target string'
    site = findDNAPattern (dna, pattern)
    while site != -1:
        print 'pattern %s found at position %d' % (pattern, site)
        site = findDNApattern (dna, pattern, site + 1)

>>> searchPattern("acgctaggct","gc")
pattern gc at position 2
pattern gc at position 7

What if DNA may contain unknown nucleotides ‘X’?

Example from Pasteur Institute Bioinformatics Using Python

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center
Write your own find function:

```python
def findDNAPattern(dna, pattern, startPosition, endPosition):
    'Finds the index of the first occurrence of DNA pattern within DNA sequence'
    dna = dna.lower()  # Force sequence and pattern to lower case
    pattern = pattern.lower()
    for i in xrange(startPosition, endPosition):
        # Attempt to match pattern starting at position i
        if matchDNAPattern(dna[i:], pattern):
            return i
    return -1
```

**Top-Down Design:** From **BIG** functions to **small** helper functions
Write your own find function:

def matchDNAPattern(sequence, pattern):
    'Determines if DNA pattern is a prefix of DNA sequence'
    i = 0
    while ((i < len(pattern)) and (i < len(sequence))):
        if (not matchDNANucleotides(sequence[i], pattern[i])):
            return False
        i = i + 1
    return (i == len(pattern))

Top-Down Design: From BIG functions to small helper functions
Write your own find function:

```python
def matchDNANucleotides(base1, base2):
    'Returns True if nucleotide bases are equal or one of them is unknown'
    return (base1 == 'x' or
            base2 == 'x' or
            (isDNANucleotide(base1) and (base1 == base2)))
```

Top-Down Design: From BIG functions to small helper functions
Using default parameters:

```python
def findDNAPattern(dna, pattern, startPosition=0, endPosition=None):
    'Finds the index of the first occurrence of DNA pattern within DNA sequence'
    if (endPosition == None):
        endPosition = len(dna)
    dna = dna.lower() # Force sequence and pattern to lower case
    pattern = pattern.lower()
    for i in xrange(startPosition, endPosition):
        # Attempt to match pattern starting at position i
        if (matchDNAPattern(dna[i:], pattern)):
            return i
    return -1
```
Top Down Design: A Recursive Process

- Start with a high level problem
- Design a high-level function assuming existence of ideal lower level functions that it needs
- Recursively design each lower level function top-down
List Values

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center

Homogeneous Lists

Lists can be heterogeneous and nested

[10, 20, 30, 40]
['spam', 'bungee', 'swallow']
['hello', 2.0, 5, [10, 20]]
[]

The empty list
Generating Integer Lists

>>> range(1,5)
[1, 2, 3, 4]

>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> range(1, 10, 2)
[1, 3, 5, 7, 9]

In General

range(first,last+1,step)
Accessing List Elements

```python
>> words=['hello', 'my', 'friend']

>> words[1]
'my'

>> words[1:3]
['my', 'friend']

>> words[-1]
'friend'

>> 'friend' in words
True

>> words[0] = 'goodbye'

>> print words
['goodbye', 'my', 'friend']
```

- Accessing single element
- Accessing slices
- Accessing negative index
- Testing list membership
- Lists are mutable
More List Slices

Slicing operator always returns a NEW list

```python
>> numbers = range(1,5)
[1, 2, 3, 4]

>> numbers[1:]
[1, 2, 3, 4]

>> numbers[:3]
[1, 2]

>> numbers[:]
[1, 2, 3, 4]
```
Modifying Slices of Lists

```python
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3] = ['x', 'y']
>>> print list
['a', 'x', 'y', 'd', 'e', 'f']
>>> list[1:3] = []
>>> print list
['a', 'd', 'e', 'f']
>>> list = ['a', 'd', 'f']
>>> list[1:1] = ['b', 'c']
>>> print list
['a', 'b', 'c', 'd', 'f']
>>> list[4:4] = ['e']
>>> print list
['a', 'b', 'c', 'd', 'e', 'f']
```
Traversing Lists ( 2 WAYS)

codons = ['cac', 'caa', 'ggg']

for codon in codons:
    print codon

i = 0
while (i < len(codons)):
    codon = codons[i]
    print codon
    i = i + 1

Which one do you prefer? Why?

Why does Python provide both for and while?
def stringToList(theString):
    'Returns the input string as a list of characters'
    result = []
    for element in theString:
        result = result + [element]
    return result

def listToString(theList):
    'Returns the input list of characters as a string'
    result = ""
    for element in theList:
        result = result + element
    return result
Complementing Sequences: Utilities

DNANucleotides='acgt'
DNAComplements='tgca'

def isDNANucleotide(nucleotide):
    'Returns True when n is a DNA nucleotide'
    return (type(nucleotide) == type('') and
            len(nucleotide)==1 and
            nucleotide.lower() in DNANucleotides)

def isDNASequence(sequence):
    'Returns True when sequence is a DNA sequence'
    if type(sequence) != type(''):
        return False;
    for base in sequence:
        if (not isDNANucleotide(base.lower())):
            return False
    return True
def getComplementDNANucleotide(n):
    'Returns the DNA Nucleotide complement of n'
    if (isDNANucleotide(n)):
        return (DNAComplements[find(DNANucleotides, n.lower())])
    else:
        raise Exception("getComplementDNANucleotide: Invalid DNA sequence: " + n)

def getComplementDNASequence(sequence):
    'Returns the complementary DNA sequence'
    if (not isDNASequence(sequence)):
        raise Exception("getComplementDNASequence: Invalid DNA sequence: " + sequence)
    result = ""
    for base in sequence:
        result = result + getComplementDNANucleotide(base)
    return result
def getComplementDNASequences(sequences):
    'Returns a list of the complements of list of DNA sequences'
    result = []
    for sequence in sequences:
        result = result + [getComplementDNASequence(sequence)]
    return result

>>> getComplementDNASequences(["acg", "ggg"])
['tgc', 'ccc']

>>>
## Python Sequence Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Elements</th>
<th>Mutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>StringType</td>
<td>Character string</td>
<td>Characters only</td>
<td>no</td>
</tr>
<tr>
<td>UnicodeType</td>
<td>Unicode character string</td>
<td>Unicode characters only</td>
<td>no</td>
</tr>
<tr>
<td>ListType</td>
<td>List</td>
<td>Arbitrary objects</td>
<td>yes</td>
</tr>
<tr>
<td>TupleType</td>
<td>Immutable List</td>
<td>Arbitrary objects</td>
<td>no</td>
</tr>
<tr>
<td>XRangeType</td>
<td>return by <code>xrange()</code></td>
<td>Integers</td>
<td>no</td>
</tr>
<tr>
<td>BufferType</td>
<td>return by <code>buffer()</code></td>
<td>arbitrary objects of one type</td>
<td>yes/no</td>
</tr>
</tbody>
</table>

These materials were developed with funding from the US National Institutes of Health grant #2T36 GM008789 to the Pittsburgh Supercomputing Center.
## Operations on Sequences

<table>
<thead>
<tr>
<th>Operator/Function</th>
<th>Action</th>
<th>Action on Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ], ( ), ''</td>
<td>creation</td>
<td></td>
</tr>
<tr>
<td>s + t</td>
<td>concatenation</td>
<td>addition</td>
</tr>
<tr>
<td>s * n</td>
<td>repetition n times</td>
<td>multiplication</td>
</tr>
<tr>
<td>s[i]</td>
<td>indexation</td>
<td></td>
</tr>
<tr>
<td>s[i:k]</td>
<td>slice</td>
<td></td>
</tr>
<tr>
<td>x in s</td>
<td>membership</td>
<td></td>
</tr>
<tr>
<td>x not in s</td>
<td>absence</td>
<td></td>
</tr>
<tr>
<td>for a in s</td>
<td>traversal</td>
<td></td>
</tr>
<tr>
<td>len(s)</td>
<td>length</td>
<td></td>
</tr>
<tr>
<td>min(s)</td>
<td>return smallest element</td>
<td></td>
</tr>
<tr>
<td>max(s)</td>
<td>return greatest element</td>
<td></td>
</tr>
</tbody>
</table>
Design and implement Python functions to satisfy the following contracts:

• Return the list of codons in a DNA sequence for a given reading frame
• Return the lists of restriction sites for an enzyme in a DNA sequence
• Return the list of restriction sites for a list of enzymes in a DNA sequence
• Find all the ORF’s of length $\geq n$ in a sequence
Dictionaries are *mutable unordered collections* which may contain objects of different sorts. The objects can be accessed using a *key*.
Genetic Code As Python Dictionary

GeneticCode =
{
    'ttt': 'F', 'tct': 'S', 'tat': 'Y', 'tgt': 'C',
    'ttc': 'F', 'tcc': 'S', 'tac': 'Y', 'tgc': 'C',
    'ttg': 'L', 'tcg': 'S', 'tag': '*', 'tgg': 'W',
    'ctt': 'L', 'cct': 'P', 'cat': 'H', 'cgc': 'R',
    'ctc': 'L', 'ccc': 'P', 'cac': 'H', 'cgc': 'R',
    'cta': 'L', 'cca': 'P', 'caa': 'Q', 'cga': 'R',
    'ctg': 'L', 'ccg': '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', 'acg': 'T', 'aag': 'K', 'agg': 'R',
    'gtt': 'V', 'gct': 'A', 'gat': 'D', 'ggt': 'G',
    'gtc': 'V', 'gcc': 'A', 'gac': 'D', 'ggc': 'G',
    'gta': 'V', 'gca': 'A', 'gaa': 'E', 'gga': 'G',
    'gtg': 'V', 'gcg': 'A', 'gag': 'E', 'ggg': 'G'
}
cds ='''atgagtgaacgtctgacattacccgctggggcgtatatcgggcCACAAA
tttcgggtgccgacctgacgcgcggccgtaaagcgataatcagttttgaacagctttaccatgcgggtg
tcgcgccatcaggtggggtttctacgcqatcaagctattacgcgcagcgagcaacgcgcgctggc
cagcgtttttggcgaattgcataattcaccctgttttacccgcatgccgaagggttgacgagatca
tcggtgctggataaccawgataaacacgacagttggcagctaaagagttacccggtgacatattt
tagaaacgcaccacgcagggcgaattcgtgcagcttaaagagttaccctcgcaccgaggtgatac
gctctggaccagcgcgtatctgagcgtctctgtttttttcgcacttgctgctgatgtgaggtgagtgggctgcgcagcatagcatrgctttccgggaataacctgcaaaaccggagag
gacatcaacgcgcggcgcagggcgtttcgtacatccggttggtgcgaacgcgtccggtgaaccagggcgtgtaaacagcgcgtttttttgtgaatgaaggctttactacgcgaattgttgatgtgagtgcgagagccgagcatagcacctgcggctgctgcaaccaaatgatattgcgatttgggataaccgcgtgacccagcactatgccaatgccgattacctgccacagcgacggataatgcatcgggcgacgatcctggggataaac
cgttttatcgggcggggtaa'''.replace('
',''').lower()
def translateDNASequence(dna):
    if (not isDNASequence(dna)):
        raise Exception('translateDNASequence: Invalid DNA sequence')
    prot = ""
    for i in range(0, len(dna), 3):
        codon = dna[i:i+3]
        prot = prot + GeneticCode[codon]
    return prot

>>> translateDNASequence(cds)
'MSERLSITPLGYPYIGAQISGADLTRPLSDNJQFEQLYHAVLHRHQVVVFRLDQAITPQQQRalaQRFGHELHIHPVYPHAEGVDEIIIVLDTNDNDNPDNDWHTDVTFIETPPAGAIALAKELPSTGGDTLWTSGIAAYEALSVPFRQLLSGLRAEHDFRKSFPFEYKRYKTEEEHKQRWREAIAKNPLLHPVVRTHPVSGKQALFVNEGFTTRIVDVSEKESEALLESLFIFAHITKPEFQVRWRWOQPNDIAIWDNRVTQHYANADYLQPQRIMHRATILGDKPFYRAG*'
<table>
<thead>
<tr>
<th>Method or Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>d[key]</td>
<td>Get the value of the entry with key key in d</td>
</tr>
<tr>
<td>d[key] = val</td>
<td>Set the value of entry with key key to val</td>
</tr>
<tr>
<td>del d[key]</td>
<td>Delete entry with key key</td>
</tr>
<tr>
<td>d.clear()</td>
<td>Removes all entries</td>
</tr>
<tr>
<td>len(d)</td>
<td>Number of items</td>
</tr>
<tr>
<td>d.copy()</td>
<td>Makes a shallow copya</td>
</tr>
<tr>
<td>d.has_key(key)</td>
<td>Returns 1 if key exists, 0 otherwise</td>
</tr>
<tr>
<td>d.keys()</td>
<td>Gives a list of all keys</td>
</tr>
<tr>
<td>d.values()</td>
<td>Gives a list of all values</td>
</tr>
<tr>
<td>d.items()</td>
<td>Returns a list of all items as tuples (key, value)</td>
</tr>
<tr>
<td>d.update(new)</td>
<td>Adds all entries of dictionary new to d</td>
</tr>
<tr>
<td>d.get(key[, otherwise])</td>
<td>Returns value of the entry with key key if it exists</td>
</tr>
<tr>
<td></td>
<td>Otherwise returns to otherwise</td>
</tr>
<tr>
<td>d.setdefaults(key [, val])</td>
<td>Same as d.get(key), but if key does not exist, sets d[key] to val</td>
</tr>
<tr>
<td>d.popitem()</td>
<td>Removes a random item and returns it as tuple</td>
</tr>
</tbody>
</table>
def findDNAORFPos(sequence, minLen, startCodon, stopCodon, startPos, endPos):
    'Finds the position and length of the first ORF in sequence'
    while (startPos < endPos):
        startCodonPos = find(sequence, startCodon, startPos, endPos)
        if (startCodonPos >= 0):
            stopCodonPos = find(sequence, stopCodon, startCodonPos, endPos)
            if (stopCodonPos >= 0):
                if ((stopCodonPos - startCodonPos) > minLen):
                    return [startCodonPos + 3, (stopCodonPos - startCodonPos) - 3]
                else:
                    startPos = startPos + 3
            else:
                return [-1, 0] # Finished the sequence without finding stop codon
        else:
            return [-1, 0] # Could not find any more start codons
def extractDNAORF(sequence, minLen, startCodon, stopCodon, startPos, endPos):
    'Returns the first ORF of length >= minLen found in sequence'
    ORFPos = findDNAORFPos(sequence, minLen, startCodon, stopCodon, startPos, endPos)
    startPosORF = ORFPos[0]
    endPosORF = startPosORF + ORFPos[1]
    if (startPosORF >= 0):
        return sequence[ORFPos[0]: ORFPos[0]+ORFPos[1]]
    else:
        return ""
• Design an ORF extractor to return the list of all ORF’s within a sequence together with their positions
• Handling files containing sequences and alignments