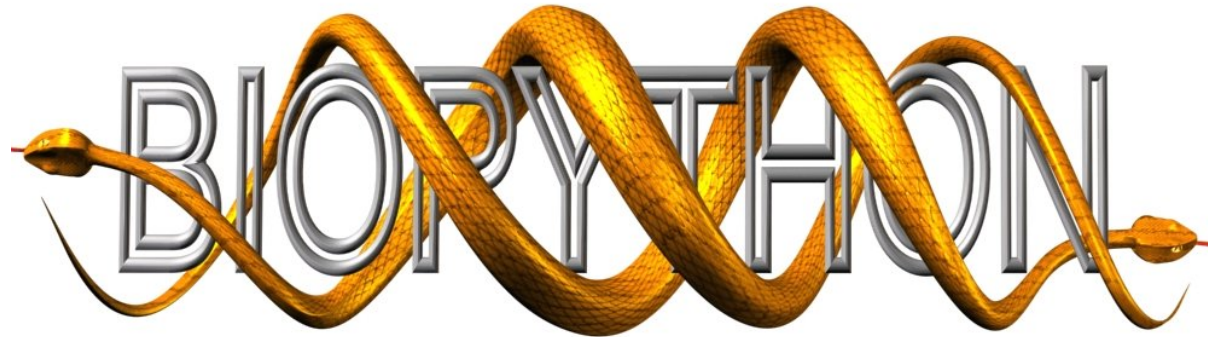


The Biopython Project: Philosophy, functionality and facts

Brad Chapman

11 March 2004



Biopython – one minute overview

- The Biopython Project is an international association of developers of freely available Python tools for computational molecular biology.
 - History of Biopython
 - Organization and makeup of the Biopython community
 - What Biopython contains and why you'd want to use it
 - Detailed examples of Biopython, for use and development
- <http://biopython.org>



Who Am I?

- Molecular biologist who drifted to programming during graduate school
- Graduating in August of this year
- Starting programming in Python in 1999
- Starting doing Biopython work in 2000
- Coordinating the project since November of 2003



Biopython history

- Biopython began in August of 1999
 - The brainchild of Jeff Chang and Andrew Dalke
 - Significant push from Ewan Birney, of BioPerl and Ensembl fame
- February 2000 – started having CVS and project essentials (stop talking, start coding)
- July 2000 – First release
- March 2001 – First 1.00-type “semi-complete” release
- December 2002 – First “semi-stable” release



Biopython and the Open-Bio Foundation

- Open Bioinformatics Foundation
 - Non profit, volunteer run organization focused on supporting open source programming in bioinformatics.
 - Grew out of initial Bio-project – BioPerl
 - <http://www.open-bio.org/>
- Main things Open-Bio does:
 - Support annual Bioinformatics Open Source (BOSC) conferences
 - Organize “hackathon” events
 - Obtain and support hardware for projects



Biopython and other Bio* projects

- Basically a sibling project with BioPerl, BioJava and BioRuby
- Work together, both informally and during organized “hackathon” events
 - BioCORBA (now mostly defunct)
 - BioSQL – standard set of SQL for storing sequences plus annotations
 - File indexing – Flat-files (FASTA, GenBank, Swissprot. . .)
 - Retrieval from web databases
 - Managing access to biological resources



Biopython developer community

Founders – Andrew and Jeff; building framework of Biopython

Coordinator – currently me;

- Benevolent dictator style organization, like Python itself (except I'm not much of a dictator)
- Development handles switches in leadership roles

Module contributors

- Requiring specialized knowledge of an area
- Create and maintain; code, tests, documentation
- Recent examples – Clustering, Structural Bioinformatics, NMR data, Sequencing related files, Wise alignments



How many total users are there?

Web site views – Since May of 2003

- 15,020 unique IP views, 52,580 total
- About 51 unique views and 178 total views a day

Release downloads

Release	Date	Unique Downloads	Per Day
1.20	July 27, 2003	67	67
1.21	July 28, 2003	759	10.4
1.22	October 9, 2003	121	13.5
1.23	October 18, 2003	1114	9.2
1.24	February 16, 2004	347	15.3



Project organization – philosophy

Contributing

- Fairly relaxed attitude; whoever codes it wins
- Maintain some “standard” ways to develop parsers and iterators, as well as coding conventions; to ease use and bug fixing
- Just want generally good coding practices in modules

License

- Short and open as possible
- Basically, keep copyright (don't pretend it's yours) and we aren't responsible if it messes something up (cover ourselves)



Project organization – what we have

CVS

- Write access: 12 active developers with access
- Anonymous CVS access: <http://cvs.biopython.org>

Mailing Lists

- Active discussion list
- Development list (patches, attachments, arguing over code details. . .)

Documentation

- Long tutorial-style doc
- Docs for specific parts (installation, BioSQL, Cluster)
- On-line courses with Biopython content



Project organization – the code

Bugs

- On-line Bugzilla tracker plus quick-fixes from mailing list
- Bug assignment normally done by module owner or others with time

Releases

- No standard schedule
- Try to work from a stable CVS base to simplify the release cycle and encourage people to work directly from CVS

Tests

- Testing framework based on unittest from standard library



Required and useful libraries

Try to keep pre-install requirements to a minimum

Python – works with 2.2 and higher

mxTextTools – Fast text manipulation library; underlies parser framework

Numerical Python – fast array manipulation; used for Cluster code, PDB, NaiveBayes and Markov Model code

Database modules – either MySQLdb or pycopg; relational database storage and access



What does Biopython provide? – Sequence-based

Sequences themselves – Sequence classes, manipulations (translation, codon usage, melting temperature. . .)

Dealing with sequence formats – FASTA, GenBank, Swissprot, GFF

Parsers for output from bioinformatics programs – BLAST, Clustalw

Access to on-line resources – EUtils at NCBI, ExPASy, SCOP

Running bioinformatics programs – BLAST, Clustalw, EMBOSS

Alignments – Clustalw, Wise, NBRF/PIR



What does Biopython provide? – Others

Microarray data – Clustering, reading and writing Tree-View type files

Structure data – PDB parsing, representation and analysis

Structure calculation – NMR, predicting NOE coordinates, nmrview style files

Sequence data – Ace and PHD files

Journal data – Medline



What does Biopython provide? – Tools

Relational databases – BioSQL: standard amongst all open-bio projects
SQL for storing sequences plus annotations

Parser development

- Martel: parser generator using mxTextTools
- ParserSupport: python-only based parser development using a Event/Consumer parsing model

File indexing – Mindy: flat-file indexing supported by all open-bio projects
(flat-file only and BerkeleyDB)



Biopython usage example

Goal – Retrieve and store in a relational database GenBank records identified by BLAST searches of a given FASTA sequence

1. BLAST sequence against Swissprot database
2. Parse BLAST results, get hit IDs
3. Retrieve GenBank sequences from NCBI
4. Store in a relational database



Standard Biopython parser setup

Parsers

RecordParser – Parse a file into a format specific record object

FeatureParser – Parse into a generic class of sequence plus features

Iterator – Go over a file one record at a time

- Use any of the parser objects
- Also return raw text
- Works like both traditional and Python 2.2 iterators



FASTA sequence

```
>gi|42821112|ref|NP_006866.2| pim-2 oncogene;  
MLTKPLQGPPAPPGTPTPPPGGKDREAFAEYRLGPLLKGGFGTVFAGHRLTDRQLQVAIKVIPRNRVLG  
WSPLSDSVTCPLEVALLWKVGAGGGHPGVIRLLDWFETQEGFMLVLERPLPAQDLFDYITEKGPLGEGPS
```

```
input_handle = open("input.fasta")
```

```
from Bio import Fasta  
parser = Fasta.RecordParser()  
iterator = Fasta.Iterator(input_handle, parser)  
rec = iterator.next()
```



BLASTing FASTA

```
from Bio.Blast import NCBIWWW
results_handle = NCBIWWW.blast('blastp', 'swissprot',
                                str(rec), expect=1e-10)
```

- Do a remote BLAST against NCBI
- Calling `str` on a Fasta Record returns a nicely formatted Fasta sequence, for writing to files or printing
- Parser can be used on HTML results to retrieve IDs from the description lines



Parsing BLAST results

<HTML>

BLASTP 2.2.8 [Jan-05-2004]

gi|20139243|sp|Q9P1W9|PIM2_HUMAN Serine/threonine-protein k...

614 e-176

```
blast_parser = NCBIWWW.BlastParser()
record = blast_parser.parse(results_handle)
```

```
swissprot_ids = []
for description in record.descriptions:
    swissprot_id = description.title.split("|")[1]
    swissprot_ids.append(swissprot_id)
```



Retrieving GenBank from NCBI

- Interface to the EUtils interface for batch retrieval of Entrez queries
- http://www.ncbi.nlm.nih.gov/entrez/query/static/eutils_help.html

```
from Bio.EUtils import DBIds
from Bio.EUtils import DBIdsClient
```

```
db_ids = DBIds("protein", swissprot_ids)
eutils_client = DBIdsClient.from_dbids(db_ids)
genbank_handle = eutils_client.efetch(retmode="text", rettype="gp")
```



Standard sequence objects

SeqRecord – main interface for sequences plus features

Seq – a sequence object, acts like a string

data – the sequence itself (a string)

alphabet – a class describing the type of sequence and which letters are allowed in it

Id information – id, name, description for the sequence

Annotations – annotations about the whole sequence (organism, references. . .)

Features – information about particular part of the sequence (exons, binding sites. . .)



Parsing GenBank into sequence objects

```
LOCUS      Q9P1W9                311 aa                linear      PRI 15-SEP-2003
DEFINITION Serine/threonine-protein kinase Pim-2 (Pim-2h).
ACCESSION  Q9P1W9
VERSION    Q9P1W9  GI:20139243
DBSOURCE   swissprot: locus PIM2_HUMAN, accession Q9P1W9;
```

```
from Bio import GenBank
parser = GenBank.FeatureParser()
iterator = GenBank.Iterator(genbank_handle, parser)
```



BioSQL

Standard set of SQL tables for storing sequences plus features and annotations

- Supported by all of the Bio projects (BioPerl, BioJava, BioRuby)
- Works on MySQL and Postgresql in Biopython
- Basic use case is to store something like a GenBank record, but extensible for individualized data
- Biopython interface which makes reading sequences exactly like dealing with a SeqRecord object
- Can also use raw SQL queries



Storing in a SQL database

```
> mysqladmin -u chapmanb -p create oncogene  
> mysql -u chapmanb -p oncogene < biosqldb-mysql.sql
```

```
from BioSQL import BioSeqDatabase  
server = BioSeqDatabase.open_database(driver = "MySQLdb", user = "chapmanb",  
                                     passwd = "biopython", host = "localhost", db = "oncogene")  
db = server.new_database("swissprot_hits")  
  
db.load(iterator)
```



Final results

```
> mysql -u chapmanb -p oncogene
mysql> select accession, identifier, division, description from bioentry;
+-----+-----+-----+-----+
| accession | identifier | division | description |
+-----+-----+-----+-----+
| Q9P1W9    | 20139243  | PRI      | Serine/threonine-protein kinase Pim-2. |
| Q62070    | 20138972  | ROD      | Serine/threonine-protein kinase Pim-2. |
| Q86V86    | 38502964  | PRI      | Serine/threonine-protein kinase pim-3. |
mysql> select length, seq from biosequence;
+-----+-----+
| length | seq |
+-----+-----+
| 311    | MLTKPLQGPPAPPGTPTPPPGGKDREAFAEYRLGP |
| 370    | MARATNLNAAPSAGASGPPDSLPLAPPSPGSPAA |
| 326    | MLLSKFGSLAHLGPGGVDHLPVKILQPAKADKESF |
```



Useful Biopython development tools – parsing

Biopython also includes tools, in addition to ready to go code

- Martel – provides a basis for developing parsers
- Uses regular expression format definitions to make parsers
- Parses into XML which can be dealt with through a SAX interface
- Fast, uses mxTextTools C code for the work
- Provides standardized indexing of files, through Mindy



Building a parser with Martel – example

Goal – Parse output from BLAT; sequence search tool from Jim Kent

1. Build a Martel Regular Expression grammar to deal with BLAT psl-style files
2. Demonstrate how this file can be used to build a parser
3. Show XML based output of the parser
4. Describe utilities to simplify dealing with the XML output



BLAT simplified input

psLayout version 3

Q	Q	Q	T	T	T
name	start	end	name	start	end
sequence_10	0	1775	sequence_12	10	1675
sequence_10	840	1244	sequence_13	940	1344

- Tab delimited hits – not the hardest to parse but nice for an example
- Many columns removed from normal BLAT psl output



Understanding Martel – basics

Provides a regular expressions on steroids interface for parsing

- Classes to match particular items, some examples:

Str – match an exact string

Spaces – match whitespace (not newlines) – `[\\t\\v\\f\\r]+`

Integer – digits with optional + or - sign – `[+-]?\\d+`

AnyEol – Match any type of newline: `\\n`, `\\r` or `\\r\\n`

ToEol – Match the rest of a line including the newline

- Classes to repeat matches:

Rep – Repeat 0 or more times

Rep1 – Repeat 1 or more times

RepN – Specify N times to repeat



Martel grammar – header

```
psLayout version 3
```

```
Q           Q       Q       T           T       T  
name       start   end   name           start   end
```

```
import Martel
```

```
title = Martel.Str("psLayout version") + Martel.Spaces() + \  
        Martel.Integer("blat_version") + Martel.Rep(Martel.AnyEol())
```

```
header_lines = Martel.RepN(Martel.ToEol(), 3)
```



Understand Martel – more information

- Individual items of interest in a file are designated with names
 - Martel emits XML
 - These names will be the XML tags surrounding the information
- Can group different items together under a mega-group – like nested XML tags

```
<hit_line>  
  <query_name>The Name</query_name>  
</hit_line>
```

- Utilities to parse files with standard separators (tabs)



Martel grammar – hit lines

```
sequence_10 0      1775 sequence_12 10      1675
```

```
hit_line = Martel.Group("hit_line",  
    # read the query information  
    Martel.UntilSep("query_name", "\t") + Martel.Str("\t") +  
    Martel.Integer("query_start") + Martel.ToSep(sep = "\t") +  
    Martel.Integer("query_end") + Martel.ToSep(sep = "\t") +  
  
    # read the target information  
    Martel.UntilSep("target_name", "\t") + Martel.Str("\t") +  
    Martel.Integer("target_start") + Martel.ToSep(sep = "\t") +  
    Martel.Integer("target_end"))
```



Using Martel grammars

- All of the regular expressions combine together to create larger regular expressions which handle records in a file and the whole file
- We use standard tags (Std) for things which occur regularly in files
 - records
 - identifiers, descriptions, database cross references
 - sequences, alphabets
 - features
 - homology searches – headers, application names



Martel grammar – putting it all together

```
from Martel import RecordReader
from Bio import Std

record = Std.record(hit_line + Martel.AnyEol())

format = Martel.HeaderFooter("blat", {"format" : "blat"},
                             title + header_lines, RecordReader.CountLines, (5,),
                             record, RecordReader.CountLines, (1,),
                             None, None, None)
```

- Can build formats in other ways
 - StartsWith** – Fasta Records >
 - EndsWith** – GenBank Records //



Using the parser – creating an iterator

```
import blat
```

```
input_handle = open("blat_ex.psl")
```

```
iterator_builder = blat.format.make_iterator("record")
```

- Standard Python 2.2 style iterator
- Since Martel emits XML – can connect different handlers to the iterator
 - XMLGenerator – just print out the XML
 - LAX – convert the XML to a dictionary
 - Custom handlers



Using the parser – XML output

```
from xml.sax import saxutils
handler = saxutils.XMLGenerator()
iterator = iterator_builder.iterateFile(input_handle, handler)
iterator.next()
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<record xmlns:bioformat="http://biopython.org/bioformat">
  <hit_line>
    <query_name>sequence_10</query_name>
    <query_start>0</query_start><query_end>1775</query_end>
    <target_name>sequence_12</target_name>
    <target_start>10</target_start><target_end>1675</target_end>
  </hit_line>
</record>
```



Using the parser – LAX handler

```
from Martel.LAX import LAX
handler = LAX(fields = ["query_name", "query_start", "query_end",
                      "target_name", "target_start", "target_end"])
iterator = iterator_builder.iterateFile(input_handle, handler)
for result in iterator:
    print result
```

```
{'target_name': ['sequence_12'], 'query_start': ['0'],
 'query_end': ['1775'], 'target_start': ['10'],
 'query_name': ['sequence_10'], 'target_end': ['1675']}
```

```
{'target_name': ['sequence_13'], 'query_start': ['840'],
 'query_end': ['1244'], 'target_start': ['940'],
 'query_name': ['sequence_10'], 'target_end': ['1344']}
```



Translating into a Biopython-style parser

- For something simpler like FASTA – use the LAX handler and write wrapper classes for Iterators and Parsers around it
- More complicated cases like GenBank
 - Derive from standard `xml.sax.handler` class
 - Helpful class in Biopython to turn Martel XML tags into Events (EventGenerator) so you can use an Event/Consumer framework for organizing the parser

Ease of developing parsers without starting from scratch



Future Biopython goals

- More modules; always more formats and programs to support
- Improved documentation; cookbook-style documentation
- Bug fixing – keeping up with changes to bioinformatics formats
- New/developmental items:
 - Standardized mechanism for running applications
 - Standardized access to bioinformatics resources
 - More reliance on Martel for parsing – conversions between formats

