Biopython: Python tools for computation biology

Brad Chapman and Jeff Chang

August 2000

Contents

1 Abstract

The Biopython project was formed in August 1999 as a collaboration to collect and produce open source bioinformatics tools written in Python, an object-oriented scripting language. It is modeled on the highly successful Bioperl project, but has the goal of making libraries available for people doing computations in Python. The philosophy of all the Bio* projects is that part of bioinformaticists' work involves software development. In order to prevent repeated efforts we believe that the field can be advanced more quickly if libraries that perform common programming functions were available. Thus, we hope to create a central source for high-quality bioinformatics tools that researchers can use.

As an open source project, Biopython can be downloaded for free from the web site at `http://www.biopython.org`. Biopython libraries are currently under heavy development. This paper describes the

current state of available Biopython tools, shows examples of their use in common bioinformatics problems, and describes plans for future development.

2 Introduction

Development of software tools is one of the most time consuming aspects of the work done by a bioinformaticist. A successful solution to this problem has been the establishment of common repositories of interworking open-source libraries. This solution not only saves development time for researchers, but also leads to more robust, well-tested code due to the contributions of multiple developers on the project. The ideas behind open source collaborative software have been explored through a series of essays by Eric Raymond (`http://www.tuxedo.org/~esr/writings/`). One example where the open source methodology has been successfully applied is the Bioperl project (`http://www.bioperl.org`). Similarly, Biopython seeks to develop and collect biologically oriented code written in python (`http://www.python.org`).

Python is an object-oriented scripting language that is well suited for processing text files and automating common tasks. Because it was designed from the ground up with an object-oriented framework, it also scales well and can be utilized for large projects. Python is portable to multiple platforms including multiple UNIX variants, Windows, and Macintosh. The standard library comes with many high-level data structures such as dictionaries, and contains numerous built in modules to accomplish tasks from parsing with regular expressions to implementing a http server.

The native python libraries interface well with C, so after initial development in python, computationally expensive operations can be recoded in C to increase program speed. In addition, jpython (`http://www.jpython.org`), an implementation of python in pure java, allows python to freely interact with java libraries. The options allow rapid development in python, followed by deployment in other languages, if desired.

A library of common code for biological analysis is essential to allow bioinformaticists to take advantage of all of the benefits of programming in python. This paper describes work towards development of such a library.

3 Parsers for Biological Data

3.1 Design Goals

The most fundamental need of a bioinformaticist is the ability to import biological data into a form usable by computer programs. Thus, much of the initial development of Biopython has been focused on writing code that can retrieve data from common biological databases and parse them into a python data structure.

Designing parsers for bioinformatics file formats is particularly difficult because of the frequency at which the data formats change. This is partially because of inadequate curation of the structure of the data, and also because of changes in the contents of the database. Biopython addresses these difficulties through the use of standard event-oriented parser design.

Then event-oriented nature of biopython parsers are similar to that utilized by the SAX (Simple API for XML) parser interface, which is used for parsing XML data files. They are different then SAX in that they are line-oriented; since nearly all biological data formats use lines as meaningful delimiters biological parsers can be built based on the assumption that line breaks are meaningful. A parser involves two components: a Scanner, whose job is to recognize and identify lines that contain meaningful information, and a Consumer, which extracts the information from the line.

The Scanner does most of the difficult work in dealing with the parsing. It is required to move through a file and send out "events" whenever an item of interest in encountered in the file. These events are the key pieces of data in the file that a user would be interested in extracting. For instance, lets imagine we were parsing a FASTA formatted file with the following sequence info (cut so the lines fit nicely):

```
>gi|8980811|gb|AF267980.1|AF267980 Stenocactus crispatus
AAAGAAAAATATACATTAAAAGAAGGGGATGCGGG
...
```

As the scanner moved through this file, it would fire off 4 different types of events. Upon reaching a new sequence begin_sequence event would be sent. This would be followed by a title event upon reaching the information about the sequence and a sequence event for every line of sequence information. Finally, everything would wrap up with an end_sequence when we have no more sequence data in the entry.

Creating and emanating these events is interesting, but is not very useful unless we get some kind of information from the events, which is where the Consumer component comes in. The consumer will register itself with a scanner and let it know that it wants to know about all events that occur while scanning through a file. Then, it will implement functions which deal with the events that it is interested in.

To go back to our FASTA example, a consumer might just be interested in counting the number of sequences in a file. So, it would implement a function begin_sequence that increments a counter by one every time that event occurs. By simple receiving the information it is interested in from the Scanner, the consumer processes and deals with the files according to the programmer's needs.

By decoupling scanners from consumers, developers can choose different consumers depending on their information or performance needs, while maintaining the same scanner. It is possible to develop multiple specialized consumer-handlers using the same scanner framework. These parsers can deal with a small subsection of the data relatively easily. For example, it may be desirable to have a parser that only extracts the sequence information from a Genbank file, without having to worry about the rest of the information. This would save time by not processing unnecessary information, and save memory by not storing it.

3.2 Usage Examples

3.2.1 Usage Scenario

To take a look at the parsers in action, we'll look at some examples based around a common theme, to make things a little more exciting. Let's suddenly become really interested in Taxol, a novel anti-cancer drug (A good quick introduction to Taxol can be found at http://www.bris.ac.uk/Depts/Chemistry/MOTM/taxol/taxol.htm) and use this newfound interest to frame our work.

3.2.2 Extracting information from a FASTA file

To start our search for Taxol information, we first head to NCBI to do an Entrez search over the Genbank nucleotide databases (http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=Nucleotide). Just searching for the keyword 'taxol' gives 22 results, so let's say we want to parse through these results and extract id numbers from those that have to do with humans. To do this, we save the search results as a FASTA file, and then proceed to parse this.

Based on the scanner-consumer discussion above, what we need to do is implement a consumer which will react every time we reach a title in the FASTA file, and check to see if the title mentions anything to do with humans. The following consumer does this job:

```
import string
from Bio.ParserSupport import AbstractConsumer

class TitleSearchConsumer(AbstractConsumer):
    def title(self, line):
        # see if the title contains a reference to humans}
        location = string.find(line, "Homo sapiens")}

        if location != -1:
            # split the string to give us the accession number}
            result = string.split(line, '|')}
            print 'Accession:', result[3]}
```

The Consumer inherits from a base consumer class which ignores any sections that we are not interested in (like sequences). Now that we've got a consumer, we need to start up a scanner, inform it of what consumer we want to send events to, and then parse the file. The following code accomplishes this:

```
# set up the scanner, consumer and file to parse
from Bio.Fasta import Fasta
scanner = Fasta._Scanner()
consumer = TitleSearchConsumer()

file = open('taxol.fasta', 'r')
# parse all fasta records in the file

for n in range(22):
    scanner.feed(file, handler)}
```

Running this example gives the following output:

```
# python fasta_ex.py
Accession: AW615564.1
Accession: NM_004909.1
Accession: AF157562
...
```

This example uses the raw scanner-consumer interface we described above. Doing this can be clunky in many ways, since we have to explicitly know how many records we have to parse, and also need to access the scanner, which is marked as an internal class. The reason for this is that there are layers developed on top of the raw scanner and consumer classes which help make them more intuitive to use.

3.2.3 Parsing Output from Swiss-Prot

Delving further into taxol we next decide to look for further information in Swiss-Prot (`http://www.expasy.ch/sprot/sprot-top.htm`), a hand curated database of protein sequences. We search for Taxol or Taxus (Taxol was first isolated from the bark of the pacific yew, Taxus brevifolia). This yields 15 results, which we save in SwissProt format. Next, we would like to print out a description of each of the proteins found. To do this, we use an iterator to step through each entry, and then use the SwissProt parser to parse each entry into a record containing all of the information in the entry.

First, we set up a parser and an iterator to use:

```
from Bio.SwissProt import SProt

file = open('taxol.swiss', 'r')

parser = SProt.RecordParser()
my_iterator = SProt.Iterator(file, parser)
```

The parser is a RecordParser which converts a SwissProt entry into the record class mentioned above. Now, we can readily step through the file record by record, and print out just the descriptions from the record class:

```
next_record = my_iterator.next()

while next_record:
    print 'Description:', next_record.description
    next_record = my_iterator.next()
```

Utilizing the iterator and record class interfaces, the code ends up being shorter and more understandable. In many cases this approach will be enough to extract the information you want, while the more general Scanner and Consumer classes are always available if you need more control over how you deal with the data.

3.2.4 Downloading and Extracting information from Pub-Med

Finally, in our search for Taxol information, we would like to search PubMed (`http://www.ncbi.nlm.nih.gov:80/entrez/query.fcgi?db=PubMed`) and get Medline articles dealing with Taxol. Biopython provides many nice interfaces for doing just this.

First, we would like to do a PubMed search to get a listing of all articles having to do with Taxol. We can do this with the following two lines of code:

```
from Bio.Medline import PubMed

taxol_ids = PubMed.search_for('taxol')
```

Of course, article ids are not much use unless we can get the Medline records. Here, we create a dictionary that can retrieve a PubMed entry by its id, and use a Medline parser to parse the entry into a usable format. A PubMed dictionary is accessible using python dictionary semantics, in which the keys are PubMed ids, and the values are the records in Medlars format.

```
my_parser = Medline.RecordParser()
medline_dict = PubMed.Dictionary(parser = my\_parser)
```

Now that we've got what we need, we can walk through and get the information we require:

```
for id in taxol_ids[0:5]:
    this_record = medline_dict[id]
    print 'Title:', this_record.title
    print 'Authors:', this_record.authors
```

Running this code will give output like the following:

```
# python medline_ex.py
Title: PKC412--a protein kinase inhibitor with a broad therapeutic potential [In Process Citation]

Authors: ['Fabbro D', 'Ruetz S', 'Bodis S', 'Pruschy M', 'Csermak K', 'Man A',
'Campochiaro P', 'Wood J', "O'Reilly T", 'Meyer T']
...
```

In this example, the Biopython classes make it easy to get PubMed information in a format that can be easily manipulated using standard python tools. For instance, the authors are generated in a python list, so they could easily be searched with code like:

```
if 'Monty P'  in authors:
    print 'found author Monty P'
```

In sum, these examples demonstrate how the Biopython classes can be used to automate common biological tasks and deal with bioinformatic data in a pythonic manner.

4 Representing Sequences

4.1 Design Goals

Sequences of biological data are ubiquitous in bioinformatics, so it is essential to develop a standard sequence representation. The most natural way to think of sequences is as strings of characters, since this is the most common way they are encountered. Although this is an intuitive representation, it also very limiting, since sequences themselves have additional properties. In addition, many sequences, such as entire genomes, may be too large to fit into memory at the same time. Therefore, a critical challenge is designing the sequence interface which makes dealing with Biopython sequences as easy as dealing with strings, but which also allows the sequences to take on additional properties.

Biopython represents sequences as a lightweight sequence class that utilizes the object oriented nature of python and makes generous use of operator overloading. This combination allows sequences to be easily manipulated using common string operations, while also allowing sequence objects to be directly used in more complex operations. The examples below demonstrate the utility of this approach.

4.2 Usage Examples

The sequence class has two important attributes: its string representation, and an Alphabet class describing the allowable characters in the sequence. The representations we will use here are from the IUPAC nomenclature (`http://www.chem.qmw.ac.uk/iupac`). The two major advantages of this approach are that it assigns some rigidity and automatic error checking to your analysis, and that it allows the identity and characteristics of a sequence to be examined.

You can create a sequence object very simply:

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

The `my_seq` object is a `Seq` object, and not just a string:

```
>>> print my_seq
Seq('GCGATGCTATG', IUPACUnambiguousDNA())
```

However, the string can be easily obtained as an attribute of the `my_seq` object:

```
>>> print my_seq.data
GCGATGCTATG
```

Additionally, you can perform common string operations like getting the length or obtaining a slice directly on the sequence object:

```
>>> print len(my_seq)
11
>>> print my_seq[3:6]
Seq('ATG', IUPACUnambiguousDNA())
>>> my_seq2 = Seq('ATATATA', IUPAC.unambiguous_dna)
>>> print my_seq + my_seq2}
Seq('GCGATGCTATGATATATA', IUPACUnambiguousDNA())
```

Note that despite the fact that you are performing string operations on the sequence object, the return values are still Seq objects whose attributes, such as alphabet, are preserved.

Although you can deal with sequences simply in this manner, we can also do more complex operations directly on these objects. For instance, you can readily transcribe the DNA sequence into the corresponding RNA sequence:

```
>>> from Bio.Tools import Transcribe
>>> my_transcriber = Transcribe.unambiguous_transcriber
>>> print my_transcriber.transcribe(my_seq)
Seq('GCGAUGCUAUG', IUPACUnambiguousRNA())
```

Similarly, we can do translations using a number of different codon tables. Here, we will use the standard codon table to do the translation of our small sequence:

```
>>> from Bio.Tools import Translate
>>> my_translator = Translate.unambiguous_dna_by_name['SGC0']
>>> print my_translator.translate(my_seq)
Seq('AML', IUPACProtein())
```

Hopefully, these examples give a taste for the way you can interact with sequences through Biopython, and show the utility of this approach.

5 Other Tools

5.1 Biocorba interface

Because each programming language has its own advantages and disadvantages, one important goal is to achieve a degree of interoperability between libraries in different languages. A natural choice for helping to achieve this goal is the Common Object Request Broker Architecture (CORBA – `http://www.corba.org`), a set of specifications which allows code written in different languages to interoperate freely, even for programs running on different machines.

Taking advantage of CORBA, the Bioperl, Biojava and Biopython projects collaborated to develop an interface description allowing the projects to inter-communicate. Written in the interface definition language (IDL) of CORBA, it defines sequences with features, as well as methods to operate on those features.

Biopython has developed a module referred to as biopython-corba, which allows biopython to communicate with implementations of the biocorba interface written in perl and java. This allows python programmers to take advantage of code written in the other languages, and also allows the sharing of results and programs between researchers at different locations.

Biopython-corba also contains experimental modules for interacting with other biological CORBA servers. The European Bioinformatics Institute (EBI) has a number of biological database running with CORBA interfaces (`http://corba.ebi.ac.uk/index.html`). Accessing databases through CORBA can be quicker and more flexible then using web based interfaces. Due to the wide support for CORBA in python (there are currently 4 different Object Request Brokers (ORBs) with python interfaces), we believe that biopython-corba can be an important resource for bioinformatics programmers.

5.2 Classification Tools

Biopython also contains tools which implement common classification (supervised machine learning) algorithms, which are generally usable for any task which involves separating data into distinct groups. Currently, these type of algorithms are widely applied in analysis of microarray data, where researchers often want to separate genes into distinct groups based on their expression patterns. Currently, three different supervised classification algorithms are implemented in Biopython: Support Vector Machines, Naive Bayes, and k-Nearer-Neighbors. These tools are designed to be easy to get started with and contain 'train' functions which allow you to utilize default training algorithms in a quick analysis. However, they also contain interfaces for implementing custom training functions, so they are usable by advanced users.

5.3 Additional Functionality

In addition to all of the modules mentioned here, Biopython also contains parsers for dealing with data generated from BLAST, Enzyme, Prosite, and SCOP. Additionally, code exists for directly accessing popular bioinformatics web services, allowing access to these services directly from python scripts.

6 Future Goals

As mentioned, Biopython is still under heavy development, and has many exciting upcoming plans. In a true open-source nature, the code is always available for scrutiny in publicly accessible CVS archives, and discussions occur regularly on several list serves.

6.1 Planned Features

Biopython has numerous projects which are planned for the future, including:

- Parsers for even more biological sequence formats, including Genbank and PDB formats.
- Code for dealing with Sequence Annotations. Many useful features of sequences can be identified using computational tools, and being able to manipulate these annotations in Biopython would be very useful.
- Development of a graphical user interface which makes using the Biopython libraries easy for non-programmers. Prototypes for this currently exist, but much more is still planned.
- Further interaction with CORBA. CORBA is being utilitized more frequently in bioinformatics projects, and python has a number of well supported CORBA interfaces. This combination makes biopython an ideal place to collect CORBA code dealing with biological objects.

7 Contact Information

The Biopython project has a home page at `http://www.biopython.org`. This page contains the biopython mailing lists, CVS archives and releases of the code available for download, as well as links to other freely available python code for biological analysis. There are no barriers to joining the development team, and we are always looking for people to contribute ideas, code or discussions.

8 Conclusions

This paper described the Biopython project, which seeks to collect and organize freely available python code that deals with biological data. It detailed the unique aspects of python which make it an extremely useful language to working in bioinformatics. Code examples illustrated the design and use of several biopython modules, and other tools were more briefly described. Some upcoming plans for the project are also listed to give an indication of the directions that Biopython is headed. Biopython provides an important resource for the development of bioinformatics tools in python, and we hope to have excited you enough to give it a try, or maybe even volunteer to help!

9 Acknowledgements

Many thanks to all of the contributors to Biopython, especially Andrew Dalke and Cayte Lindner, and to Chris Dagdigian for setting up and maintaing the web site, mailing lists and bug tracking system. Brad Chapman is supported by a Howard Hughes Medical Institute Predoctoral Fellowship. Jeffrey Chang is supported in Stanford Medical Informatics by the Stanford Graduate Fellowship and the Burroughs Wellcome Fund.