同济大学 TONGJI UNIVERSITY

生物信息学系 DEPARTMENT OF BIOINFORMATICS

Section 15

Python: Data Visualization Biopython

Yong Zhang, Ph. D

School of Life Science and Technology
Tongji University

Apr 28, 2019

yzhang@tongji.edu.cn

Story: nucleotide frequencies in the ribosome

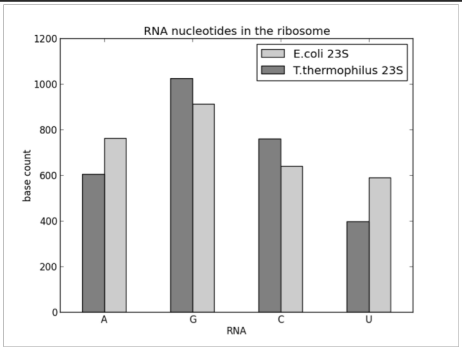
Species	A	C	G	U
T. thermophilus	606	759	1024	398
E. coli	762	639	912	591

- The above table contains the exact number of nucleotides for the 23S subunit of the *Thermus thermophilus* and *Escherichia coli* ribosomes.
- You will learn how to display these data in a more attractive way.

Example Python session

```
from pylab import figure, title, xlabel, ylabel, xticks, bar, \
                  legend, axis, savefig

nucleotides = ["A", "G", "C", "U"]

counts = [
    [606, 1024, 759, 398],
    [762, 912, 639, 591],
    ]

figure()
title('RNA nucleotides in the ribosome')
xlabel('RNA')
ylabel('base count')

x1 = [2.0, 4.0, 6.0, 8.0]
x2 = [x - 0.5 for x in x1]

xticks(x1, nucleotides)

bar(x1, counts[1], width=0.5, color="#cccccc", label="E.coli 23S")
bar(x2, counts[0], width=0.5, color="#808080", label="T.thermophilus 23S")

legend()
axis([1.0, 9.0, 0, 1200])
savefig('barplot.png')
```

Example Python session



The *matplotlib* library

- You can use matplotlib to create a diagram in just four steps:
  - The library is imported
  - You start a new figure.
  - You plot some data.
  - You save the figure to a .png file.

figure() function

```
from pylab import figure, title, xlabel, ylabel, xticks, bar, \
                  legend, axis, savefig

nucleotides = ["A", "G", "C", "U"]

counts = [
    [606, 1024, 759, 398],
    [762, 912, 639, 591],
    ]

figure()
title('RNA nucleotides in the ribosome')
xlabel('RNA')
ylabel('base count')

x1 = [2.0, 4.0, 6.0, 8.0]
x2 = [x - 0.5 for x in x1]

xticks(x1, nucleotides)

bar(x1, counts[1], width=0.5, color="#cccccc", label="E.coli 23S")
bar(x2, counts[0], width=0.5, color="#808080", label="T.thermophilus 23S")

legend()
axis([1.0, 9.0, 0, 1200])
savefig('barplot.png')
```

- Each time you call the *figure()* function, the background is cleared, and you start creating a new plot.

Drawing vertical bars

```
from pylab import figure, title, xlabel, ylabel, xticks, bar, \
                  legend, axis, savefig

nucleotides = ["A", "G", "C", "U"]

counts = [
    [606, 1024, 759, 398],
    [762, 912, 639, 591],
    ]

figure()
title('RNA nucleotides in the ribosome')
xlabel('RNA')
ylabel('base count')

x1 = [2.0, 4.0, 6.0, 8.0]
x2 = [x - 0.5 for x in x1]

xticks(x1, nucleotides)

bar(x1, counts[1], width=0.5, color="#cccccc", label="E.coli 23S")
bar(x2, counts[0], width=0.5, color="#808080", label="T.thermophilus 23S")

legend()
axis([1.0, 9.0, 0, 1200])
savefig('barplot.png')
```

Drawing vertical bars

- The bar() function simply draws bars.
- In the simplest version, it takes as parameters two lists: one for the *x*-axis values and one for the bar heights:

```
bar([1, 2, 3], [20, 50, 100])
```

  - The first list [1, 2, 3] indicates where the bars should start on the *x*-axis; the second list [20, 50, 100] indicates how high each bar should be.
- Similarly, you can create horizontal bars with the barh() function:

```
barh([1, 2, 3], [20, 50, 100])
```

Adding tick marks

```
from pylab import figure, title, xlabel, ylabel, xticks, bar, \
                  legend, axis, savefig

nucleotides = ["A", "G", "C", "U"]

counts = [
    [606, 1024, 759, 398],
    [762, 912, 639, 591],
    ]

figure()
title('RNA nucleotides in the ribosome')
xlabel('RNA')
ylabel('base count')

x1 = [2.0, 4.0, 6.0, 8.0]
x2 = [x - 0.5 for x in x1]

xticks(x1, nucleotides)

bar(x1, counts[1], width=0.5, color="#cccccc", label="E.coli 23S")
bar(x2, counts[0], width=0.5, color="#808080", label="T.thermophilus 23S")

legend()
axis([1.0, 9.0, 0, 1200])
savefig('barplot.png')
```

Adding tick marks

- The *xticks()* and *yticks()* functions draw customized ticks.
- *xticks(xpos, bases)* writes each element of the bases list of strings at the positions xpos on the *x*-axis. The *yticks* function works similarly.

Adding a legend box

```
from pylab import figure, title, xlabel, ylabel, xticks, bar, \
                  legend, axis, savefig

nucleotides = ["A", "G", "C", "U"]

counts = [
    [606, 1024, 759, 398],
    [762, 912, 639, 591],
    ]

figure()
title('RNA nucleotides in the ribosome')
xlabel('RNA')
ylabel('base count')

x1 = [2.0, 4.0, 6.0, 8.0]
x2 = [x - 0.5 for x in x1]

xticks(x1, nucleotides)

bar(x1, counts[1], width=0.5, color="#cccccc", label="E.coli 23S")
bar(x2, counts[0], width=0.5, color="#808080", label="T.thermophilus 23S")

legend()
axis([1.0, 9.0, 0, 1200])
savefig('barplot.png')
```

- The *legend()* function takes the labels of all data sets that were plotted so far and writes them to a legend box in the order of appearance. It does not need any argument.

Adding a figure title

```
from pylab import figure, title, xlabel, ylabel, xticks, bar, \
                  legend, axis, savefig

nucleotides = ["A", "G", "C", "U"]

counts = [
    [606, 1024, 759, 398],
    [762, 912, 639, 591],
    ]

figure()
title('RNA nucleotides in the ribosome')
xlabel('RNA')
ylabel('base count')

x1 = [2.0, 4.0, 6.0, 8.0]
x2 = [x - 0.5 for x in x1]

xticks(x1, nucleotides)

bar(x1, counts[1], width=0.5, color="#cccccc", label="E.coli 23S")
bar(x2, counts[0], width=0.5, color="#808080", label="T.thermophilus 23S")

legend()
axis([1.0, 9.0, 0, 1200])
savefig('barplot.png')
```

- The *title()* function simply adds text on the top of the diagram.

Setting the boundaries of the diagram

```
from pylab import figure, title, xlabel, ylabel, xticks, bar, \
                  legend, axis, savefig

nucleotides = ["A", "G", "C", "U"]

counts = [
    [606, 1024, 759, 398],
    [762, 912, 639, 591],
    ]

figure()
title('RNA nucleotides in the ribosome')
xlabel('RNA')
ylabel('base count')

x1 = [2.0, 4.0, 6.0, 8.0]
x2 = [x - 0.5 for x in x1]

xticks(x1, nucleotides)

bar(x1, counts[1], width=0.5, color="#cccccc", label="E.coli 23S")
bar(x2, counts[0], width=0.5, color="#808080", label="T.thermophilus 23S")

legend()
axis([1.0, 9.0, 0, 1200])
savefig('barplot.png')
```

Setting the boundaries of the diagram

- *matplotlib* automatically chooses the extent of the diagram in such a way that all data will be visible, but sometimes, this is not what you want.
- *axis()* takes a list of four values: the lower and upper boundaries for both the *x*-axis and the *y*-axis, as follows:

```
axis([lower_x, upper_x, lower_y, upper_y])
```

Exporting an image file

```
from pylab import figure, title, xlabel, ylabel, xticks, bar, \
                  legend, axis, savefig

nucleotides = ["A", "G", "C", "U"]

counts = [
    [606, 1024, 759, 398],
    [762, 912, 639, 591],
    ]

figure()
title('RNA nucleotides in the ribosome')
xlabel('RNA')
ylabel('base count')

x1 = [2.0, 4.0, 6.0, 8.0]
x2 = [x - 0.5 for x in x1]

xticks(x1, nucleotides)

bar(x1, counts[1], width=0.5, color="#cccccc", label="E.coli 23S")
bar(x2, counts[0], width=0.5, color="#808080", label="T.thermophilus 23S")

legend()
axis([1.0, 9.0, 0, 1200])
savefig('barplot.png')
```

Exporting an image file

- The *savefig()* function writes the entire diagram to an image file in .png format.
  - By default, a 600 × 600 pixel image with 100 dpi will be created.
  - You can create higher resolution images by adding a precise dpi value:

```
savefig('barplot.png', dpi = 300)
```

  - You can also export to .tif and .eps formats directly:

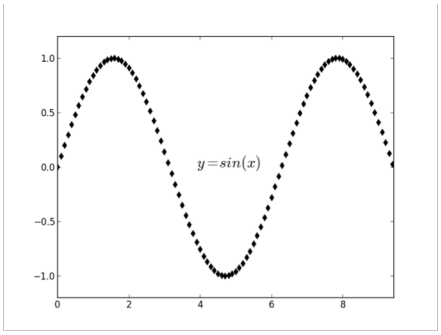
```
savefig('barplot.tif', dpi = 300)
```

Plot a function

- The *plot()* function requires two lists of values that have the same length.
- The following example plots a sine function.

```
from pylab import figure, plot, text, axis, savefig
import math

figure()

xdata = [0.1 * i for i in range(100)]
ydata = [math.sin(j) for j in xdata]

plot(xdata, ydata, 'kd', linewidth=1)
text(4.8, 0, "$y = sin(x)$", horizontalalignment='center', fontsize=20)
axis([0, 3 * math.pi, -1.2, 1.2])

savefig('sinfunc.png')
```

Plot a function



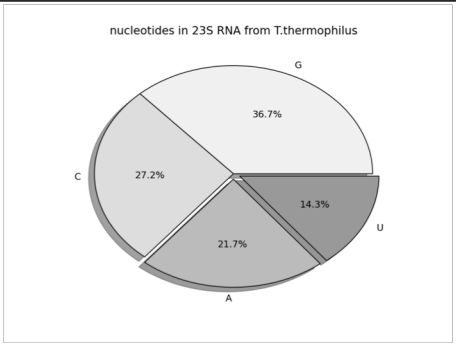
Plot a function

- The third parameter in the *plot* function, *'kd'*, indicates the color and style of the line.
  - The first character stands for the color (*k*: black, *r*: red, *g*: green, *b*: blue; others exist).
  - The second character indicates the symbol used for drawing (*o*: circles, *s*: squares, *v* and ^: triangles, *d*: diamonds, +: crosses, -: lines, :: dotted lines, .: dots).
- The *text()* function adds a text marked by the enclosing *$* symbols at a given *x*/*y* position using the mathematical TeX notation.

Drawing a pie chart

```
from pylab import figure, title, pie, savefig

nucleotides = 'G', 'C', 'A', 'U'
count = [1024, 759, 606, 398]
explode = [0.0, 0.0, 0.05, 0.05]

colors = ["#f0f0f0", "#dddddd", "#bbbbbb", "#999999"]

def get_percent(value):
    '''Formats float values in pie slices to percent.'''
    return "%4.1f%%" % (value)

figure(1)
title('nucleotides in 23S RNA from T.thermophilus')

pie(count, explode=explode, labels=nucleotides, shadow=True,
    colors=colors, autopct=get_percent)

savefig('piechart.png', dpi=150)
```

Drawing a pie chart



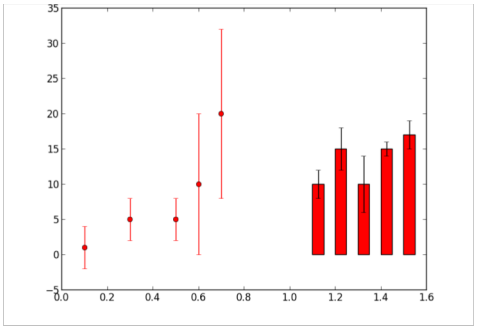
Drawing a pie chart

- By the values in the *explode* list, you can move pie slices out from the center (a *0.0* means it attaches to the other slices).
- The *autopct* parameter is a function that gets the size fraction of a slice (from 0.0 to 1.0) and returns a string.
- The *get_percent* function converts the number to a string and adds a percent symbol.

Adding error bars

```
from pylab import figure, errorbar, bar, savefig

figure()

# scatterplot with error bars
x1 = [0.1, 0.3, 0.5, 0.6, 0.7]
y1 = [1, 5, 5, 10, 20]
err1 = [3, 3, 3, 10, 12]
errorbar(x1, y1, err1 , fmt='ro')

# barplot with error bars
x2 = [1.1, 1.2, 1.3, 1.4, 1.5]
y2 = [10, 15, 10, 15, 17]
err2 = (2, 3, 4, 1, 2)
width = 0.05
bar(x2, y2, width, color='r', yerr=err2, ecolor="black")

savefig('errorbars.png')
```

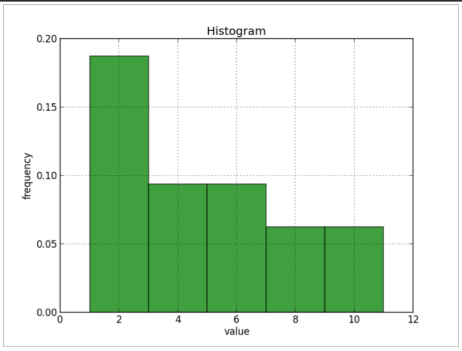
Adding error bars



Drawing a histogram

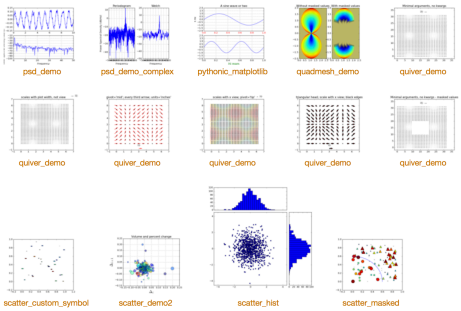
```
from pylab import figure, title, xlabel, ylabel, \
                  hist, axis, grid, savefig

data = [1, 1, 9, 1, 3, 5, 8, 2, 1, 5, 11, 8, 3, 4, 2, 5]
n_bins = 5

figure()
num, bins, patches = hist(data, n_bins, normed=1.0, histtype='bar', \
                          facecolor='green', alpha=0.75)

title('Histogram')
xlabel('value')
ylabel('frequency')
axis()
grid(True)

savefig('histogram.png')
```

Drawing a histogram



The *matplotlib* gallery

- http://matplotlib.org/gallery.html



Biopython introduction

- Biopython (http://biopython.org/wiki/Main_Page) is a collection of modules for computational molecular biology, which allows performing many of the basic tasks required in a bioinformatics project:
  - parsing (i.e., extracting information from) file formats for gene and protein sequences, protein structures, PubMed records, etc.;
  - downloading files from repositories such as NCBI, ExPASy, etc.;
  - running (locally or remotely) popular bioinformatics algorithms such as BLAST, ClustalW, etc.;
  - running Biopython implementations of algorithms for clustering, machine learning, data analysis, and data visualization.

Biopython introduction

- Use Biopython when
  - its modules and/or methods fit your needs,
  - your task is unchallenging or boring (don't "re-invent the wheel" unless you're doing it as a learning exercise),
  - your task will take you a lot of writing effort.
- Extend Biopython (i.e., modify the Biopython source code) when
  - the Biopython modules and/or methods almost do what you need but do not exactly fit your need.

Story: translate a DNA sequence

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

# read the input sequence
dna = open("hemoglobin-gene.txt").read().strip()
dna = Seq.Seq(dna, IUPAC.unambiguous_dna)

# transcribe and translate
mrna = dna.transcribe()
protein = mrna.translate()

# write the protein sequence to a file
protein_record = SeqRecord(protein, id='sp|P69905.2|HBA_HUMAN',
                           description="Hemoglobin subunit alpha, Homo sapiens")

outfile = open("HBA_HUMAN.fasta", "w")
SeqIO.write(protein_record, outfile,"fasta")
outfile.close()
```

The *Seq* object

- *Seq.Seq* class creates a *sequence* object.

```
>>> from Bio import Seq
>>> my_seq = Seq.Seq("AGCATCGTAGCATGCAC")
>>> my_seq
Seq('AGCATCGTAGCATGCAC', Alphabet())
```

Bio.Alphabet module

- Biopython contains a set of precompiled alphabets that cover all biological sequence types.
  - IUPAC-defined alphabets are the most frequently used.
  - You should import the *IUPAC* module from the *Bio.Alphabet* module.
    - IUPACUnambiguousDNA (basic ACGT letters)
    - IUPACAmbiguousDNA (includes ambiguous letters)
    - ExtendedIUPACDNA (includes modified bases)
    - IUPACUnambiguousRNA
    - IUPACAmbiguousRNA
    - IUPACProtein (IUPAC standard amino acids)
    - ExtendedIUPACProtein (includes selenocysteine, X, etc.)

Transcribing sequences

- You can obtain the transcription of a DNA sequence using the *transcribe* method.

```
>>> from Bio import Seq
>>> from Bio.Alphabet import IUPAC
>>>
>>> # input DNA sequence is the coding strand
...
>>> my_seq = Seq.Seq("AGCATCGTAGCATGCAC")
>>> my_seq.transcribe()
Seq('AGCAUCGUAGCAUGCAC', RNAAlphabet())
>>>
>>> # input DNA sequence is thetemplate strand
...
>>> dna = Seq.Seq("AGCATCGTAGCATGCAC", IUPAC.unambiguous_dna)
>>> cdna = dna.reverse_complement()
>>> mrna = cdna.transcribe()
>>> mrna
Seq('GUGCAUGCUACGAUGCU', IUPACUnambiguousRNA())
>>>
>>> # in a single command line
...
>>> dna.reverse_complement().transcribe()
Seq('GUGCAUGCUACGAUGCU', IUPACUnambiguousRNA())
```

CodonTable module

- A number of genetic codes are available through the *CodonTable* module of the *Bio.Data* module:

```
>>> from Bio.Data import CodonTable
>>> standard_table = CodonTable.unambiguous_dna_by_name["Standard"]
>>> standard_table.start_codons
['TTG', 'CTG', 'ATG']
>>> standard_table.stop_codons
['TAA', 'TAG', 'TGA']
>>> mito_table =CodonTable.unambiguous_dna_by_name["Vertebrate Mitochondrial"]
>>> mito_table.start_codons
['ATT', 'ATC', 'ATA', 'ATG', 'GTG']
>>> mito_table.stop_codons
['TAA', 'TAG', 'AGA', 'AGG']
```

Translating sequences

```
>>> from Bio import Seq
>>> from Bio.Alphabet import IUPAC
>>> mrna = Seq.Seq('AUGGCCAUUGUAAUGGGCCGCUGAAAGGGAUAG',\
... IUPAC.unambiguous_rna)
>>> mrna.translate(table = "Standard")
Seq('MAIVMGR*KG*', HasStopCodon(IUPACProtein(), '*'))
>>> mrna.translate(table = "Vertebrate Mitochondrial")
Seq('MAIVMGRWKG*', HasStopCodon(IUPACProtein(), '*'))
>>>
>>> # to stop at the first encountered stop codon
...
>>> mrna.translate(to_stop = True, table = 1)
Seq('MAIVMGR', IUPACProtein())
>>> mrna.translate(to_stop = True, table = 2)
Seq('MAIVMGRWKG', IUPACProtein())
```

Working with sequences as strings

```
>>> from Bio import Seq
>>> from Bio.Alphabet import IUPAC
>>>
>>> my_seq = Seq.Seq("AGCATCGTAGCATGCAC", IUPAC.unambiguous_dna)
>>> my_seq[0]
'A'
>>> my_seq[0:3]
Seq('AGC', IUPACUnambiguousDNA())
>>> my_seq.split('TC')
[Seq('AGCA', IUPACUnambiguousDNA()), Seq('GTAGCATGCAC', IUPACUnambiguousDNA())]
>>> my_seq.count('A')
5
>>> my_seq.count('A') / float(len(my_seq))
0.2941176470588235
>>>
>>> my_seq_2 = Seq.Seq("CGTC", IUPAC.unambiguous_dna)
>>> my_seq + my_seq_2
Seq('AGCATCGTAGCATGCACCGTC', IUPACUnambiguousDNA())
>>>
>>> my_seq.find("TCGT")  # position of the leftmost matching character
4
>>> my_seq.find("TTTT")  # subsequence is not found
-1
>>>
>>> # Some functions can be used on strings directly
...
>>> my_seq_str = "AGCATCGTAGCATGCAC"
>>> Seq.transcribe(my_seq_str)
'AGCAUCGUAGCAUGCAC'
>>> Seq.translate(my_seq_str)
'SIVAC'
```

The *MutableSeq* object

```
>>> from Bio import Seq
>>> from Bio.Alphabet import IUPAC
>>>
>>> my_seq = Seq.Seq("AGCATCGTAGCATGCAC", IUPAC.unambiguous_dna)
>>> my_seq[5] = 'T' # Seq objects are immutable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Seq' object does not support item assignment
>>>
>>> my_m_seq = Seq.MutableSeq("AGCATCGTAGCATG", IUPAC.unambiguous_dna)
>>> my_m_seq[5] = "T" # MutableSeq objects are mutable
>>> my_m_seq
MutableSeq('AGCATTGTAGCATG', IUPACUnambiguousDNA())
>>>
>>> my_mut_seq = my_seq.tomutable()  # Seq to MutableSeq
>>> my_mut_seq
MutableSeq('AGCATCGTAGCATGCAC', IUPACUnambiguousDNA())
>>> my_imu_seq = my_m_seq.toseq()  # MutableSeq to Seq
>>> my_imu_seq
Seq('AGCATTGTAGCATG', IUPACUnambiguousDNA())
```

The *SeqRecord* object

- The *SeqRecord* class provides a container for a sequence and its annotation.
  - *seq*: This is a biological sequence, typically in the form of a *Seq* object.
  - *id*: This is the primary ID used to identify the sequence.
  - *name*: This is a "common" molecule name.
  - *description*: This is a description of the sequence/molecule.
  - *letter_annotations*: This is a dictionary with per-residue annotations. Keys are the type of annotation (e.g., "secondary structure"), and values are Python sequences (lists, tuples, or strings) having the same length as the sequence, where each element is a per-residue annotation. It is useful for assigning quality scores, secondary structure or accessibility preferences, etc. to residues.

The *SeqRecord* object

- *annotations*: This is a dictionary of additional information about the sequence. The keys are the type of information, and the information is contained in the value.
- *features*: This is a list of *SeqFeature* objects, with more structured information about sequence features (e.g., position of genes on a genome, or domains on a protein sequence; see following text).
- *dbxrefs*: This is a list of database cross-references.

The *SeqRecord* object

- To retrieve and assign *SeqRecord* object features.

```
>>> protein_record.id
'sp|P69905.2|HBA_HUMAN'
>>> protein_record.description
'Hemoglobin subunit alpha, Homo sapiens'
>>> protein_record.name = "Hemoglobin"  # to assign features
>>>
>>> protein_record.annotations["origin"] = "human"
>>> protein_record.annotations["subunit"] = "alpha"
>>> protein_record.annotations
{'origin': 'human', 'subunit': 'alpha'}
```

Converting *SeqRecord* objects to file formats

```
>>> print protein_record.format("fasta")
>sp|P69905.2|HBA_HUMAN Hemoglobin subunit alpha, Homo sapiens
MVLSPADKTNVKAAWGKVGAHAGEYGAEALERMFLSFPTTKTYFPHFDLSHGSAQVKGHG
KKVADALTNAVAHVDDMPNALSALSDLHAHKLRVDPVNFKLLSHCLLVTLAAHLPAEFTP
AVHASLDKFLASVSTVLTSKYR*

>>> print protein_record.format("genbank")
LOCUS       Hemoglobin               143 aa                      UNK 01-JAN-1980
DEFINITION  Hemoglobin subunit alpha, Homo sapiens
ACCESSION   sp|P69905
VERSION     sp|P69905
KEYWORDS    .
SOURCE      .
  ORGANISM  .
            .
FEATURES             Location/Qualifiers
ORIGIN
        1 mvlspadktn vkaawgkvga hageygaeal ermflsfptt ktyfphfdls hgsaqvkghg
       61 kkvadaltna vahvddmpna lsalsdlhah klrvdpvnfk llshcllvtl aahlpaeftp
      121 avhasldkfl asvstvltsk yr*
//
```

The *SeqIO* module

- Arguments of *SeqIO.parse()* and *SeqIO.read()* methods:
  - a file (mandatory; also called a "handle" object) that specifies where the data must be read from;
  - a string indicating the format of the data (mandatory; e.g., "fasta" or "genbank";
  - an argument that specifies the alphabet of the sequence data (optional).
- *SeqIO.parse()* returns an iterator that produces *SeqRecord* objects from an input file of several records.
  - You can use the iterator like a list in *for* or *while* loops.
- *SeqIO.read()* parses only single-record files by first checking whether there is only one record in the handle and raises an error if this condition is not met.

The *SeqIO* module

- For a big number of records, you can use *SeqIO.index()* method.
  - Two arguments are required: a record filename and a file format.
  - It returns a dictionary-like object that gives you access to all records without keeping all data in the memory.
  - The dictionary keys are the IDs of the records, and the values contain the entire record.
  - This method allows you to manipulate huge files, with a little cost in flexibility and speed.

The *SeqIO* module

- The *SeqIO.write()* method writes one or more *SeqRecord* objects to a file in the format specified by the user.
  - The method requires three arguments: one or more *SeqRecord* objects, a handle object (i.e., a file opened with the "*w*" modality) or a filename to write to, and a sequence format (e.g., "*fasta*" or "*genbank*").
  - The first argument can be a list, an iterator, or an individual *SeqRecord*.
  - When writing GenBank files, the alphabet must be set for the sequence.

Examples

- Using the *Bio.SeqIO* module to parse a multiple sequence FASTA file.

```
from Bio import SeqIO

fasta_file = open("Uniprot.fasta","r")
for seq_record in SeqIO.parse(fasta_file, "fasta"):
    print seq_record.id
    print repr(seq_record.seq)
    print len(seq_record)

fasta_file.close()
```

Examples

- Using the *SeqIO* module to parse a record file and store its content in a list or a dictionary.

```
from Bio import SeqIO

# read fasta entries to a list
uniprot_iterator = SeqIO.parse("Uniprot.fasta", "fasta")
records = list(uniprot_iterator)
print records[0].id
print records[0].seq

# read fasta entries to a dictionary
uniprot_iterator = SeqIO.parse("Uniprot.fasta", "fasta")
records = SeqIO.to_dict(uniprot_iterator)
print records['sp|P03372|ESR1_HUMAN'].id
print records['sp|P03372|ESR1_HUMAN'].seq
```

Examples

- Using *SeqIO.index()* to parse a big file.

```
from Bio import SeqIO

records = SeqIO.index("Uniprot.fasta","fasta")
for key in records.keys():
    print key, len(records[key].seq)
```

Examples

- Converting between sequence file formats.

```
from Bio import SeqIO

genbank_file = open ("AY810830.gbk", "r")
output_file = open("AY810830.fasta", "w")
records = SeqIO.parse(genbank_file, "genbank")
SeqIO.write(records, output_file, "fasta")
output_file.close()
```

Story: searching publications

```
from Bio import Entrez
from Bio import Medline

keyword = "PyCogent"

# search publications in PubMed
Entrez.email = "my_email@address.com"
handle = Entrez.esearch(db="pubmed", term=keyword)
record = Entrez.read(handle)
pmids = record['IdList']
print pmids

# retrieve Medline entries from PubMed
handle = Entrez.efetch(db="pubmed", id=pmids, rettype="medline", retmode="text")
medline_records = Medline.parse(handle)
records = list(medline_records)

n = 1
for record in records:
    if keyword in record["TI"]:
        print n, ')', record["TI"]
        n += 1
```

The *Entrez* module

```
from Bio import Entrez
from Bio import Medline

keyword = "PyCogent"

# search publications in PubMed
Entrez.email = "my_email@address.com"
handle = Entrez.esearch(db="pubmed", term=keyword)
record = Entrez.read(handle)
pmids = record['IdList']
print pmids

# retrieve Medline entries from PubMed
handle = Entrez.efetch(db="pubmed", id=pmids, rettype="medline", retmode="text")
medline_records = Medline.parse(handle)
records = list(medline_records)

n = 1
for record in records:
    if keyword in record["TI"]:
        print n, ')', record["TI"]
        n += 1
```

- *Entrez* provides a connection to the *esearch* and *efetch* tools on the NCBI servers.

The *Entrez* module

```
from Bio import Entrez
from Bio import Medline

keyword = "PyCogent"

# search publications in PubMed
Entrez.email = "my_email@address.com"
handle = Entrez.esearch(db="pubmed", term=keyword)
record = Entrez.read(handle)
pmids = record['IdList']
print pmids

# retrieve Medline entries from PubMed
handle = Entrez.efetch(db="pubmed", id=pmids, rettype="medline", retmode="text")
medline_records = Medline.parse(handle)
records = list(medline_records)

n = 1
for record in records:
    if keyword in record["TI"]:
        print n, ')', record["TI"]
        n += 1
```

- The *mail* attribute tells your email address to NCBI. This is not mandatory.

Entrez.esearch()

```
from Bio import Entrez
from Bio import Medline

keyword = "PyCogent"

# search publications in PubMed
Entrez.email = "my_email@address.com"
handle = Entrez.esearch(db="pubmed", term=keyword)
record = Entrez.read(handle)
pmids = record['IdList']
print pmids

# retrieve Medline entries from PubMed
handle = Entrez.efetch(db="pubmed", id=pmids, rettype="medline", retmode="text")
medline_records = Medline.parse(handle)
records = list(medline_records)

n = 1
for record in records:
    if keyword in record["TI"]:
        print n, ')', record["TI"]
        n += 1
```

- The *Entrez.esearch()* conducts searches in NCBI databases using a query text. The function takes two mandatory arguments: *db*, the database to search (default is *pubmed*), and *term*, the query text.

Entrez.esearch()

- If you want to search more than one keyword you can use "*AND*" or "*OR*".
- You can also use keyword specifications such as *[Year]*, *[Organism]*, *[Gene]*, etc.
- *Entrez.esearch()* returns a list of database identifiers in the form of a "handle," which can be read using the *Entrez.read()* function.
- *Entrez.read()* function returns a dictionary with the keys that include "*IdList*" (its value is a list of IDs matching the text query) and "*Count*" (its value is the total number of IDs).

Entrez.esearch()

- You can use the optional parameter *retmax* (maximum retrieved) to set how many entries matching the query text are to be retrieved
- Other useful optional arguments are *datetype*, *reldate*, *mindate*, and *maxdate* (both in the form "YYYY/MM/DD").
  - *datetype* can be used to choose a type of date ("*mdat*": modification date, "*pdat*": publication date, "*edat*": Entrez date) to limit your search.
  - *reldate* must be an integer *n* and tells the *esearch()* method to return only the IDs of the records matching *datetype* within the last *n* days.
  - *mindate* and *maxdate* specify a date range that can be used to limit the search by the date type specified by *datetype*.

Entrez.esearch()

- To count matches, you can also use the *Entrez.egquery()* method, which returns the number of matches of the search term in each of the Entrez databases. It takes a single mandatory argument, which is the term to be searched:

```
>>> handle = Entrez.egquery(term = "PyCogent")
>>> record = Entrez.read(handle)
>>> for r in record["eGQueryResult"]:
...     print r["DbName"], r["Count"]
...
pubmed 3
pmc 92
mesh 0
books 0
...
```

Entrez.efetch()

```
from Bio import Entrez
from Bio import Medline

keyword = "PyCogent"

# search publications in PubMed
Entrez.email = "my_email@address.com"
handle = Entrez.esearch(db="pubmed", term=keyword)
record = Entrez.read(handle)
pmids = record['IdList']
print pmids

# retrieve Medline entries from PubMed
handle = Entrez.efetch(db="pubmed", id=pmids, rettype="medline", retmode="text")
medline_records = Medline.parse(handle)
records = list(medline_records)

n = 1
for record in records:
    if keyword in record["TI"]:
        print n, ')', record["TI"]
        n += 1
```

- You can use the *Entrez.efetch()* to download records from the NCBI server. *Entrez.fetch()* returns a handle that "contains" your records. You can read the raw data from the handle like you would read an open Python file (with a *for* loop) or parse them using specialized functions.

Entrez.efetch()

- Arguments of the *Entrez.efetch()* function:
  - The database from which the records are to be retrieved.
  - One ID or the list of IDs of the records to be downloaded.
  - Optional arguments:
    - *retmode* specifies the format of the record(s) retrieved (text, HTML, XML).
    - *rettype* specifies what types of records are shown. It depends on the database you are accessing.
      - For PubMed the *rettype* value can be, for example, *abstract*, *citation*, or *medline*.
      - For Uniprot you can set *rettype* to *fasta* to retrieve the sequence of a protein record.
    - *retmax* is the total number of records to be retrieved (up to a maximum of 10,000).

The *Medline* module

```
from Bio import Entrez
from Bio import Medline

keyword = "PyCogent"

# search publications in PubMed
Entrez.email = "my_email@address.com"
handle = Entrez.esearch(db="pubmed", term=keyword)
record = Entrez.read(handle)
pmids = record['IdList']
print pmids

# retrieve Medline entries from PubMed
handle = Entrez.efetch(db="pubmed", id=pmids, rettype="medline", retmode="text")
medline_records = Medline.parse(handle)
records = list(medline_records)

n = 1
for record in records:
    if keyword in record["TI"]:
        print n, ')', record["TI"]
        n += 1
```

- The *Medline* module provides the *Medline.parse()* function to parse the PubMed records. The result of this function can be conveniently converted into a list.

The *Medline* module

```
from Bio import Entrez
from Bio import Medline

keyword = "PyCogent"

# search publications in PubMed
Entrez.email = "my_email@address.com"
handle = Entrez.esearch(db="pubmed", term=keyword)
record = Entrez.read(handle)
pmids = record['IdList']
print pmids

# retrieve Medline entries from PubMed
handle = Entrez.efetch(db="pubmed", id=pmids, rettype="medline", retmode="text")
medline_records = Medline.parse(handle)
records = list(medline_records)

n = 1
for record in records:
    if keyword in record["TI"]:
        print n, ')', record["TI"]
        n += 1
```

- *Bio.Medline.Record* objects work like dictionaries. The most common keys are *TI* (Title), *PMID*, *PG* (pages), *AB* (Abstract), and *AU* (Authors).

Examples

- What are the available Entrez databases?

```
from Bio import Entrez

handle = Entrez.einfo()
info = Entrez.read(handle)
print info

raw_input('... press enter for a list of fields in PubMed')

handle = Entrez.einfo(db="pubmed")
record = Entrez.read(handle)
print record.keys()
print record['DbInfo']['Description']
print record['DbInfo']
```

Examples

- Searching PubMed with more than one term, combining keywords with *AND*/*OR*.

```
from Bio import Entrez

handle = Entrez.esearch(db="pubmed", term="PyCogent AND RNA")
record = Entrez.read(handle)
print record['IdList']

handle = Entrez.esearch(db="pubmed", term="PyCogent OR RNA")
record = Entrez.read(handle)
print record['Count']

handle = Entrez.esearch(db="pubmed", term="PyCogent AND 2008[Year]")
record = Entrez.read(handle)
print record['IdList']

handle = Entrez.esearch(db="pubmed", \
        term="C. elegans[Organism] AND 2008[Year] AND Mapk[Gene]")
record = Entrez.read(handle)
print record['Count']
```

Examples

- Retrieving and parsing nucleotide database entries in GenBank format.

```
from Bio import Entrez

# search sequences by a combination of keywords
handle = Entrez.esearch(db="nucleotide", term="Homo sapiens AND mRNA AND MapK")
records = Entrez.read(handle)
print records['Count']

top3_records = records['IdList'][0:3]
print top3_records

# retrieve the sequences by their GI numbers
gi_list = ','.join(top3_records)
print gi_list
handle = Entrez.efetch(db="nucleotide", id=gi_list, rettype="gb", retmode="xml")
records = Entrez.read(handle)
print len(records)

print records[0].keys()
print records[0]['GBSeq_organism']
```

Examples

- Searching for NCBI protein database entries by keywords.

```
from Bio import Entrez

# search IDs of protein sequences by keywords
handle = Entrez.esearch(db="protein", \
        term="Human AND cancer AND p21")
records = Entrez.read(handle)
print records['Count']
id_list = records['IdList'][0:3]

# retrieve sequences
id_list = ','.join(id_list)
print id_list
handle = Entrez.efetch(db="protein", \
         id=id_list, rettype="fasta", retmode="xml")
records = Entrez.read(handle)
rec = list(records)
print rec[0].keys()
print rec[0]['TSeq_defline']
```

Examples

- Retrieving SwissProt database entries and writing them to a file in FASTA format.

```
from Bio import ExPASy
from Bio import SeqIO

handle = ExPASy.get_sprot_raw("P04637")
seq_record = SeqIO.read(handle, "swiss")
print seq_record.id
print seq_record.description

out = open('myfile.fasta','w')
fasta = SeqIO.write(seq_record, out, "fasta")
out.close()
```

Summary

- Managing Your Biological Data with Python
  - Chapter 16. Creating Scientific Diagrams
  - Chapter 19. Working with Sequence Data
  - Chapter 20. Retrieving Data from Web Resources
  - Python codes in https://bitbucket.org/krother/python-for-biologists/src/