Biopython and why you should love it

Brad Chapman

6 March 2003

Talk Goals

- Talk briefly about open-source projects.
- Describe the Open-Bio projects and what exactly they do.
- Quick introduction to Python.
- Introduction to the things that the Biopython library provides.
- Show useful things you can do with Biopython.

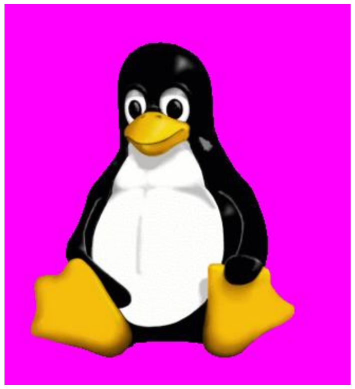
Open Source Programming – the philosophy

- Source code for programs is easily available to anyone who wants it
- Work together on programs and libraries in a distributed manner; can work with people from around the world
- Sections of code are written and maintained by many people
- Lots of people working on it means that lots of bugs can get fixed
- As a person – contribute sections of code on things you know (or want to learn) about; no barriers to working on it

Open Source Programming – the practical stuff

- Code is available freely to be looked at and re-used
- Anyone can see or work on the code while it is development
  - Concurrent Versions System (CVS)
  - Mailing Lists where people coordinate themselves

- Open Source Success Examples
  - Linux
  - Apache Web Server
  - Perl/Python/Ruby



Open-Bio Foundation

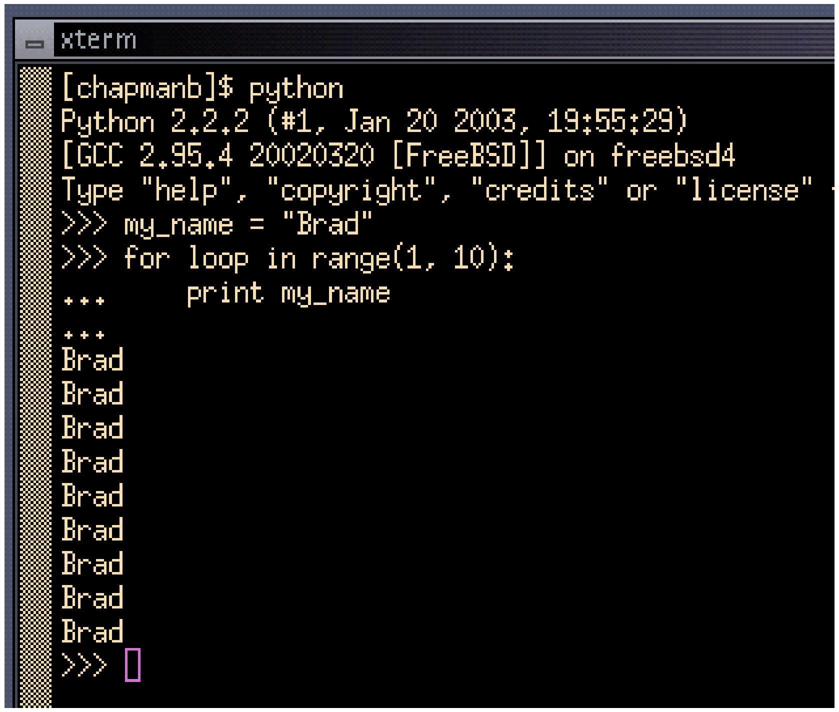
- Open-Bio is open-source for Bioinformatics
- http://www.open-bio.org/
- BioPerl, BioJava, Biopython, BioRuby, BioWhateverYouWant
- What does it do?
  - Managing the Bio* projects mentioned above (Servers, Webpages, FTP sites, CVS, all of the ugly technical details)
  - Hold annual meetings – Bioinformatics Open Source Conference (BOSC) tied in with ISMB
  - Organizing "hackathon" events for developers to get together and work

Now what is Python?

- Interpreted, interactive, object-oriented programming language
- http://www.python.org/
- Portable – runs just about anywhere
- Clear syntax – can be easy to learn

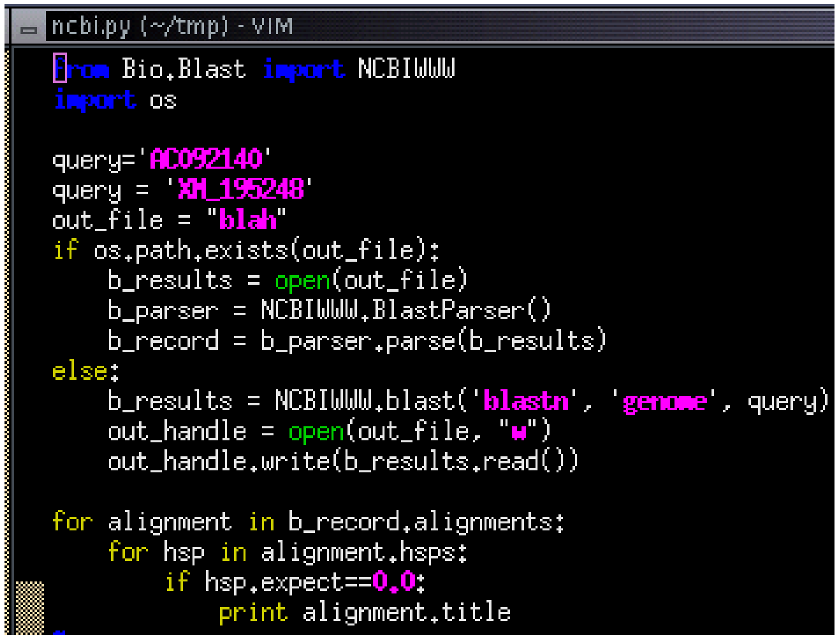
Interacting with Python

The Python Interpreter



```
xterm
[chapmanb]$ python
Python 2.2.2 (#1, Jan 20 2003, 19:55:29)
[GCC 2.95.4 20020320 [FreeBSD]] on freebsd4
Type "help", "copyright", "credits" or "license"
>>> my_name = "Brad"
>>> for loop in range(1, 10):
...     print my_name
...
Brad
Brad
Brad
Brad
Brad
Brad
Brad
Brad
Brad
>>>
```

Writing a script in a file with a text editor



```
ncbi.py (~/tmp) - VIM
from Bio.Blast import NCBIWWW
import os

query='AC092140'
query = 'XM_195248'
out_file = "blah"
if os.path.exists(out_file):
    b_results = open(out_file)
    b_parser = NCBIWWW.BlastParser()
    b_record = b_parser.parse(b_results)
else:
    b_results = NCBIWWW.blast('blastn', 'genome', query)
    out_handle = open(out_file, "w")
    out_handle.write(b_results.read())

for alignment in b_record.alignments:
    for hsp in alignment.hsps:
        if hsp.expect==0.0:
            print alignment.title
```

Some Advantages of Python for Biologists

- Easy to write quick programs
- Very easy to interact with files and the operating system
- Nice libraries for dealing with web pages, databases, other useful things
- Many useful libraries for Scientists
  - Numerical Python – fast, multidimensional arrays; math, math, math (`http://www.pfdubois.com/numpy/`)
  - Scientific Python – statistics, python MPI interfaces (`http://starship.python.net/~hinsen/ScientificPython/`)
  - Of course, Biopython...

Quick Python – Python Objects

- Python is an object-oriented programming language so you need to at least understand a little about objects to understand Biopython
- An object is our own personalized data structure
  - A single object models something you want to work with
  - It has its own personal variables
  - It has it's own functions
  - Self-contained way of representing something
- In most cases objects with be the major way of representing and containing programming

Quick Python – Something to model

FASTA record

```
>gi|1348917|gb|G26685|G26685 human STS STS_D11734.
CGGAGCCAGCGAGCATATGCTGCATGAGGACCTTTCTATCTTACATTATGGCTGGGAATCTTACTCTTTC
ATCTGATACCTTGTTCAGATTTCAAAATAGTTGTAGCCTTATCCTGGTTTTACAGATGTGAAACTTTCAA
GAGATTTACTGACTTTCCTAGAATAGTTTCTCTACTGGAAACCTGATGCTTTTATAAGCCATTGTGATTA
GGATGACTGTTACAGGCTTAGCTTTGTGTGAAANCCAGTCACCTTTCTCCTAGGTAATGAGTAGTGCTGT
TCATATTACTNTAAGTTCTATAGCATACTTGCNATCCTTTANCCATGCTTATCATANGTACCATTTGAGG
AATTGNTTTGCCCTTTTGGGTTTNTTNTTGGTAAANNNTTCCCGGGTGGGGGNGGTNNNGAAA
```

We'll show the real Biopython way of dealing with FASTA later, but use this as an example of something we want to represent as an object

Quick Python – An Example Object

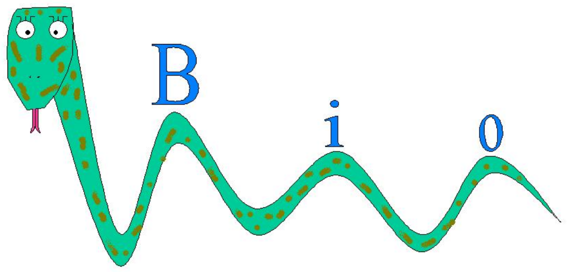
```
class FastaRecord:
    def __init__(self):
        self.title = ’’
        self.sequence = ’’

    def write_to_fasta(self):
        return ">" + self.title + "\n" + self.sequence + "\n"


>>> rec = FastaRecord()
>>> rec.title = "my sequence"
>>> rec.sequence = "GATGAC"
>>> print rec.write_to_fasta()
>my sequence
GATGAC
```

Open-Bio + Python = Biopython

- Official blurb: international association of developers of freely available Python tools for computational molecular biology; established in 1999.
- http://biopython.org
- Library of functionality for dealing with common problems biologists programming in python might face.



What can Biopython do?

Parse Blast results (standalone and web); run biology related programs (blastall, clustalw, EMBOSS); deal with FASTA formatted files; parse GenBank files; parse PubMed, Medline and work with on-line resource; parse Expasy, SCOP, Rebase, UniGene, SwissProt; deal with Sequences; data classification (k Nearest Neighbors, Bayes, SVMs); Aligning sequences; CORBA interaction with Bioperl and BioJava; SQL database storage through BioSQL; Neural Networks; Genetic Algorithms; Hidden Markov Models; creating pretty PDF files for posters; format flatfiles with random access to entries; structural biology PDB, FSSP; create specialized substitution matrices; okay, my head hurts. . .

Getting started with Biopython – a Sequence

- As mentioned before, most things in Biopython are objects – sequences are no exception

- Represented as a `Seq` object in Biopython; each `Seq` object has two important attributes:

  data – the actual sequence string (`GATCAC`)
  alphabet – another object describing what the characters making up the string "mean"

- The alphabet addition to "just a string" has two advantages:
  - Identifies the information in the string
  - Provides a mean for error checking

Creating a Sequence

```
>>> from Bio.Alphabet import IUPAC
>>> my_alpha = IUPAC.unambiguous_dna
>>> from Bio.Seq import Seq
>>> my_seq = Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', my_alpha)
>>> print my_seq
Seq('GATCGATGGGCCTATATAGGATCGAAAATCGC', IUPACUnambiguousDNA())
```

Seq objects act like python sequences

```
>>> my_seq[4:12]
Seq('GATGGGCC', IUPACUnambiguousDNA())
>>> len(my_seq)
32
```

Doing something – Transcription/Translation

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> my_seq = Seq('GCCATTGTAATGGGCCGCTGAAAGGGTGCCCGA', IUPAC.unambiguous_dna)

>>> from Bio import Transcribe
>>> transcriber = Transcribe.unambiguous_transcriber
>>> transcriber.transcribe(my_seq)
Seq('GCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGA', IUPACUnambiguousRNA())

>>> from Bio import Translate
>>> translator = Translate.unambiguous_dna_by_name["Standard"]
>>> translator.translate(my_seq)
Seq('AIVMGR*KGAR', HasStopCodon(IUPACProtein(), '*'))
```

What about annotations and features

- Biopython has a `SeqRecord` class which holds a sequence plus all of the information about it (Features, Annotations)
- Can store all of the information from a GenBank style record

```
LOCUS       ATCOR66M      513 bp    mRNA            PLN 02-MAR-1992
DEFINITION  A.thaliana cor6.6 mRNA.
ACCESSION   X55053
[...]
FEATURES             Location/Qualifiers
     source          1..513
                        /organism="Arabidopsis thaliana"
                        /strain="Columbia"
                        /db_xref="taxon:3702"
```

The Biopython Representation

```
>>> seq_record.seq[:20]
Seq('AACAAAACACACATCAAAAA', IUPACAmbiguousDNA())
>>> seq_record.id
'X55053.1'
>>> seq_record.description
'A.thaliana cor6.6 mRNA.'
>>> seq_record.name
'ATCOR66M'
>>> print seq_record.features[0]
type: source
location: (0..513)
qualifiers:
        Key: db_xref, Value: ['taxon:3702']
        Key: organism, Value: ['Arabidopsis thaliana']
        Key: strain, Value: ['Columbia']
```

Let’s Actually do something – Fasta Files

- FASTA formatted files are one of the most commonly used file formats since they are very simple
- Some common problems:
  - Reading sequences from FASTA file (might want to translate them, rewrite only a subset of them,. . .)
  - Writing sequences to FASTA format – preparing some sequences you have in a different format for input into a program
- Lots of potential workarounds (Excel Macros, cutting and pasting and editing in Word) but nice to be able to deal with these files in an automated way when you have tons of them

Reading a FASTA file – Setup

Standard python – open the file

```
file_handle = open(your_fasta_file, "r")
```

Biopython – create a parser and iterator

- Parser – Takes a FASTA record and returns a Biopython object
- Iterator – Loop over FASTA records one at a time

```
from Bio import Fasta
parser = Fasta.RecordParser()
iterator = Fasta.Iterator(file_handle, parser)
```

Reading a FASTA file – using the Iterator

Loop over the file one record at a time

```
while 1:
    record = iterator.next()
    if record is None:
        break
    # do something with each record
    print record.sequence
```

- Each `record` returned in a python object.
- In this case it is a simple object directly representing FASTA
- Can have different parsers that return different objects (`Seq` objects)

Writing a FASTA file – the Record object

- The Biopython FASTA Record object is very similar to the example FASTA object we looked at previously

- It has a bit fancier internals (for nice output of FASTA), but has the same two basic attributes:

  title – The title line of the FASTA sequence
  sequence – The actual sequence itself

```
from Bio import Fasta

rec = Fasta.Record()
rec.title = "My Sequence"
rec.sequence = "GATCGATC"
```

Writing a FASTA file – doing it

The string representation of a FASTA record class is pretty formatted FASTA

```
>>> print rec
>gi|1348912|gb|G26680|G26680 human STS
CGGACCAGACGGACACAGGGAGAAGCTAGTTTCTTTCATGTGATTGANATNATGACTCTA
AGCTCAGTCACAGGAGAGANCACAGAAAGTCTTAGGATCATGANCTCTGAAAAAAAGAGA
AACCTTATCTTTNCTTTGTGGTTCCTTTAAACACACTCACACACACTTGGTCAGAGATGC
```

We can write Records to a file very simply

```
out = open(out_file, "w")
out.write(str(rec) + "\n")
out.close()
```

Detailed Biopython example: batch primer design

- Simplified from a real example where we needed to generate primers for several hundred sequences
- Our goals:
  - Start with a file full of sequences
  - Design primers for each sequence with specific criterion (have to span a central region)
  - Write the files to a format that can be loaded into Excel
- Accomplished in python with the help of the Biopython libraries

Our starting sequences

- Begin with files in Fasta format, a standard sequence format

```
>CL031541.94_2268.amyltp
CGCGGCCGCSGACCTYGCCGCAGCCTCCCCTTCMATCCTCCTCCCGCTCC
TCCTACGCGACGCCGGTGACCGTGATGAGGCCGTCGCCGCYTCCGCGCTC
CCTCAAGGCNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
AAGTGCCTTGGCTTCACGCTCTACCATCCGGTCCTCGTCTCCACCGTCTC
AGGTAGAATGGCTCCCCTGTATCCACAACTCGTCAGTGAAATGTGCAGTT
```

- Biopython has a module (`Bio.Fasta`) which can read these Fasta formatted files.

Our primer design program

- Primer3, from the Whitehead Institute – `http://www-genome.wi.mit.edu/genome_software/other/primer3.html`
- EMBOSS, a free set of bioinformatics programs, has an interface (eprimer3) to make the program easier to use – `http://www.emboss.org`
- Biopython has code to work with eprimer3
  - `Primer3Commandline` – run the program
  - `Primer3Parser` – parse the output from the program

Program Step 1 – read the Fasta File

Load the biopython module that deals with Fasta

```
from Bio import Fasta
```

Open the Fasta file from python

```
open_fasta = open(fasta_file, "r")
```

Initialize an Iterator that lets us step through the file

```
parser = Fasta.RecordParser()
iterator = Fasta.Iterator(open_fasta, parser)
```

Program Step 2 – Read through the file

We want to go through the file one record at a time.

Generate a loop and use the iterator to get the next record.

We stop if we reach the end of the file (get a record that is None).

```
while 1:
    cur_record = iterator.next()
    if not(cur_record):
        break
```

Program Step 3 – Set up the primer3 program

We want to create a run of the primer3 program with our parameters

```
from Bio.Emboss.Applications import Primer3Commandline

primer_cl = Primer3Commandline()
primer_cl.set_parameter("-sequence", "in.pr3")
primer_cl.set_parameter("-outfile", "out.pr3")
primer_cl.set_parameter("-productsizerange", "350,10000")
primer_cl.set_parameter("-target", "%s,%s" % (start, end))
```

start and end are the middle region we want to design primers around

Program Step 4 – Run the primer3 program

Biopython has a single way to run a program and get the output.

Python interacts readily with the operating system, making it easy to run other programs from it.

```
from Bio.Application import generic_run

result, messages, errors = generic_run(primer_cl)
```

The Primer3 output file

Primer3 generates an output file that looks like:

```
# PRIMER3 RESULTS FOR CLB11512.1_789.buhui

#             Start Len  Tm     GC%   Sequence

1 PRODUCT SIZE: 227
FORWARD PRIMER  728 20  59.91  50.00  TTCACCTACTGCAAACGCAC

REVERSE PRIMER  935 20  59.57  50.00  TTGGTACGTTGTCCATCTCG
```

A ton of these files are not easy to deal with.

Program Step 5 – parse the output file

We use the Biopython parser to parse these files.

```
open_outfile = open("out.pr3", "r")

from Bio.Emboss.Primer import Primer3Parser

parser = Primer3Parser()
primer_record = parser.parse(open_outfile)
```

The result is that we get the information into a python ready format that we can readily output.

Program Step 6 – Write the output

We want to write the output to an Excel ready format

We write the forward and reverse sequences along with the sequence name to a comma separated value file.

```
open_excelfile = open(excel_file, "w")

primer = primer_record.primers[0]

open_excelfile.write("%s,%s,%s" % (
  sequence_name, primer.forward_seq, primer.reverse_seq))
```

The result is a file full of primers that you can then deal with.

Wrapup

- The Biopython toolkit contains tons of useful functionality
  - Libraries you don't have to write
  - Documentation for how to use things
  - Other people to work with/fix bugs
- There are similar libraries for other programming languages you might like to use (BioPerl, BioJava, BioRuby)
- These toolkits are not perfect, but the initial investment in learning one is worth your time when you need additional functionality