Introduction

Arafat_Lecture_03,June, 2017

- Matplotlib is the most frequently used plotting package.
- Written in pure Python
- Heavily dependent on NumPy.

Matplotlib Philosophy:

- Plots should look great - publication quality. One important requirement for me is that the text looks good (antialiased, etc.)
- Postscript output for inclusion with TeX documents
- Embeddable in a graphical user interface for application development
- Code should be easy enough that I can understand it and extend it
- Making plots should be easy

`Pyplot` Tutorial

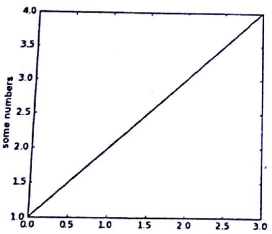
`matplotlib.pyplot` is a collection of command style functions that make matplotlib work like MATLAB.

Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

In `matplotlib.pyplot` various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current axes (please note that "axes" here and in most places in the documentation refers to the axes part of a figure and not the strict mathematical term for more than one axis).

Line Plot

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```

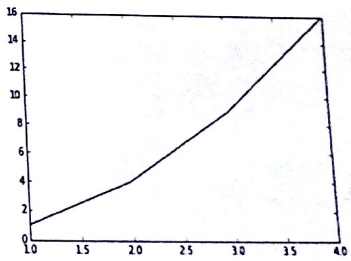


You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to the plot() command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are [0,1,2,3].

Line Plot (cont.)

`plot()` is a versatile command, and will take an arbitrary number of arguments. For example, to plot x versus y, you can issue the command:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

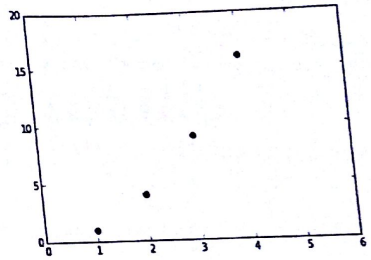


More on Plot

For every `x, y` pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string. The default format string is 'b-', which is a solid blue line. For example, to plot the above with red circles, you would issue:

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```


More on Plot (cont.)



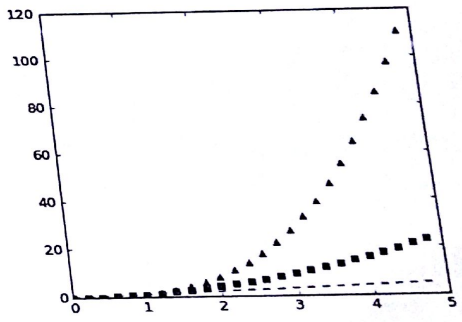
See the `plot()` documentation for a complete list of line styles and format strings. The axis() command in the example above takes a list of [xmin, xmax, ymin, ymax] and specifies the viewport of the axes.

Plots Using NumPy

If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use numpy arrays. In fact, all sequences are converted to numpy arrays internally. The example below illustrates a plotting several lines with different format styles in one command using arrays.

```
import numpy as np
import matplotlib.pyplot as plt

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```


Plots Using NumPy (Cont.)



Working with Multiple Figures and Axes

- MATLAB, and `pyplot`, have the concept of the current figure and the current axes. All plotting commands apply to the current axes.
- The function gca() returns the current axes (a matplotlib.axes.Axes instance), and gcf() returns the current figure (matplotlib.figure.Figure instance).

- Normally, you don't have to worry about this, because it is all taken care of behind the scenes.

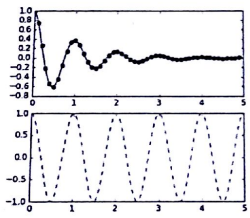
- In next slide a script to create two subplots is shown.

Working with Multiple Figures and Axes [Cont.]

```
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```

Working with Multiple Figures and Axes [Cont.]

- The `figure()` command here is optional because `figure(1)` will be created by default, just as a `subplot(111)` will be created by default if you don't manually specify any axes.
- The `subplot()` command specifies numrows, numcols, fignum where fignum ranges from 1 to numrows*numcols. The commas in the subplot command are optional if numrows*numcols<10. So `subplot(211)` is identical to `subplot(2, 1, 1)`.
- You can create an arbitrary number of subplots and axes. If you want to place an axes manually, i.e., not on a rectangular grid, use the `axes()` command, which allows you to specify the location as axes([left, bottom, width, height]) where all values are in fractional (0 to 1) coordinates.

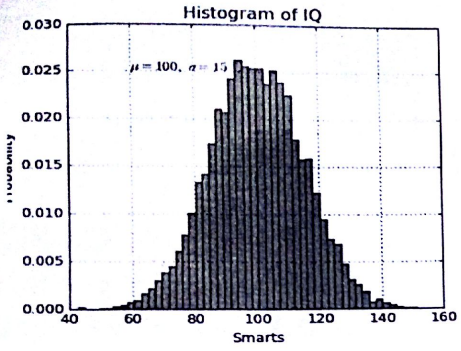


Working with text

The `text()` command can be used to add text in an arbitrary location, and the `xlabel()`, `ylabel()` and title() are used to add text in the indicated locations.

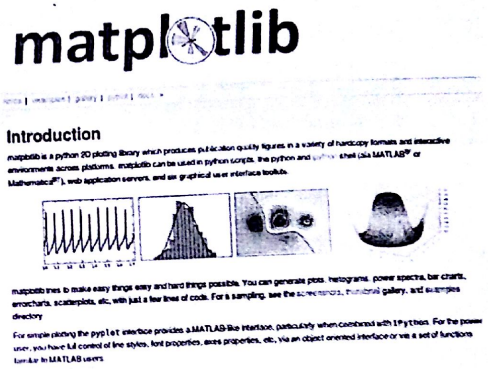
```
import numpy as np
import matplotlib.pyplot as plt
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='g'
  alpha=0.75)
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```

Working with text (cont.)



Visit Documentation for More ...

- Go to `http://matplotlib.org/index.html` for more



Mini Project I

DNA Base-Stacking Energy & Propeller Twist Calculation

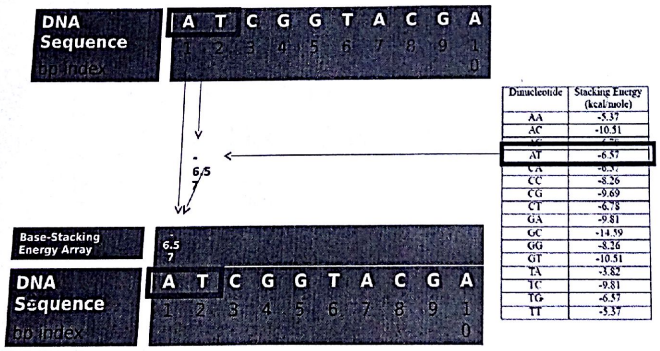
A DNA Sequence with index

Base Stacking Energy of all possible di-nucleotides

Dinucleotide	Stacking Energy (kcal/mole)
AA	-5.37
AC	-10.51
AG	-6.78
AT	-6.57
CA	-6.57
CC	-8.26
CG	-9.69
CT	-6.78
GA	-9.81
GC	-14.59
GG	-8.26
GT	-10.51
TA	-3.82
TC	-9.81
TG	-6.57
TT	-5.37

DNA Base-Stacking Energy Array

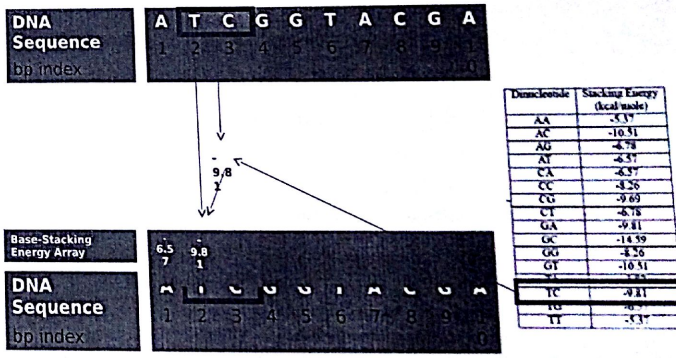
(How to Calculate)



Dinucleotide	Stacking Energy (kcal/mole)
AA	-5.37
AC	-10.51
AG	-6.78
AT	-6.57
CA	-6.57
CC	-8.26
CG	-9.69
CT	-6.78
GA	-9.81
GC	-14.59
GG	-8.26
GT	-10.51
TA	-3.82
TC	-9.81
TG	-6.57
TT	-5.37

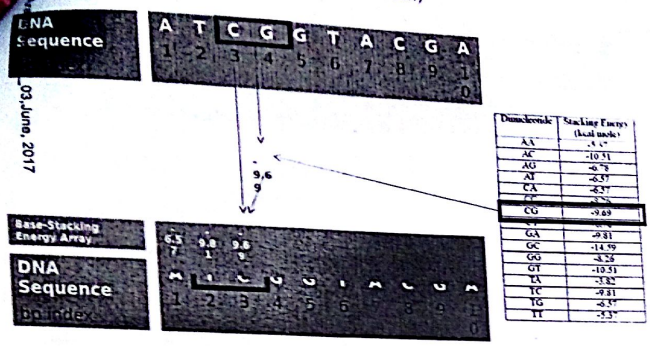
DNA Base-Stacking Energy Array

(How to Calculate...)



Dinucleotide	Stacking Energy (kcal/mole)
AA	-5.37
AC	-10.51
AG	-6.78
AT	-6.57
CA	-6.57
CC	-8.26
CG	-9.69
CT	-6.78
GA	-9.81
GC	-14.59
GG	-8.26
GT	-10.51
TA	-3.82
TC	-9.81
TG	-6.57
TT	-5.37

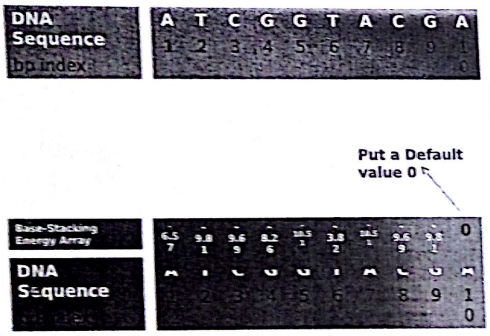
NA Base-Stacking Energy Array
(How to Calculate...)



Dinucleotide	Stacking Energy (kcal/mole)
AA	-8.17
AC	-10.51
AG	-9.78
AT	-8.37
CA	-8.57
CC	-8.26
CG	-9.69
GA	-9.81
GC	-14.59
GG	-8.26
GT	-10.51
TA	-3.82
TC	-9.81
TG	-8.57
TT	-8.17

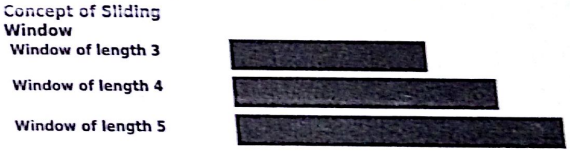
In this way.....

DNA Base-Stacking Energy Array
(Completed)



Dinucleotide	Stacking Energy (kcal/mole)
AA	-8.17
AC	-10.51
AG	-9.78
AT	-8.37
CA	-8.57
CC	-8.26
CG	-9.69
GA	-9.81
GC	-14.59
GG	-8.26
GT	-10.51
TA	-3.82
TC	-9.81
TG	-8.57
TT	-8.17

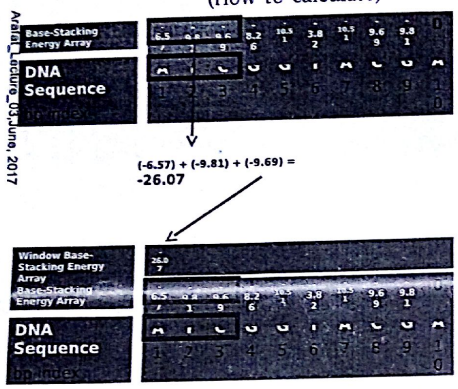
Window-Based DNA Base-Stacking Energy Array
(What is Sliding Window)



Lets work with sliding window of length 3

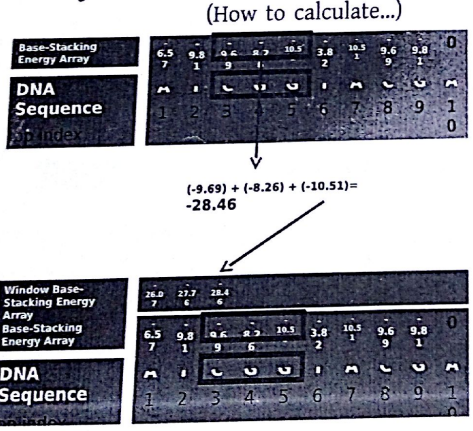
Window-Based DNA Base-Stacking Energy Array

(How to calculate)



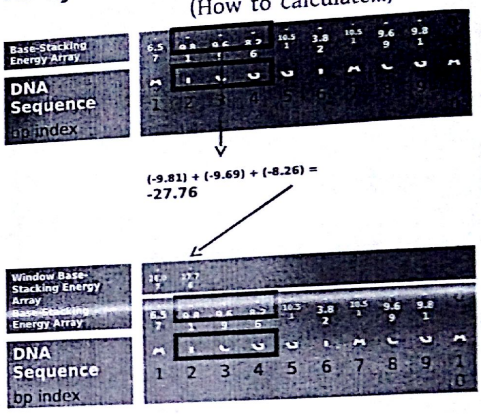
Window-Based DNA Base-Stacking Energy Array

(How to calculate...)



Window-Based DNA Base-Stacking Ener Array

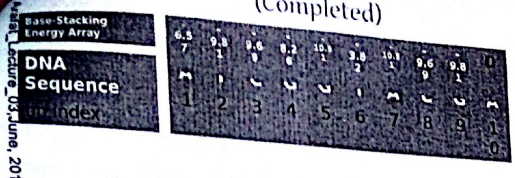
(How to calculate...)

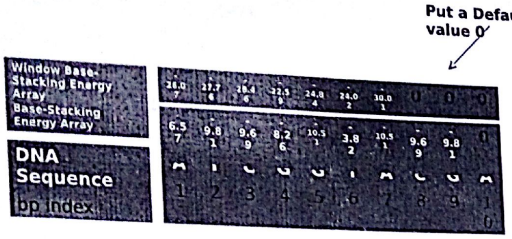


In this way.....

...ow-Based DNA Base-Stacking Energy ...ray

(Completed)





Problem Statements (Problem-I)

(Code this Problem)

Inputs :

1.DNA Sequence (fasta file)
2.Sliding Window Length

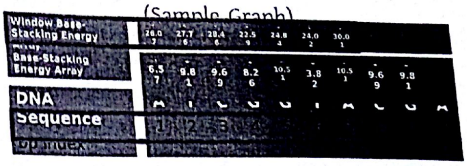
Outputs:

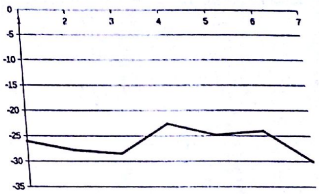
1.Print "bp index", "Window-Based Base-Stacking Energy"
2.Graph "bp index"[x-axis] Vs "Window-Based Base-Stacking Energy"[y-axis] (use matplotlib)
3.Return/Store info in the list of format [[x1,x2,x3,....],[y1,y2,y3, ...]]

where, x1 = bp index value, y1= Window-Based Base-Stacking Energy for bp-index x1.

Window-Based DNA Base-Stacking Energy Array

(Sample Graph)





Problem Statements (Problem-I)

(Code this Problem)

Inputs :

1.DNA Sequence (fasta file)
2.Sliding Window Length

Outputs:

1.Print "bp index", "Window-Based Base-Stacking Energy"
2.Graph "bp index"[x-axis] Vs "Window-Based Base-Stacking Energy"[y-axis] (use matplotlib)
3.Return/Store info in the list of format [[x1,x2,x3,....],[y1,y2,y3, ...]]

where, x1 = bp index value, y1= Window-Based Base-Stacking Energy for bp-index x1.

Problem Statements (Problem-II)

(Code this Problem)

Arafat_Lecture_03,June, 2017

Solve the similar problem of "Problem-I", just use the following table of "Propeller Twist" instead of " Base-Stacking Energy"

Dinucleotide	Propeller Twist (degrees)
AA	-18.66
AC	-13.10
AG	-14.00
AT	-15.01
CA	-9.45
CC	-8.11
CG	-10.03
CT	-14.00
GA	-13.48
GC	-11.08
GG	-8.11
GT	-13.10
TA	-11.85
TC	-13.48
TG	-9.45
TT	-18.66

Biopython (Part I)

Waiting for your CODE...

Acknowledgement:
This slide has been adopted from *Saddam Hossain*, Bio-Bio-1

What is Biopython?

- Biopython is a library for the Python programming language.
- Package that assists with processing biological data
- Consists of several modules – some with common operations, some more specialized
- Website: biopython.org

Object oriented programming

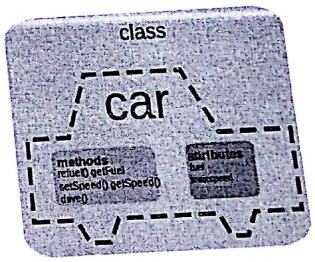
- Biopython is object-oriented
- Some knowledge helps understand how biopython works
- OOP is a way of organizing data and methods that work on them in a coherent package
- OOP helps structure and organize the code

Classes and objects

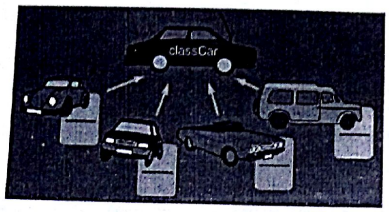
- A class:
  - is a user defined type
  - is a mold for creating objects
  - specifies how an object can contain and process data
  - represents an abstraction or a template for how an object of that class will behave

- An object is an *instance* of a class
- All objects have a type – shows which class they were made from

Attributes and methods



Class: Methods and Attributes



Class and
Instances

Class and object example

- Class: `MySeq`
- `MySeq` has:
  - attribute `length`
  - method `translate`

- An object of the class MySeq is created like this:
  - `myseq = MySeq("ATGGCCG")`
- Get sequence length:
  - `myseq.length`
- Get translation:
  - `myseq.translate()`

OOP Summary

- *An object has to be instantiated,* i.e. created, to exist
- *Every object has a certain type,* i.e. is of a certain class
- *The class decides which attributes and methods an object has*
- *Attributes and methods are accessed using . after the object variable name*

Installing Biopython

- Biopython relies on a few other Python packages for extra functionality. We'll use these:
  - numpy — efficient numerical functions and data structures (for Bio.PDB)
  - matplotlib — plotting (for Bio.Phylo)
- You can install Biopython from here:
  - http://biopython.org/wiki/Download
- Test your Biopython installation
  - `>>> import Bio`
  - `>>> print Bio. version`

Biopython functionality and tools

- Tools to parse bioinformatics files into Python data structures
- Supports the following formats
  - BLAST, Clustalw, FASTA
  - PubMed and Medline
  - ExPASy files
  - SCOP
  - SwissProt, PDB
- Files in the supported formats can be iterated over record by record or indexed and accessed via a dictionary interface

Biopython Documentation

- The Biopython Tutorial and Cookbook contains the bulk of Biopython documentation. It provides information to get you started with Biopython, in addition to specific documentation on a number of modules

  - http://biopython.org/DIST/docs/tutorial/Tutorial.pdf

Alphabets

Working with sequences

- Biopython has many ways of working with sequence data
- -Some components
  - Alphabet
  - Seq
  - SeqRecord
  - SeqIO
- There are other useful classes for working with alignments, blast searches and results etc are also available

Class `Alphabet`

- Every sequence needs an alphabet
- CCTTGGCC – DNA or protein?
- Biopython contains several alphabets
  - DNA
  - RNA
  - Protein
  - the three above with IUPAC codes ...and others
- Can all be found in Bio.Alphabet package

`Alphabet` example

```
>>> import Bio.Alphabet
>>> Bio.Alphabet.ThreeLetterProtein.letters
['Ala', 'Asx', 'Cys', 'Asp', 'Glu', 'Phe', 'Gly',
'His', 'Ile',
'Lys', 'Leu', 'Met', 'Asn', 'Pro', 'Gln', 'Arg',
'Ser', 'Thr',
'Sec', 'Val', 'Trp', 'Xaa', 'Tyr', 'Glx']

>>> from Bio.Alphabet import IUPAC
>>> IUPAC.IUPACProtein.letters
'ACDEFGHIKLMNPQRSTVWY'
>>> IUPAC.unambiguous_dna.letters
'GATC'
>>>
```

The `alphabet` attribute

- Alphabets are defined in the Bio.Alphabet module
- We will use the IUPAC alphabets (http://www.chem.qmw.ac.uk/iupac)
- `Bio.Alphabet.IUPAC` provides definitions for DNA, RNA and proteins + provides extension and customisation of basic definitions:
  - IUPACProtein (IUPAC standard AA)
  - ExtendedIUPACProtein (+ selenocysteine, X, etc)
  - IUPACUnambiguousDNA (basic GATC letters)
  - IUPACAmbiguousDNA (+ ambiguity letters)
  - ExtendedIUPACDNA (+ modified bases)
  - IUPACUnambiguousRNA
  - IUPACAmbiguousRNA

Packages, modules and classes

- What happens here?

```
>>> from Bio.Alphabet import IUPAC
>>> IUPAC.IUPACProtein.letters
```

- `Bio` and `Alphabet` are packages
  - packages contain modules
- IUPAC is a module
  - a module is a file with python code
- IUPAC module contains class IUPACProtein and other classes specifying alphabets
- IUPACProtein has attribute letters

Seq

Seq

Represents one sequence with its alphabet

- - Methods:
  - - translate()
  - - transcribe()
  - - complement()
  - - reverse_complement()
  - ...

Sequence Object

- - Seq objects vs Python strings:
  - - They have different methods
  - - The Seq object has the attribute alphabet
  - (biological meaning of Seq)

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

Using Seq

```
>>> from Bio.Seq import Seq
>>> import Bio.Alphabet
>>> seq = Seq("CCGGGTT", Bio.Alphabet.IUPAC.unambiguous_dna)
>>> seq
Seq('CCGGGTT', IUPACUnambiguousDNA())
>>> seq.transcribe()
Seq('CCGGGUU', IUPACUnambiguousRNA())
>>> seq.translate()
Seq('PG', IUPACProtein())
>>> seq = Seq("CCGGGUU", Bio.Alphabet.IUPAC.unambiguous_rna)
>>> seq.transcribe()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/site/VERSIONS/python-2.6.2/lib/python2.6/site-
packages/Bio/Seq.py",
  line 830, in transcribe
    raise ValueError("RNA cannot be transcribed!")
ValueError: RNA cannot be transcribed!
>>> seq.translate()
Seq('PG', IUPACProtein())
```

Transcription

```
from Bio.Seq import Seq
coding_dna = Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")
messenger_rna = coding_dna.transcribe()
cDNA = messenger_rna.back_transcribe()
print coding_dna
print messenger_rna
print cDNA
```

- All transcribe() does is a switch T --> U
- The Seq object also includes a back-transcription method

Translation

```
from Bio.Seq import Seq

coding_dna =
Seq("ATGGCCATTGTAATGGGCCGCTGAAAGGGTGCCCGATAG")
messenger_rna = coding_dna.transcribe()
protein1 = coding_dna.translate()
protein2 = messenger_rna.translate()

print protein1
Print protein2
```

Seq as a string

- Most string methods work on Seqs
- If string is needed, do str(seq)

```
>>> seq = Seq('CCGGGTTAACGTA',Bio.Alphabet.IUPAC.unambiguous_dna)
>>> seq[:5]
Seq('CCGGG', IUPACUnambiguousDNA())
>>> len(seq)
13
>>> seq.lower()
Seq('ccgggttaacgta', DNAAlphabet())
>>> print seq
CCGGGTTAACGTA
>>> list(seq)
['C', 'C', 'G', 'G', 'G', 'T', 'T', 'A', 'A', 'C', 'G', 'T', 'A']
>>> mystring = str(seq)
>>> print mystring
CCGGGTTAACGTA
>>> type(seq)
<class 'Bio.Seq.Seq'>
>>> type(mystring)
<type 'str'>
```

SeqRecord

SeqRecord('important')

- Seq contains the sequence and alphabet
- But sequences often come with a lot more
- SeqRecord = Seq + metadata
- Main attributes:
  - id - name or identifier
  - seq - seq object containing the sequence

```
>>> seq
Seq('CCGGGTTAACGTA', IUPACUnambiguousDNA())
>>> from Bio.SeqRecord import SeqRecord
>>> seqRecord = SeqRecord(seq, id='001')
>>> seqRecord
SeqRecord(seq=Seq('CCGGGTTAACGTA', IUPACUnambiguousDNA()),
id='001', name='<unknown name>', description='<unknown
description>',
dbxrefs=[])
>>>
```

SeqRecord attributes

Main attributes:

id - Identifier such as a locus tag (string)
seq - The sequence itself (Seq object or similar)

Additional attributes:

name - Sequence name, e.g. gene name (string)
description - Additional text (string)
dbxrefs - List of database cross references (list of strings)
features - Any (sub)features defined (list of SeqFeature objects)
annotations - Further information about the whole sequence (dictionary)
Most entries are strings, or lists of strings.
letter_annotations - Per letter/symbol annotation (restricted dictionary).
This holds Python sequences (lists, strings or tuples) whose length matches that of the sequence. A typical use would be to hold a list of integers representing sequencing quality scores, or a string representing the secondary structure.

Sequence Record objects

```
>>> from Bio.SeqRecord import SeqRecord
>>> help(SeqRecord)
```

Attributes:

id	- Identifier such as a locus tag (string)
seq	- The sequence itself (Seq object)
name	- Sequence name, e.g. gene name (string)
description	- Additional text (string)
dbxrefs	- List of db cross references (list of strings)
features	- Any (sub)features defined (list of SeqFeature objects)
annotations	- Further information about the whole sequence (dictionary)

Sequence Record objects

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord

tmp_seq =
Seq('MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF')

#define a Seq Record
seq_rec = SeqRecord(tmp_seq)
seq_rec.id = 'YP_025292.1'
seq_rec.description = 'toxic membrane protein'
print seq_rec
```

NCBI for sequence

SeqRecord('important')

- Seq contains the sequence and alphabet
- But sequences often come with a lot more
- SeqRecord = Seq + metadata
- Main attributes:
  - id – name or identifier
  - seq – seq object containing the sequence

```
>>> seq
Seq('CCGGGTTAACGTA', IUPACUnambiguousDNA())
>>> from Bio.SeqRecord import SeqRecord
>>> seqRecord = SeqRecord(seq, id='001')
>>> seqRecord
SeqRecord(seq=Seq('CCGGGTTAACGTA', IUPACUnambiguousDNA()),
id='001', name='<unknown name>', description='<unknown
description>',
dbxrefs=[])
>>>
```

Sequence Record objects

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord

tmp_seq =
Seq('MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF')

#another way to define a Seq Record
seq_rec =
SeqRecord(Seq('MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTE
VAVF'), id = 'YP_025292.1', name='HokC',
description='toxic membrane protein',dbxrefs=[])

print seq_rec
```

SeqRecord in practice...

```
>>> from Bio.SeqRecord import SeqRecord
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import DNAAlphabet
>>> seqRecord = SeqRecord(Seq('GCAGCCTCAAACCCCAGCTG',
… DNAAlphabet), id = 'NM_005368.2', name = 'NM_005368',
… description = 'Myoglobin var 1',
… dbxrefs = ['GeneID:4151', 'HGNC:6915'])
>>> seqRecord.annotations['note'] = 'Information goes
here'
>>> seqRecord
SeqRecord(seq=Seq('GCAGCCTCAAACCCCAGCTG',
 <class 'Bio.Alphabet.DNAAlphabet'>), id='NM_005368.2',
name='NM_005368', description='Myoglobin var 1',
dbxrefs=['GeneID:4151', 'HGNC:6915'])
>>> seqRecord.annotations
{'note': 'Information goes here'}
```

The `format()` method

It returns a string containing your record formatted using one of the output file formats supported by Bio.SeqIO

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio.Alphabet import generic_protein

rec =
SeqRecord(Seq("MGSNKSKPKDASQRRRSLEPSENVHGAGGAFPA
SQTPSKPASADGHRGPSAAFVPPAAEPKLFGGFNSSDTVTSPQRAGAL
AGGVTTFVALYDYESRTETDLSFKKGERLQIVNNTRKVDVREGDWWLA
HSLSTGQTGYIPS", generic_protein), id = "P05480",
description = "SRC_MOUSE Neuronal proto-oncogene
tyrosine-protein kinase Src: MY TEST")

print rec.format("fasta")
```

SeqIO

SeqIO

- How to get sequences in and out of files
- Retrieves sequences as SeqRecords, can write SeqRecords to files

- Reading:
  - parse(filehandle, format)
  - returns a generator that gives SeqRecords
- Writing:
  - write(SeqRecord(s), filehandle, format)

Bio.SeqIO

- Sequence I/O
  - Parsing or Reading Sequences
  - Writing Sequence Files

- A simple interface for working with assorted file formats in a uniform way

```
>>>from Bio import SeqIO
>>>help(SeqIO)
```

SeqIO.parse()

- Reads in sequence data as SeqRecord objects

- It expects two arguments:
  - An object (called handle) to read the data. It can be:
    - a file opened for reading
    - the output from a command line program
    - data downloaded from the internet
  - A lower case string specifying the sequence format

- The object returned by SeqIO.parse() is an iterator which returns SeqRecord objects

SeqIO.parse() Examples

```
>>> from Bio import SeqIO
>>> handle = open("P05480.fasta")
>>> for seq_rec in SeqIO.parse(handle,"fasta"):
... print seq_rec.id
... print repr(seq_rec.seq)
... print len(seq_rec)
... sp|P05480|SRC_MOUSE
Seq('MGSNKSKPKDASQRRRSLERGPSA...ENL',
SingleLetterAlphabet())
541
>>> handle.close()
```

SeqIO.parse() Examples

```
>>> from Bio import SeqIO
>>> handle = open("P05480.fasta")
>>> for seq_rec in SeqIO.parse(handle,"fasta"):
... print seq_rec.id
... print repr(seq_rec.seq)
... print len(seq_rec)
... sp|P05480|SRC_MOUSE
Seq('MGSNKSKPKDASQRRRSLERGPSA...ENL',
SingleLetterAlphabet())
541
>>> handle.close()
```


SeqIO.parse() Examples

```
from Bio import SeqIO

handle = open("yeast_genes.gbk")
record = SeqIO.parse(handle, "genbank")

for seq_rec in record:
    print seq_rec.id
    print str(seq_rec.seq)
    print len(seq_rec)

handle.close()
```

SeqIO.parse() Examples

```
from Bio import SeqIO

handle = open("yeast_genes.gbk")

for seq_rec in SeqIO.parse(handle, "genbank"):
    print seq_rec.id
    print repr(seq_rec.seq)
    print len(seq_rec)

handle.close()
```


Iterating over the records in a multiple sequence file

```
>>> from Bio import SeqIO
>>> handle = open("SwissProt-Human.fasta")
>>> all_rec = SeqIO.parse(handle,"fasta")
>>> for rec in all_rec:
... print rec.id
...
sp|P31946|1433B_HUMAN
sp|P62258|1433E_HUMAN
sp|Q04917|1433F_HUMAN
sp|P61981|1433G_HUMAN
sp|P31947|1433S_HUMAN
sp|P27348|1433T_HUMAN
sp|P63104|1433Z_HUMAN
>>> handle.close()
```

Parsing sequences from the net

- Handles are not always from files

```
import urllib2

handle = urllib2.urlopen("http://
www.uniprot.org/uniprot/B2TYV6.fasta")

F = handle.read()

print F
```


Writing files

```
from Bio import SeqIO
sequences = ... # add code here
output_handle = open("example.fasta", "w")
SeqIO.write(sequences, output_handle, "fasta")
output_handle.close()
```

sequences are here a list of SeqRecords

- Note: sequences is here a list
- Can write any iterable containing SeqRecords to a file
- Can also write a single sequence

Bio.SeqIO.write()

- This function takes three arguments:
  - 1. SeqRecord objects
  - 2. a handle to write to
  - 3. a sequence format

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO

Rec1 = SeqRecord(Seq("ACCA"), id="1", description="")
Rec2 = SeqRecord(Seq("CDRFAA"), id="2", description="")
Rec3 = SeqRecord(Seq("GRKLM"), id="3", description="")

My_records = [Rec1, Rec2, Rec3]
handle = open("MySeqs.fas","w")
SeqIO.write(My_records, handle, "fasta")

handle.close()
```


Bio.SeqIO.write()

- This function takes three arguments:
  - 1. SeqRecord objects
  - 2. a handle to write to
  - 3. a sequence format

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord
from Bio import SeqIO

Rec1 = SeqRecord(Seq("ACCA"), id="1", description="")
Rec2 = SeqRecord(Seq("CDRFAA"), id="2", description="")
Rec3 = SeqRecord(Seq("GRKLM"), id="3", description="")

My_records = [Rec1, Rec2, Rec3]
handle = open("MySeqs.fas","w")
SeqIO.write(My_records, handle, "fasta")

handle.close()
```

Sequence Conversion

- You can do file conversion by combining Bio.SeqIO.parse() and Bio.SeqIO.write()

```
from Bio import SeqIO

In_handle = open ("1293613.gbk", "r")
Out_handle = open("1293613.fasta", "w")

records = SeqIO.parse(In_handle, "genbank")
SeqIO.write(records, Out_handle, "fasta")

In_handle.close()
Out_handle.close()
```

- More to come next week ...

Arafat_Lecture_03,June,2017

Eutils ব্যবহার করে Entrez থেকে NCBI থেকে সরাসরি ডাটাবেজ ,

stand alone blast software দিয়ে সরাসরি blast করা যায় ,