PyBio: An Open Source Bioinformatics Library for Python.

PyBio: An Open Source Bioinformatics Library for Python.

Jon Robert Ellis

A Thesis in the Field of Information Technology

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

May 2016

Abstract

PyBio is an easy-to-install, open-source library for working with bioinformatics data in Python, designed to encourage interactive data exploration and scripting.

The Pybio API is designed to be explorable in an IPython session or in the Jupyter Notebook. The modules are laid out with hierarchical structure. All interface classes, functions, and modules are documented. Function hiding and module interface declarations are used to present users with a simple, powerful interface.

PyBio's modules are integrated with each other to encourage smooth workflows and simple, direct code. The same class abstractions are used throughout the library. The output of one function can often be used as the input for another. Opinionated, higher-level APIs abstract away complexity, lower the barrier to entry, and encourage expressive, functional code.

Bioinformatics data from NCBI's Entrez service is available from within Python through the `pybio.entrez` module. The module parses sequences directly from Entrez into PyBio's sequence representation, so acquiring and using new sequence data is fast and seamless.

PyBio is built for speed; Numpy and Cython are used extensively to achieve fast execution speeds not normally associated with Python, while maintaining Python's simplicity and clarity. Execution speed is critical for productive interactive data science. Performant tools written mostly in Python make bioinformatics code more accessible to less advanced programmers, fostering a closer connection between de-

velopers and biologists.

PyBio is a tool for bioinformatics developers. Classes like `Sequence` and `Alignment`, and modules like `pybio.parse`, provide a shared environment for new code, reducing the need to write parsers and data abstraction classes, and making it easier to write code that interoperates with existing code. PyBio could potentially provide a platform for new useful bioinformatics algorithms and implementations to become quickly available to the community.

PyBio has the potential to become a powerful tool for bioinformatics, encouraging a data science approach to bioinformatics data and stimulating innovation in the implementations of bioinformatics algorithms.

Acknowledgements

I would like to first thank my advisor, Edward Freedman. He has been a source of knowledge and experience, both while writing PyBio and at the Extension School. Thanks also to Jeff Parker, who supported me when things didn't go according to plan. I'll always appreciate his faith in me. I also need to thank Alain Viel, who gave his time to teach this programmer a little biology.

Harvard University and the Extension School have given me experiences and opportunities I would never have had otherwise. It was privilege to work and study at the University.

I want to thank my parents for supporting me, even when they didn't understand what I was doing or why. And finally, I want to thank my wife, Whitney, for her technical and writing advice, and for her patience and love.

Contents

List of Figures

List of Tables

List of Code Fragments

Chapter 1: Introduction

Bioinformatics is the application of computer science and engineering to biological data. It is a relatively new field, with a history going back to the 1980's and the first textbooks being published in the second half of the 1990's. (Jones & Pevzner, 2004) Bioinformatics-related technologies have since become ubiquitous for the every day work of many biologists.

Many biologists interact with bioinformatics technologies primarily through web interfaces, using code and data maintained on distant servers. This method avoids the difficulties of local setup and install of complicated libraries and tools. Web interfaces have proven to be a popular and effective way for biologists to use these tools.

Most data scientists do not interact with their data using web interfaces; many instead opt for general purpose programming languages with libraries and environments oriented toward data analysis, like R or Python with NumPy and Pandas. Although it requires more skill development, this is a powerful approach, allowing application of more general statistics and data analysis tools, integration of custom algorithms and workflows, and the ability to use local or hosted clusters. Live coding environments like R and Python encourage an exploratory approach to data that can enhance understanding and enable discovery in a way that is not possible with less interactive ways of working.

In this thesis, I present PyBio, a simple-to-install bioinformatics library for

Python, designed to assist data analysis approaches to bioinformatics by providing the basic classes and infrastructure for working with biosequences along with a rich set of available tools.

1.1. Design Philosophy

Several design considerations are leveraged to make PyBio a well-designed data analysis platform for bioinformatics.

1.1.1 IPython Design

The PyBio API is designed to work with the IPython command line or in an IPython Notebook. All interface functions, classes, and modules are documented. The modules are laid out with a hierarchical structure, which works well with the module exploration capabilities of IPython. Although Python does not allow private functions in the manner of C++ or Java, it does allow function hiding and interface declarations at the module level. PyBio uses those techniques to constrain the search space for users.

1.1.2 Performance

PyBio is designed to be performant. It can seem contradictory to design a Python library for performance; Python is usually chosen for simplicity, convenience, and programmer productivity. But with modern Python tools, performance is an achievable goal.

Array programming in NumPy is often very fast; the NumPy ufuncs that operate on its fixed type arrays are written in C, with the performance that implies. The array data is stored contiguous in memory, minimizing cache misses. NumPy is a mature system with code highly optimized for speed.

Cython is a system for compiling a superset of Python into C, with the primary purpose of shortening running time.

PyBio makes use of both NumPy and Cython, along with integrating C code, to achieve performance and productivity.

1.1.3 Workflow Integration

Programming libraries can sometimes become unusable lists of unrelated functions and classes. To avoid this pitfall, PyBio consciously attempts to integrate its components by emphasizing interoperability of functions, so that the output of one function can often be used for the input of another. This allows smooth workflows and simpler scripting code.

A standard pattern for PyBio is to provide a lower level API for the most direct access to functions and tools, but to encourage the use of a higher level API that more closely integrates the disparate modules of PyBio.

1.1.4 Opinionated Software

The higher level APIs of PyBio can also be understood in the context of 'opinionated software'. The idea of opinionated software is to structure an API to constrain the default behavior along a path that is applicable to most library users in order to increase code productivity and simplify the resulting code (Dumbill, 2005). An opinionated API offloads much of the work and design decisions from the application programmer, only requiring those decisions, along with the extra lines of code, when the program deviates from the default behavior. A good opinionated API should decrease the amount of time a software developer spends copying boilerplate, which in turn increases the time spent writing functional code. To quote David Heinemeier Hansson, the creator of Ruby on Rails, "Seeing your application work in the fraction of the time it took you before is a very concrete, individually rewarding goal" (Dumbill, 2005)

Opinionated software is closely associated with 'Convention over configuration', a rule of thumb that an API should be designed so that functions and classes have conventional behavior without any configuration, and that configuration should be used only to deviate from conventional behavior.

PyBio's API requires very little setup or boilerplate. Functions are direct, with few side effects, and are designed to be usable with a minimal number of arguments.

1.1.5 Batteries Included

Like Python's famous motto, PyBio has ambitions to be *batteries included*. Tools should be available in PyBio immediately after installing the package, rather than being implemented as wrappers around tools that must be separately acquired and installed or compiled.

PyBio leverages existing tools with public domain or free, open source licenses such as an MIT License or a GNU General Public License (GPL). PyBio's implementation of Smith-Waterman is an example of this philosophy; it leverages an existing tool with an MIT / Expat License, written in C, and combines this seamlessly in a PyBio module. A user of the library does not need to be aware of the implementation to the use the library; in fact, the implementation might change between releases of the library as better implementations become available.

PyBio's use of NumPy arrays as the underlying sequence data structure encourage the development and integration of new code by allowing the direct use of NumPy or Pandas, with all of the attendant speed and flexibility, and by allowing easy exposure of the underlying fixed-type array that is a natural data structure in C.

PyBio also includes the setup infrastructure necessary for using Cython in development, making it easier for programmers to make use of Cython or to more easily incorporate C code in new extensions.

Libraries like this that combine previous and new work to produce an integrated system have been very successful in the past, both as tools and for providing reference implementations of common algorithms, encouraging their further development within an open source framework. One of the most successful examples is NumPy, created by Travis Oliphant in 2005 by combining features and code from two competing platforms, Numarray and Numeric, with modifications to create an integrated user experience.

1.2. Objectives

The major objective of PyBio is to provide a stable, powerful foundation for integrating and developing bioinformatics tools, particularly for biosequences. I have made care to select good underlying representations of data that allow easier integration with the standard Python data libraries and existing C code.

PyBio supports common transformations of biosequences, comprehensive access to NCBI's Entrez databases, and a fast implementation of the Smith-Waterman alignment algorithm. The Smith-Waterman and Entrez systems are integrated with PyBio sequences; the `entrez` module can return a PyBio `Sequence`, and a PyBio `Sequence` can be used directly with the Smith-Waterman function.

1.3. Organization of Thesis

This thesis will start with this introduction and additional chapters to place PyBio in a proper setting, explaining the biology, bioinformatics, and technologies related to PyBio.

The next section is a User Guide. The User Guide is meant to be a self-contained introduction to using the PyBio package in IPython or the IPython Notebook, appropriate for a user with some understanding of biosequences and basic Python. For this thesis, it provides an explanation of the functionality of the library.

The next several chapters explain the internal implementation of the major divisions of PyBio. I then present a comparison of PyBio with existing bioinformatics libraries.

In the last chapter, I give some final thoughts: what lessons have been learned creating PyBio, and where PyBio is going from here.

Chapter 2: Bioinformatics

Bioinformatics can be viewed as part of computational science, an entirely new branch of science focusing on answering questions through massive data analysis and simulation, alongside traditional observational and experimental techniques. This has led to a common new term in biology, *in silico*, apparently introduced in 1989 by Pedro Miramontes from the National Autonomous University of Mexico at a workshop in Los Alamos, New Mexico (Vanjari et al., 2012), indicating experiments performed on a computer.

Bioinformatics is a broad interdisciplinary field, ranging from biology and chemistry to statistics, computer science, and software engineering. Bioinformatics subjects include sequence analysis, computational evolutionary biology, comparative genomics, analysis of gene and protein expression, biomolecular simulation, and network and systems biology.

PyBio focuses on the bioinformatics of biosequences, their biological transformations, and sequence analysis. This chapter provides a brief overview of the biology of the gene as it relates to PyBio functionality, some thoughts on biosequences as they relate to bioinformatics, and an introduction to sequence alignment.

2.1. Biology of the Gene

The genetics of life are written in DNA, deoxyribonucleic acid, a double helix of two polymer strands of nucleotides of four types of bases: guanine, adenine, thymine, and cytosine, denoted by the letters GATC. The nucleotides are connected to each other across the two polymer strands with hydrogen bonds according to the base pairing rule: A with T and C with G.

Segments of DNA are *transcribed* into RNA, ribonucleic acid, a more chemically active polymer of nucleotides, usually single-stranded or folded upon itself. The four nucleotide bases of RNA are guanine, adenine, uracil and cytosine, denoted by GAUC. The transcription process maps DNA to RNA according to base pairing, with uracil taking the place of thymine for RNA: A to U, T to A, G to C, and C to G.

RNA strands are used for many purposes in an organism. Ribosomal RNA (rRNA) makes up 60% of the mass of a ribosome, a cellular assembly composed of RNA and protein. Transfer RNA (tRNA) acts as an adapter molecule. Both ribosomes and tRNA are essential to the mechanism by which cells create proteins.

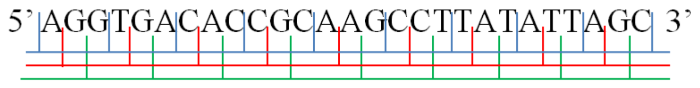

Figure 2.1: The three reading frames of a genomic sequence[1]

Messenger RNA (mRNA) encodes the sequences of proteins. mRNA is *translated* into polymers of amino acids, with sequences of three consecutive nucleotides in the mRNA called *codons* corresponding to a single amino acid. This amino acid polymer folds to create the final protein corresponding to the mRNA.

Since codons are composed of three nucleotides, a genomic sequence can be

[1]Image from Hornung Akos, available under the Creative Commons Attribution-Share Alike 3.0 Unported license

read starting at the first, second, or third position. The set of nucleotide triplets starting from a given position is called a *reading frame*.

2.2. Biosequences

A biosequence is arguably the fundamental class of biology. A biosequence might be a nucleotide sequence of DNA or RNA, or the amino acid sequence in a protein chain. These sequences are related; DNA is transcribed into RNA, RNA is translated into proteins.

In this sense, a biosequence is an abstraction of the data encoded in a DNA, RNA, or amino acid sequence. The biological transformations that occur to these molecules can be thought of approximately as the functional mapping between sequences, divorced from the chemistry that occurs between the molecules *in vitro* or *in vivo*.

2.3. Sequence Alignment

One fundamental set of informatics problems in biology is comparing two sequences to see if they have a functional or genetic relationship. Mutation of DNA might lead to insertions, deletions, or substitutions of nucleotides, so the sequences to compare may not even have the same length. To account for this, the order of the sequences is retained, but gap characters are inserted as needed to make the alignment.

```
GTTA-TGGAT-
GTAACTG-A-C
```

Figure 2.2: A global alignment of two DNA sequences

The alignment is scored for each position in the alignment according to a matrix of scores for each pair of symbols in the sequence alphabet, including the gap character. Mismatches and indels (insertions or deletions) are penalized, while matches are rewarded. A scoring matrix for a DNA sequence alignment would be five by five (four letters in the sequence alphabet plus a gap character), and they typically penalize matches by $-\mu$, penalize indels by $-\sigma$, and reward matches with a $+1$.

$$\begin{array}{c|ccccc} & A & C & G & T & - \\ A & 1 & -\mu & -\mu & -\mu & -\sigma \\ C & & 1 & -\mu & -\mu & -\sigma \\ G & & & 1 & -\mu & -\sigma \\ T & & & & 1 & -\sigma \\ - & & & & & -\sigma \end{array}$$

Figure 2.3: A DNA scoring matrix

Scoring matrices for protein sequences are considerably more complicated. Some mutations are more likely to alter the structure and function of proteins than others. The scoring matrix is adjusted for this by estimating the values from sets of known protein alignments. The BLOSUM62 matrix, used as the default in NCBI BLAST for protein sequences, is a common example.

Aligning the entirety of two sequences is called *global alignment.* It is useful for comparing gene and protein sequences with similar functions.

If the similarities are only in some portions of the sequence, a *local alignment* that maximizes the alignment over subsequences of each sequence is more appropriate. Local alignments are used for genes in which only a region of the gene is highly conserved, like the homeodomain in homeobox genes (Gehring et al., 1994), or when looking for similar subsequences of long sequences, as in a newly sequenced genome.

There are dynamic programming algorithms for finding the optimal global and local alignments, the Needleman-Wunsch algorithm (Needleman & Wunsch, 1970) and the Smith-Waterman algorithm (Smith & Waterman, 1981) respectively.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	-
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0	-2	-1	0	-4
R		5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3	-1	0	-1	-4
N			6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3	3	0	-1	-4
D				6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3	4	1	-1	-4
C					9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1	-3	-3	-2	-4
Q						5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2	0	3	-1	-4
E							5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2	1	4	-1	-4
G								6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3	-1	-2	-1	-4
H									8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3	0	0	-1	-4
I										4	2	-3	1	0	-3	-2	-1	-3	-1	3	-3	-3	-1	-4
L											4	-2	2	0	-3	-2	-1	-2	-1	1	-4	-3	-1	-4
K												5	-1	-3	-1	0	-1	-3	-2	-2	0	1	-1	-4
M													5	0	-2	-1	-1	-1	-1	1	-3	-1	-1	-4
F														6	-4	-2	-2	1	3	-1	-3	-3	-1	-4
P															7	-1	-1	-4	-3	-2	-2	-1	-2	-4
S																4	1	-3	-2	-2	0	0	0	-4
T																	5	-2	-2	0	-1	-1	0	-4
W																		11	2	-3	-4	-3	-2	-4
Y																			7	-1	-3	-2	-1	-4
V																				4	-3	-2	-1	-4
B																					4	1	-1	-4
Z																						4	-1	-4
X																							-1	-4
-																								1

Figure 2.4: The BLOSUM62 scoring matrix

```
    taaAGTTCAG-ATC-Ggggcacag
       ||| ||| ||| |
aattcggAGTACAGAATCGGacacgg
```

Figure 2.5: A local alignment of two DNA sequences

These algorithms, while exact, are slow for alignments over large databases of sequences. To perform alignments quickly at scale, heuristic algorithms based on using shared sets of subsequences were developed. One of the first successful attempts was FASTA, developed by David Lipman and William Pearson in 1985. (Lipman & Pearson, 1985) In 1990, FASTA was superseded by BLAST, the Basic Local Alignment Search Tool, developed by Stephen Altschul, Warren Gish, Webb Miller, Eugene Myers, and David Lipman. (Altschul et al., 1990) BLAST is more time-efficient than FASTA with similar sensitivity. BLAST has become a standard tool for biologists and bioinformaticians. FASTA lives on as a common file format for biosequences.

Chapter 3: Technologies

PyBio builds on a number of existing technologies to make it a flexible, useful, and performant tool for bioinformatics scripting.

3.1. C

C is a low-level, structured, general-purpose programming language (Kernighan & Ritchie, 1988) developed by Dennis Ritchie between 1969 and 1973 (Ritchie, 1993).

The syntax of C is designed to compile straightforwardly to efficient assembly code, allowing it to replace assembly code for many applications. C is often used for systems programming and programming language implementations.

3.2. Python

Python, first released in 1991, is a popular, high-level, dynamic, multi-purpose, interpreted programming language, well-suited to scripting and rapid prototyping (Perkel, 2015) (Scopatz & Huff, 2015). CPython, written in C, is the reference implementation of the interpreter (Python Software Foundation, 2015). Python is available on many platforms, including Windows, Macintosh, and Linux, as a free download from `http://www.python.org`.

Python is often said to be *batteries included*, a phrase usually attributed to Frank Stajano, meaning that the standard library, already available in any normal

Python installation, has a sufficient set of mature tools for most common tasks (Lutz, 2010). Although some standard library modules may be obsolete, poorly executed, or immature (Kuchling, 2005), *batteries included* mostly does work; third-party libraries are often unnecessary.

Where third-party libraries, such as PyBio, are wanted, Python comes included with the program `pip` for installing python packages from external repositories. The Python Software Foundation maintains the Python Package Index (*i.e.* PyPI), an official third-party package repository. Installation from PyPI is very simple; PyBio can be installed from PyPI simply by typing `pip install pybio`.

Python also makes an excellent "glue language", with many tools for working with code written in different programming languages (Johansson, 2015). Code generators like SWIG and SIP can be used to automate linking compiled modules. Cython allows writing Python-like code with static compilation and provides another method of using compiled C libraries. There are versions of Python that run in the Java enviroment and Microsoft's .NET, and there are packages for working with .NET modules and Windows COM components (Lutz, 2013).

These features, combined with Python's easy syntax and popularity, make Python a great environment for scientific programming. Python has a growing set of packages designed for scientific programming and data analysis including IPython, NumPy, SciPy, and pandas. PyBio can be thought of as an attempt to bring more accessible bioinformatics tools into this environment.

3.3. IPython

IPython is an interactive programming language command shell, originally designed for Python, but now available for many interpreted programming languages (Pérez & Granger, 2007). For end users, it provides an interactive prompt with introspection, tab completion, shell commands, rich media integration, and support for parallel programming.

The interactive introspection aspect of IPython is sometimes described as *namespace exploration*. By tab-completion, a user can explore through the namespace of a package or object, and, just by adding a question mark to the end of a query, IPython can introspect on the object and return documentation or the source code. This encourages an exploratory approach, very different from the *edit-compile-run* workflow common in traditional languages, that is ideal for interactive data analysis (McKinney, 2012).

3.4. Jupyter Notebook

Jupyter, released in 2015, is a spin-off project of IPython, which provides a language agnostic browser-based computing environment called a Notebook (Jupyter Project, 2015). A Jupyter Notebook is a JSON document, describing a list of input and output cells with code, text, images, web links, or other content, much like a Mathematica or Maple notebook. The Jupyter Notebook attaches to a running programming language kernel, such as IPython, using 0MQ, an asynchronous messaging library designed for communicating between programming languages (iMatix Corporation and Contributors, 2014).

Jupyter Notebook cells may also contain widgets, which use a Javascript API to interface IPython code with Javascript/HTML/CSS to allow interactive programs. These can be used to great effect in making large data much more explorable by providing an intuitive interface.

3.5. Numpy

NumPy is, as its developers claim, “the fundamental package for scientific computing with Python” (NumPy Developers, 2013). NumPy provides array programming through a N-dimensional array object, functions for element-wise computations and array operations, mathematical functions, and tools for integrating C, C++, and Fortran code (McKinney, 2012). Most of the code is written in C. It is fast, stable, and can simplify some coding patterns.

3.6. Numba

Numba is a just-in-time compiler for Python and NumPy, available as a Python package. This can often produce code that runs much faster than the original Python code (Gorelick & Ozsvald, 2014).

3.7. Cython

Cython is a superset of Python that compiles to C by integrating with the CPython interpreter. It includes a foreign function interface for calling C/C++ routines, and the ability to declare static types for function arguments, return values, local variables, and class attributes. (Bradshaw & Behnel, 2015)

The primary purpose of Cython is to speed up Python code. A substantial speed up can usually be made just by compiling Python code with Cython without even changing the code. Declaring static types often speeds up the code produced by the compiler. Functions can also be replaced with C/C++ code through the foreign function interface to produce code segments running at the speed of low-level code.

Chapter 4: Prior Work

PyBio builds on a great deal of prior work and existing software. In this chapter, I will review bioinformatics formats and software related to PyBio.

4.1. Biological File Formats

Bioinformatics data is often distributed though plain-text, human readable formats. Typically, a format is created as the input for some software release, and, over time, it becomes a *de facto* standard. This has led to some consensus over file formats, but there are still many formats, and the formats themselves are often not well-defined or well-designed (Cock et al., 2010). Any bioinformatics library needs to deal with a number of file formats to be a useful tool.

4.1.1 FASTA

The FASTA format is a simple plain-text format for recording nucleotide or peptide sequences, devised by Bill Pearson for the FASTA sequence alignment package (Pearson & J, 1998). FASTA has since become a common format for biosequences in other packages and services (Lipman & Pearson, 1985). FASTA is the most commonly used format for sequence analysis software, but not all software accepts the same set of codes (Mount, 2004) (Gibas & Jambeck, 2001).

The first line of a FASTA entry starts with a > symbol, and it may optionally contain an identifier and description. The nucleotides or amino acids are represented with single-letter codes. The sequence may be any number of lines, with any number of characters per line. Typically they are shorter than eighty characters. Here is a typical example, with identifier `gi|129295|sp|P01013|OVAX_CHICK`:

```
>gi|129295|sp|P01013|OVAX_CHICK GENE X PROTEIN (OVALBUMIN-RELATED)
QIKDLLVSSSTDLDTTLVLVNAIYFKGMWKTAFNAEDTREMPFHVTKQESKPVQMMCMNNSFNVATLPAE
KMKILELPFASGDLSMLVLLPDEVSDLERIEKTINFEKLTEWTNPNTMEKRRVKVYLPQMKIEEKYNLTS
VLMALGMTDLFIPSANLTGISSAESLKISQAVHGAFMELSEDGIEMAGSTGVIEDIKHSPESEQFRADHP
FLFLIKHNPTNTIVYFGRYWSP
```

The single-letter codes are not defined by the format, but NCBI tools like BLAST accept common sets for amino acids and nucleic acids based on standard IUB/IUPAC codes (IUPAC-IUB Joint Commission on Biochemical Nomenclature, 1984), but accepting both lower-case and upper-case letters, plus a hyphen or dash to represent a gap, and U and * in amino acids to respectively represent Selenocysteine and a translation stop.

NCBI supported nucleic acid codes are:

A	adenine	C	cytosine	G	guanine
T	thymine	U	uracil		
N	A, C, G, T, or U (any)	V	A, C, or G		
K	G, T, or U (keto)	M	A or C (amino)		
Y	C, T, or U (pyrimidine)	R	A or G (purine)		
S	C or G (strong)	W	A, T, or U (weak)		
B	not A	D	not C	H	not G
-	gap				

NCBI supported amino acid codes are:

A	alanine	B	D or N	C	cysteine
D	aspartic acid	E	glutamic acid	F	phenylalanine
G	glycine	H	histidine	I	isoleucine
J	L or I	K	lysine	L	leucine
M	methionine	N	asparagine	O	pyrrolysine
P	proline	Q	glutamine	R	arginine
S	seine	T	threonine	U	selenocysteine
V	valine	W	tryptophan	Y	tyrosine
Z	E or Q	X	any	*	translation stop
-	gap				

4.1.2 FASTQ

The FASTQ format is a plain-text format for recording a biological sequence along with quality scores.

The first two lines of a FASTQ file are similar to FASTA, except that the first line containing the title and optional description begins with a @ character. The second line is the encoded sequence. The third line, beginning with a + character, is an optional repeat of the title line. The last line contains the encoded quality string.

```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

Like FASTA, the biosequence encoding is not set by the format. The lines of the entry may be divided with line endings according to the most common specification of the format, but most FASTQ writers will not include line endings within the sequences (Cock et al., 2010).

The quality string records the PHRED quality score of a base call, so-called because it originated from a file format used by the software Phred base calling, used for identifying base sequences from flourescence data from an automated DNA sequencer. The PHRED quality score is defined as

$$Q_{PHRED} = -10 \times \log_{10} P_e$$

where P_e is the probability of error.

The scores, in the range 0-93, are recorded as ASCII characters 33-126, so that, for example, ASCII character 33, !, represents a score of 0. This was chosen because ASCII character 32 is a space, and ASCII character 126, ~, is the last printable ASCII character in the original ASCII character set. This range of scores allows assigments of probabilities to individual bases from 1.0, meaning that this base is definitely wrong, to $10^{-9.3}$.

4.2. Open Source Bioinformatics Software

Many people have put their time and skills into developing open source bioinformatics software. This has been an enormous benefit to the community, providing good tools accessible to all. Open source bioinformatics tools are critical to good science, because only code that can be read and verified by anyone allows the repeatability and candor that lie at the heart of the scientific method.

Many of these programs are written as command line tools with output as text or a file. Typically, they are written in C, C++, or Fortran. They can be difficult to install or configure (Gibas & Jambeck, 2001).

One of the goals of Pybio is to leverage this work to provide an integrated environment for bioinformaticians, making it easier for people to use these tools. Pybio could expand the use of well-written scientific tools, as well as providing a community to focus work on improving useful and popular code.

4.2.1 SSW Library

The SSW Library is an open-source implementation of the Smith-Waterman algorithm to find the optimal pairwise alignment between two sequences (Zhao et al., 2013). It extends earlier work by Michael Farrar on a Striped Smith-Waterman implementation (Farrar, 2006), adding the use of SIMD (Single Instruction, Multiple Data) for x86 processors. The SSW implementation is about 50 times faster than a typical optimized Smith-Waterman implementation.

SSW is written as a standalone C library, and it doesn't require special hardware like FPGAs (field-programmable gate arrays) or GPUs (graphical processing units). This makes it relatively straight-forward to integrate it with other systems.

4.3. Python Bioinformatics Libraries

PyBio is not the first attempt to create a bioinformatics library for Python. Here I discuss two of the most successful.

4.3.1 Biopython

Biopython is a mature, open-source bioinformatics library for Python, targeting a large set of bioinformatics applications (Cock et al., 2009). It was intially released by Brad Chapman and Jeffrey Chang in 2000. Biopython receives support from the Open Bioinformatics Foundation, along with related libraries targeting other platforms: BioPerl, BioJava, BioRuby, and BioSQL.

Biopython provides support for biosequence manipulations, parsing of various bioinformatics input and output formats, api's for online resources, wrapping code for bioinformatics command line programs like BLAST, functions for structure calculations, and more.

Biosequences are represented by the `Seq` class, which behaves as a normal Python string, augmented with biological methods, annotations, and the concept of an allowed alphabet. Internally, the sequence is stored as a Python string, and manipulated using Python string functions.

4.3.2 scikit-bio

scikit-bio is a more recently developed bioinformatics library for python. It was first released in 2014, and is currently in beta (scikit-bio developers, 2015).

Like Biopython, scikit-bio also represents a sequence with a core class, here called `Sequence`. Although it also has the concept of an allowed alphabet, it includes derived classes for difference types of biosequences: `DNA`, `RNA`, and `Protein`.

Before the version 0.4.0 release in July 2015, the internal representation of the sequence was as a Python string. In current code, it is now a numpy `ndarray` of type uint8.

Along with sequences and parsing of bioinformatics formats, scikit-bio also has support for biological diversity functions, some general statistics, and workflow generation.

Chapter 5: User Guide

PyBio is an open-source library for bioinformatics data, designed both for scripting and for interactive use in the IPython and IPython Notebook environments. The goals are fast execution speed, local code, and an intuitive, integrated API for encouraging bioinformatics data exploration and scripting.

This user guide will take you through installation and usage of the major PyBio functions and classes.

5.1. Prerequisites

The only necessary prerequisite for installing PyBio is a working Python 2.7+ or Python 3.3+ install.

PyBio runs fine in a standard Python environment, but IPython and the Jupyter IPython notebook are better environments for Python data analysis. IPython can be installed with `pip`.

```
pip install ipython
```

After installing, just type `ipython` to start the IPython environment.

Jupyter can also be installed with `pip`.

```
pip install jupyter
```

To start the Jupyter application, type `jupyter notebook`. This will bring up the application in a browser tab or window. To start an IPython notebook, click on 'New' and select 'Python 2' or 'Python 3'.

5.2. Installation

The easiest way to install PyBio is with `pip`:

```
pip install pybio
```

This will download, install, and update all dependencies.

5.2.1 Installing from source

The source code for PyBio can be downloaded from `https://github.com/jrellis/pybio`. To install, run `setup.py`.

```
python setup.py install
```

This is a standard python installation using the `setuptools` module.

5.2.2 Installation from source for development

If a project is installed with the `setuptools install` command, it must be rebuilt and reinstalled every time a change is made. `setuptools` supports a `develop` command, which creates an `.egg-link` file in the Python deployment directory that links the the project's source code (Ziade, 2008). To use this command with the PyBio source code, just run

```
python setup.py develop
```

A few extra commands have been added to PyBio's `setup.py` to assist further in development.

Running the `install` command does not build PyBio's Cython files and accompanying C source code, to avoid having a dependence on a local C compiler. To rebuild the Cython files, use the `cythonize` command.

```
python setup.py cythonize
```

As a convenience, PyBio's `setup.py` also has a `cython_develop` command which rebuilds the Cython and C code and runs the `develop` command.

```
python setup.py cython_develop
```

5.3. Starting PyBio in IPython

To demonstrate working with PyBio, I am going to assume an IPython or Jupyter IPython Notebook session. Although PyBio doesn't require IPython, PyBio's API is designed to be productive in an IPython environment.

After starting an IPython (by typing `ipython`) or Jupyter Notebook session, import the PyBio package.

```
In [1]: import pybio
```

IPython allows introspection on the doc strings of Python classes, modules, and functions by adding a '?' after the name. We can use this to start finding out what PyBio is all about.

```
In [2]: pybio?
Type:        module
String form: <module 'pybio' from '/home/rob/code/pybio/pybio/__init__.
   py'>
File:        ~/code/pybio/pybio/__init__.py
```

```
Docstring:
PyBio - a bioinformatics data analysis library for Python

See https://github.com/jrellis/pybio for more documentation, or see the
docstrings of classes, modules, and functions in the pybio namespace:

Sequence
DnaSequence
RnaSequence
ProteinSequence
entrez
alignment
parse
```

PyBio's public API is fully documented, so this will work with any module, class, or function in PyBio.

IPython does namespace exploration through tab completion.

```
In [3]: pybio.<tab>
pybio.DnaSequence      pybio.Sequence         pybio.logging
pybio.ProteinSequence  pybio.alignment        pybio.parse
pybio.RnaSequence      pybio.entrez           pybio.sequence
```

The design of the PyBio API exposes useful functions and classes shallowly in the namespace, while hiding references to internally used packages and internal functions deeper in the namespace structure. IPython also recognizes the Python convention of beginning the names of 'private' functions with an underscore by not exposing those with a tab completion, unless the user explicitly writes the underscore before the name. PyBio takes advantage of this by uniformly naming private functions with the underscore convention.

5.4. Working with Biosequences

Like other bioinformatics libraries, a central object in PyBio is the sequence. In PyBio, this is the `Sequence` class.

The `Sequence` class can be instantiated with any Python string. An `alphabet` incorporating the unique characters in the string is created. A string representation of the sequence can be returned with the `string` property or the `to_string` method. A list representation can be returned with the `to_list` method. Sequences can be sliced to produce subsequences or used as an iterator, like a Python `string`.

```
In [1]: import pybio

In [2]: seq = pybio.Sequence('an arbitrary string')

In [3]: seq
Out[3]: Sequence(an arbitrary string)

In [4]: seq.alphabet
Out[4]: ['a', ' ', 'b', 'g', 'i', 'n', 's', 'r', 't', 'y']

In [5]: seq.string
Out[5]: 'an arbitrary string'

In [6]: seq.to_string()
Out[6]: 'an arbitrary string'

In [7]: seq.to_list()
Out[7]:
['a', 'n', ' ', 'a', 'r', 'b', 'i', 't', 'r', 'a', 'r', 'y', ' ', 's', '
   t', 'r', 'i', 'n', 'g']

In [8]: seq[3:12]
Out[8]: Sequence(arbitrary)

In [9]: ''.join([x.upper() for x in seq])
Out[9]: 'AN ARBITRARY STRING'
```

PyBio has three classes that inherit from the `Sequence` class: `DnaSequence`, `RnaSequence`, and `ProteinSequence`. Each of these is defined by a limited alphabet and a set of methods unique to each.

The alphabet for the `DnaSequence` class is A, C, G, T, and N [1].

[1] I plan to extend the allowed alphabets for `DnaSequence` and `RnaSequence` to include the full set of IUPAC codes.

It has four methods: `transcribe`, `complement`, `reverse_complement`, and `translate`. `transcribe` converts the `DnaSequence` into an `RnaSequence`, according to gene transcription. `complement` returns the complement to the `DnaSequence`. `reverse_complement` returns the reverse complement as a `DnaSequence`. `translate` returns the translated protein sequence, up to the first stop codon encountered, as a `ProteinSequence`.

```
In [2]: seq = pybio.DnaSequence('ACGTTGCAC')

In [3]: seq.complement()
Out[3]: DnaSequence(TGCAACGTG)

In [4]: seq.reverse_complement()
Out[4]: DnaSequence(GTGCAACGT)

In [5]: seq.transcribe()
Out[5]: RnaSequence(ACGUUGCAC)

In [6]: seq.translate()
Out[6]: ProteinSequence(TLH)
```

The `RnaSequence` has an alphabet of A, C, G, U, and N. Like the `DnaSequence`, `RnaSequence` also has `complement`, `reverse_complement`, and `translate` methods. There is also the `reverse_transcribe` method, which returns the reverse transcription of the sequence as a `DnaSequence`.

```
In [2]: seq = pybio.RnaSequence('ACGUUGCAC')

In [3]: seq.complement()
Out[3]: RnaSequence(UGCAACGUG)

In [4]: seq.reverse_complement()
Out[4]: RnaSequence(GUGCAACGU)

In [5]: seq.reverse_transcribe()
Out[5]: DnaSequence(ACGTTGCAC)

In [6]: seq.translate()
Out[6]: ProteinSequence(TLH)
```

The `ProteinSequence`'s alphabet is the entire IUPAC Protein code set, including X for any and * for a translation stop. It has no public methods beyond those defined for the `Sequence` class.

5.5. Parsing

PyBio provides support for parsing to and from FASTA and FASTQ formats. `pybio.parse` is a low-level module supporting parsing to and from Python `namedtuple` objects. There are also `Sequence` class methods for supporting reading files directly into `Sequence` objects.

5.5.1 `pybio.parse`

There are two classes in `pybio.parse`, `Fasta` and `Fastq`. They are both implementations of Python's `namedtuple`, with appropriate fields for storing the data associated with each type of record. Both classes have the same methods.

FASTQ and FASTA are related formats, and both encode an identifier, a description, and a sequence. The FASTQ format also has a quality score sequence. Each of these has a corresponding field in the type.

The `parse_string` class method takes a FASTA or FASTQ entry as a string and creates a `Fasta` or `Fastq` object. The `to_string` method reverses the process, producing a string from the object.

```
In [2]: fasta = pybio.parse.Fasta.parse_string('>gi|995970649|pdb|5EJK|
    NN Chain n, Crystal Structure Of The Rous Sarcoma Virus Intasome\
    nAGTGTCTT')

In [3]: fasta
Out[3]: Fasta(identifier='gi|995970649|pdb|5EJK|NN', description='Chain
    n, Crystal Structure Of The Rous Sarcoma Virus Intasome', sequence='
    AGTGTCTT')
```

```
In [4]: fasta.to_string()
Out[4]: '>gi|995970649|pdb|5EJK|NN Chain n, Crystal Structure Of The
    Rous Sarcoma Virus Intasome\nAGTGTCTT'
```

The `parse_iterator` class method takes a FASTA or FASTQ file, or a string, with multiple entries, and returns an iterator over those entries that returns `Fasta` or `Fastq` objects. There is an optional `description` parameter that will return only those entries whose description line matches a description string. The `description` allows using Unix shell-style wildcards.

For example, here the descriptions are returned for each sequence that contains the string 'Gallus gallus' in `mito.nt`, NCBI's BLAST database of mitochondrial genomes as a FASTA-formatted file.

```
In [2]: [fasta.description for fasta in  pybio.parse.Fasta.
    parse_iterator('mito.nt', description='*Gallus gallus*')]
Out[2]:
['Gallus gallus mitochondrion, complete genome',
 'Gallus gallus spadiceus mitochondrion, complete genome',
 'Gallus gallus gallus mitochondrion, complete genome',
 'Gallus gallus bankiva mitochondrion, complete genome']
```

5.5.2 Parsing FASTA and FASTQ into `Sequence` Objects

Instead of returning `Fasta` or `Fastq` named tuples, class methods of the `Sequence` class can directly return `Sequence` objects from FASTA or FASTQ formatted files or strings.

The class methods `from_fasta` and `from_fastq` return `Sequence` objects from FASTA and FASTQ files, or `Fasta` and `Fastq` objects. These functions also take parameters for returning only entries whose description lines match a description string, or a specific sequence by order (starting from zero, so that, for example, '2' returns the third entry).

All `Sequence` objects have an `identifier` and a `description`, but these are not set by default. Typically, a `Sequence` created from a function in a PyBio module, like the `from_fasta` method, will attempt to fill these fields.

```
In [2]: dna = pybio.DnaSequence.from_fasta('mito.nt', 2, description='*
   Gallus gallus*')

In [3]: dna
Out[3]: DnaSequence(AATTTTATTTTTTAACCTAA...TTGTTCTCAACTACGGGAAC)

In [4]: dna.description
Out[4]: 'Gallus gallus gallus mitochondrion, complete genome'

In [5]: dna.describe()
gi|71658078|ref|NC_007236.1|
Gallus gallus gallus mitochondrion, complete genome
AATTTTATTTTTTAACCTAA...TTGTTCTCAACTACGGGAAC
```

The class methods `sequences_from_fasta` and `sequences_from_fastq` return iterators over the FASTA and FASTQ formatted file that return `Sequence` objects. Like `pybio.parse.Fasta.parse_iterator`, these functions can also take an optional description pattern.

```
In [2]: [(fasta, fasta.description) for fasta in pybio.DnaSequence.
   sequences_from_fasta('mito.nt', description='*Gallus gallus*')]
Out[2]:
[(DnaSequence(AATTTTATTTTTTAACCTAA...TTGTTCTCAACTACGGGAAC),
  'Gallus gallus mitochondrion, complete genome'),
 (DnaSequence(AATTTTATTTTTTAACCTAA...TTGTTCTCAACTACGGGAAC),
  'Gallus gallus spadiceus mitochondrion, complete genome'),
 (DnaSequence(AATTTTATTTTTTAACCTAA...TTGTTCTCAACTACGGGAAC),
  'Gallus gallus gallus mitochondrion, complete genome'),
 (DnaSequence(AATTTTATTTTTTAACCTAA...TTGTTCTCAACTACGGGAAC),
  'Gallus gallus bankiva mitochondrion, complete genome')]
```

Both of these methods will also work directly with FASTA and FASTQ files compressed with Zip, gzip, or bzip2.

```
In [2]: dna = pybio.DnaSequence.from_fasta('mito.nt.gz', 2, description
    ='*Gallus gallus*')

In [3]: dna.describe()
gi|71658078|ref|NC_007236.1|
Gallus gallus gallus mitochondrion, complete genome
AATTTTATTTTTTAACCTAA...TTGTTCTCAACTACGGGAAC

In [4]: [(fasta, fasta.description) for fasta in pybio.DnaSequence.
    sequences_from_fasta('mito.nt.gz', description='*Gallus gallus*')]
Out[4]:
[(DnaSequence(AATTTTATTTTTTAACCTAA...TTGTTCTCAACTACGGGAAC),
  'Gallus gallus mitochondrion, complete genome'),
 (DnaSequence(AATTTTATTTTTTAACCTAA...TTGTTCTCAACTACGGGAAC),
  'Gallus gallus spadiceus mitochondrion, complete genome'),
 (DnaSequence(AATTTTATTTTTTAACCTAA...TTGTTCTCAACTACGGGAAC),
  'Gallus gallus gallus mitochondrion, complete genome'),
 (DnaSequence(AATTTTATTTTTTAACCTAA...TTGTTCTCAACTACGGGAAC),
  'Gallus gallus bankiva mitochondrion, complete genome')]
```

5.5.3 FASTA and FASTQ from `Sequence` objects

The `Sequence` class has two pairs of methods for writing to FASTA and FASTQ formats.

A `Fasta` object can be produced from a `Sequence` with the `to_fasta` method. A `Sequence` can also be written directly to a file, using the `write_fasta` method. By default, the file name written will be the `Sequence identifier` with a `.fasta` extension, but the function can also take a file name or file handle as an argument. The `overwrite` argument allows overwriting or appending the FASTA entry to the file.

```
In [4]: seq.describe()
gi|995970649|pdb|5EJK|NN
Chain n, Crystal Structure Of The Rous Sarcoma Virus Intasome
AGTGTCTT

In [5]: seq.to_fasta()
```

```
Out[5]: Fasta(identifier=u'gi|995970649|pdb|5EJK|NN', description=u'
    Chain n, Crystal Structure Of The Rous Sarcoma Virus Intasome',
    sequence='AGTGTCTT')

In [6]: seq.write_fasta()

In [7]: %ls *.fasta
gi|995970649|pdb|5EJK|NN.fasta

In [8]: %cat gi\|995970649\|pdb\|5EJK\|NN.fasta
>gi|995970649|pdb|5EJK|NN Chain n, Crystal Structure Of The Rous Sarcoma
     Virus Intasome
AGTGTCTT
```

`Sequence` objects have similar functions for the FASTQ format. `to_fastq` returns a `Fastq` object, and `write_fastq` write the `Sequence` to disk in the FASTQ format.

5.6. Entrez

Entrez is a text search and retrieval system provided by the NCBI. The data includes forty biological literature and molecular databases, covering PubMed entries, DNA sequences, protein sequences, and much more (NCBI, 2014).

NCBI provides the Entrez Programming Utilities, also called E-utilities, for accessing Entrez (Sayers, 2010). The E-utilities are a set of server-side programs with a REST-like url interface.

PyBio provides thin wrappers around the entire set of E-utilities, a small set of standard pipelines using these utilities, and a set of functions for returning `DnaSequence` and `ProteinSequence` objects from Entrez's sequence databases. All of these functions are in the `pybio.entrez` module.

5.6.1 Entrez Usage Guidelines

Entrez maintains a set of requirements and guidelines for making E-utility requests. These are mostly common sense limits to avoid overwhelming NCBI's servers.

NCBI asks users to limit URL requests to no more than three per second. PyBio enforces this rule in code without any intervention by the user.

NCBI also recommends that users run large jobs on weekends or between 9:00 PM and 5:00 AM Eastern standard time. PyBio leaves this consideration to the user.

5.6.2 tool and email values

NCBI may block an IP address if usage limits are exceeded (Sayers, 2010). If this happens, NCBI will attempt to contact the user through a provided email.

PyBio allows setting this email with the `set_email` function. Although PyBio does not enforce setting an email, it will log a warning when Entrez requests are made without setting an email.

The `tool` value identifies a registered tool with NCBI. In PyBio, this value is set by default to 'pybio'. The function `set_tool` allows resetting this tool value, for use by applications or libraries that make use of PyBio as an imported package.

5.6.3 Common Parameters for E-utilities

The E-utilties are implemented as web service, with the tool name and a set of parameters encoded in an HTTP request. The set of required and available parameters varies per tool.

While none of the parameters are available in every tool, some of the parameters are recurring. Except for `EGQuery`, all tools have a `db` option to specify the Entrez database.

5.6.3.1 Retrieval Parameters

Many of the tools offer some of the retrieval parameters: `retmode`, `rettype`, `retstart`, and `retmax`.

`retmode` specifies the data format of the returned record. The available formats depend on the tool and database. Common `retmode` values are 'xml', 'json', and 'txt'.

`rettype` specifies the view of the returned record. The available types vary considerably with the database and tool. As an example, in most databases that return a sequence, 'native' will return XML in the form associated with that database, but 'fasta' will return the sequences in FASTA format. Only some combinations of `retmode` and `rettype` may work for a particular database: for `rettype` 'native', `retmode` must be 'xml', but for `rettype` 'fasta', `retmode` may be 'xml' or 'text'.

For tools returning multiple records, `retstart` specifies the sequential index of the first record to be returned. `retmax` limits the total number of records returned, up to 10,000. Using these values together allows getting more records than the maximum by repeated requests, each incrementing `retstart` by the value of `retmax`.

5.6.3.2 Entrez History Parameters

The Entrez E-utilities `epost` and `esearch` both allow saving a set of UIDs to the Entrez History server. They return two parameters, `WebEnv` and `query_key` that can be used by many of the tools to refer to the saved history.

5.6.4 PyBio Entrez Wrappers

The Entrez E-utilities typically return a variety of documents types, including XML, json, FASTA, and others, depending on the tool used and parameters that are passed.

PyBio's Entrez wrappers return a `Response` object from Python's `requests` package. These objects have fields that give information about the request, information about the request response, and the response itself.

```
In [3]: einfo_request = pybio.entrez.einfo()
INFO:urllib3.connectionpool:Starting new HTTPS connection (1): eutils.
   ncbi.nlm.nih.gov

In [4]: einfo_request.ok
Out[4]: True

In [5]: einfo_request.status_code
Out[5]: 200

In [6]: einfo_request.url
Out[6]: u'https://eutils.ncbi.nlm.nih.gov/entrez/eutils/einfo.fcgi?tool=
   pybio&email=jrellis%40fas.harvard.edu'

In [7]: print einfo_request.text
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE eInfoResult PUBLIC "-//NLM//DTD einfo 20130322//EN" "http://
   eutils.ncbi.nlm.nih.gov/eutils/dtd/20130322/einfo.dtd">
<eInfoResult>
<DbList>
        <DbName>pubmed</DbName>
        <DbName>protein</DbName>
        <DbName>nuccore</DbName>]
...
        <DbName>unigene</DbName>
        <DbName>gencoll</DbName>
        <DbName>gtr</DbName>
</DbList>
</eInfoResult>
```

PyBio implements all nine of the Entrez E-utilities tools. Each wrapper requires the required parameters for the E-utilities tool as function arguments, then takes non-required parameters as keyword arguments. These correspond almost exactly with the parameters described in the Entrez E-utilities documentation, with the exception that the Python wrappers do not allow '+' instead of a space character, which is an optional syntax in the E-utilities.

Here, I briefly describe each E-utility tool and PyBio wrapper. For more complete information, read Eric Sayers' *The E-utilities In-Depth: Parameters, Syntax and More* (Sayers, 2015). The E-utility tool name and the PyBio function name match.

5.6.4.1 einfo

`einfo` provides meta information on Entrez databases as XML or JSON. The `db` parameter specifies the database. If called without the `db` parameter, `einfo` returns the list of databases in Entrez.

This information includes links between databases and a list of search terms that can be used in `esearch`.

```
# Returns all Entrez database names
pybio.entrez.einfo()

# Returns statistics for the 'protein' database
pybio.entrez.einfo(db='protein')
```

5.6.4.2 esearch

`esearch` returns UIDs matching a query in a database. `esearch` requires a `db` paramter and a `term` parameter that carries the search terms. The search terms may be plain text, or they can refer to a specific field. For example, setting `term='X98419[accession]'` will search for entries with accession number X98419. The `usehistory` parameter allows saving the UIDs returned from the query to the Entrez History server.

```
# Search in the 'protein' database within a molecular weight range
pybio.entrez.esearch(db='protein', term='4000:8000[molecular weight]')
# Search for all tRNA in the 'nucleotide' database
pybio.entrez.esearch(db='nucleotide', term='biomol trna[prop]')
```

5.6.4.3 epost

`epost` uploads a list of UIDs to the Entrez History server. `db` and `id` are required parameters. `id` is a comma-separated list of UIDs.

```
# Post three UIDs from the 'protein' database, and return the WebEnv and
    query\_key
pybio.entrez.epost(db='protein', id='15718680,157427902,119703751')
```

5.6.4.4 esummary

`esummary` returns document summaries that match a set of IUDs in a database. `db` is a required parameter. The input set of UIDs can be set with the `id` parameter, or from the Entrez History.

```
# Return document summaries for two UIDs in the 'nucleotide' database
pybio.entrez.esummary(db='nucleotide', id='28864546,28800981')
```

5.6.4.5 efetch

`efetch` returns data records matching a set of UIDs. `db` is a required parameter. The input set of UIDs can be set with the `id` parameter, or from the Entrez History.

```
# Return data records for two UIDs in the 'nucleotide' database
pybio.entrez.efetch(db='nucleotide', id='28864546,28800981')
```

5.6.4.6 elink

`elink` returns UIDs that are linked to an input set of UIDs. The `db` parameter indicates the database from which the returned UIDs are retrieved and the `dbfrom` parameter indicates the database corresponding to the input UIDs. The input set of UIDs can be set with the `id` parameter, or from the Entrez History. The optional `cmd` parameter indicates how the UIDs are linked; for example, `cmd=neighbor`, the default, returns directly linked UIDs, and `cmd=neighbor_score` returns UIDs within the same database as the input UIDs with a similarity score.

```
# Return UIDs in the 'gene' database that are linked to from a set of
   UIDs in the 'protein' database
pybio.entrez.elink(db='gene', dbfrom='protein', id='15718680,157427902')
```

5.6.4.7 egquery

`egquery` gets the number of records in all the Entrez databases corresponding to a text query. There is only one parameter, `term`, which is required.

```
# Return the number of records matching 'cancer'
pybio.entrez.egquert(term='cancer')
```

5.6.4.8 espell

`espell` returns spelling suggestions, given a database, `db`, and a `term`.

```
# Give spelling suggestions matching 'canser'
pybio.entrez.espell(db='PubMed', term='canser')
```

5.6.4.9 ecitmatch

`ecitmatch` returns PubMed IDs given a citation string in the form 'journal|year|volume|first page|author|label|', where label is an arbitrary label provided by the user.

```
pybio.entrez.ecitmatch(db='PubMed', retmode='xml', bdata='science
   |1987|235|182|palmenberg ac|label|')
```

5.6.5 Entrez Pipelines

Entrez is designed so that many common operations require two or more requests to the E-utilities along with parsing of the results of the intermediate requests. PyBio provides three of the functions for common Entrez pipelines.

5.6.5.1 esearch_id

esearch_id performs an esearch Entrez query and returns the list of UIDs as a Python list.

```
In [2]: pybio.entrez.esearch_id(db='gene', term='homo sapiens')
INFO:urllib3.connectionpool:Starting new HTTPS connection (1): eutils.
    ncbi.nlm.nih.gov
Out[2]:
['7157', '7124', '348', '1956', '7422', '4524', '3569', '7040', '1636',
    '2099', '351', '6532', '3123', '3586', '155971', '2064', '672',
    '3091', '4790', '9370']
```

5.6.5.2 esearch_esummary

esearch_esummary performs an esearch, saves the resulting UIDs to the Entrez History server, then returns the summaries as a Python requests Response object.

```
# Return the summary information for up to 20 proteins with a molecular
    weight between 7000 and 9000
pybio.entrez.esearch_esummary(db='protein', term='7000:9000[molecular
    weight]', retmax=20)
```

5.6.5.3 esearch_efetch

esearch_efetch performs just as esearch_esummary, but it fetches the actual entry for each rather than the summary.

```
# Return the data record for up to 20 proteins with a molecular weight
    between 7000 and 9000
pybio.entrez.esearch_efectch(db='protein', term='7000:9000[molecular
    weight]', retmax=20)
```

5.6.6 Sequences from Entrez

PyBio provides a higher-level interface for reading `Sequences` directly from the ‘protein’ and ‘nucleotide’ Entrez databases.

Given an accession number, `get_nucleotide_sequence` makes a request to search the ‘nucleotide’ Entrez database, and returns a `DnaSequence` object.

```
In [3]: seq = pybio.entrez.get_nucleotide_sequence('X94419')
INFO:urllib3.connectionpool:Starting new HTTPS connection (1): eutils.
   ncbi.nlm.nih.gov
INFO:urllib3.connectionpool:Starting new HTTPS connection (1): eutils.
   ncbi.nlm.nih.gov

In [4]: seq.describe()
gi|1181236|emb|X94419.1|
M.musculus mRNA (3C10) for IgA V-D-J-heavy chain
GGAGCTGAGATTATAAGGCC...TTCCGGCACGATGAATGTGA
```

`get_protein` is a similar function that searches the ‘protein’ Entrez database and returns a `ProteinSequence` object.

PyBio has another pair of functions, `search_nucleotide_sequences` and `search_proteins`, that return iterators over `Sequence` objects that match a set of search terms. The complete set of search parameters is included in the doc string for each function.

Each function has three optional parameters to control the iteration. The `max_count` parameters give the maximum number of `Sequence` objects that can be returned. `max_per_call` and `queue_size` are handles for controlling the asynchronous queueing system used by the functions.

```
In [3]: [x for x in pybio.entrez.search_proteins(max_count=10,
   molecular_weight='120000:120002')]
INFO:urllib3.connectionpool:Starting new HTTPS connection (1): eutils.
   ncbi.nlm.nih.gov
INFO:urllib3.connectionpool:Starting new HTTPS connection (1): eutils.
   ncbi.nlm.nih.gov
Out[3]:
```

```
[ProteinSequence(MAEDESNSSSMIAQIEKVVT...SPFPMTMKIQWDGDLPARKE),
 ProteinSequence(MNIWQYLKPKNDKTFEVLAL...FYLGGELKDNTFYFDPRSIW),
 ProteinSequence(MPKRTDISNILLIGSGPIVI...LALEESSKEDELLALQDYLK),
 ProteinSequence(MSIITHQVRSVAATDTIYDD...FESTIDKLRRELSNAIAGTS),
 ProteinSequence(MPKRTDISSILVIGAGPIII...AIAEVSPDQLEVRSLQDYYS),
 ProteinSequence(MPKRIDISNILLIGSGPIVI...LALEESSKEDELLALQDYLV),
 ProteinSequence(MPKRIDISNILLIGSGPIVI...LALEESSKEDELLALQDYLV),
 ProteinSequence(MLHLFYSNRHETLVDALLDD...FGALAHLVFDPLVEHLRRSS),
 ProteinSequence(MLHLFYSNRHETLVDALLDD...FGALAHLVFDPLVEHLRRSS),
 ProteinSequence(MERKSANIDDVIRTVEIASA...MRTVPLSIDDPLMNDYARVD)]
```

5.7. Smith-Waterman Alignment

PyBio provides an implementation of the Smith-Waterman algorithm for optimal local alignment. PyBio makes use of C code from the SSW Library, an open-source implementation of Smith-Waterman optimized for use of SIMD on x86 processors (Zhao et al., 2013).

The API consists of a single public function, `pybio.alignment.smith_waterman`, which takes a query sequence, a target sequence or file, and arguments for tuning the alignment.

```
In [2]: target = pybio.DnaSequence.from_fasta('test_target.fa')

In [3]: query = pybio.DnaSequence.from_fasta('test_query.fa')

In [4]: target
Out[4]: DnaSequence(CCATACTGAACTGACTAAC)

In [5]: query
Out[5]: DnaSequence(ACTAGAATGGCT)

In [6]: alignment = pybio.alignment.smith_waterman(query, target,
    gap_open_penalty=0)

In [7]: alignment
Out[7]:
Alignment

target: TARGET
```

```
 query: QUERY

ACT-GAACT-GACT
ACTAGAA-TGG-CT

optimal alignment score: 10, suboptimal alignment score: 0
```

The `smith_waterman` function allows specifying the `match_matrix`, the `gap_open_penalty`, and the `gap_extend_penalty`. These tuning parameters have default values that depend on the input sequence. `DnaSequence`s and RnaSequences match the defaults for NCBI BLAST's `blastn` nucleotide blast algorithm, with a gap open penalty of 5 and a gap extend penalty of 2, and using the `blastn` default matrix. `ProteinSequence`s match the defaults for NCBI BLAST's `blastp` protein blast algorithm, with a gap open penalty of 11 and a gap extend penalty of 1, and using the BLOSUM62 matrix.

These and several other preset match matrices are available in `pybio.alignment.matrix`. A custom matrix can be input as a numpy array.

The `smith_waterman` function returns an `Alignment` object. This object stores the alignment scores, references to the query and target sequences, and information describing the alignment itself: the beginning and end of the alignment along the query and target sequences, and the alignment's CIGAR, a string describing the alignment matches, insertions, and deletions, as defined in the Sequence Alignment / Map (SAM) Format specification (Heng et al., 2009).

The `smith_waterman` function allows the file name of a FASTQ or FASTA formatted file as the target argument. The file may be uncompressed or compressed with Zip, gzip, or bzip2. When called with a file name, all of the alignments are returned.

```
In [2]: dna = pybio.DnaSequence.from_fasta('mito.nt.gz', 2, description
   ='*Gallus gallus*')
```

```
In [3]: alignments = pybio.alignment.smith_waterman(dna, '100k_illumina1
   .fastq')

In [4]: len(alignments)
Out[4]: 100000

In [5]: alignments[4]
Out[5]:
Alignment

target: M00785:2:000000000-A60JC:1:1101:16982:1552
 query: gi|71658078|ref|NC_007236.1|

CCCCCTGAACCTGA
CCCCCTGAACCTGA

optimal alignment score: 14, suboptimal alignment score: 0
```

Chapter 6: Sequences

The `Sequence` class is central to the design of PyBio. It is a base class to all of the biological sequence classes in Pybio: `DnaSequence`, `RnaSequence`, and `ProteinSequence`. Much of the functionality of the sequences is implemented directly in the `Sequence` class.

6.1. Class Structure

Like other class structures in PyBio, the structure of the sequence-related classes is extremely simple and flat: all sequence classes inherit from the `Sequence` class. None of the existing sequence classes has any other parent class.

6.2. Internal Representation

The sequence is implemented internally with a numpy `ndarray` of type `np.int8`. The alphabet used to convert from the values in the sequence array to a character string is kept as an ordered list of characters. The sequence is treated as immutable by the `Sequence` interface.

This representation was chosen for performance in operations on the sequence. Numpy `ndarray`s are implemented in C as homogeneous arrays with good locality of reference, and numpy's ufuncs are implemented in C, avoiding the looping through the sequence that occurs in interpreted Python. Because the sequence is immutable, extra copies are avoided. Some operations only require a different alphabet, avoiding both iterating through the sequence and an unnecessary copy.

These performance characteristics along with the speed and expressiveness of array programming open up the possibility of efficiently implementing bioinformatics algorithms in pure Python (or mostly Python with small sections in C), with all of the attendant benefits of accessibility and readability.

`ndarray`s also have benefits for interoperability with existing code. The underlying representation of an `ndarray` is a simple continguous C array, along with information of the data type and logical shape of the array. These simple arrays are easily accessible from the `ndarray` and are often directly usable by existing or new C code.

6.3. Sequence Manipulation

The `Sequence` class acts, in many ways, like an immutable Python `string`. A `Sequence` can be sliced using array slicing syntax, and a `Sequence` can be iterated, character by character. A `Sequence` has a length, accessible through the Python `len` keyword.

Slicing is implemented by overriding the `__getitem__` method. The method returns a new `Sequence` intialized with a slice of the internal sequence using numpy's array slicing.

```
    def __getitem__(self, slc):
        return self._from_ndarray(self._sequence[slc], self.
   _alphabet)
```

Iteration is implemented by overriding the `__iter__` method with a simple generator operating over the numpy `flat` iterator, with a conversion for each char using the objects internal alphabet.

```
    def __iter__(self):
        for char in self._sequence.flat:
            yield self._alphabet[char]
```

The length is a simple override of the `__len__` method to return the length of the internal sequence.

```
    def __len__(self):
        return len(self._sequence)
```

Two `Sequence` objects are considered equal if all characters in the sequence match, regardless of the alphabet or internal representation.

```
    def __eq__(self, other):
        if isinstance(other, self.__class__):
            return self.to_list() == other.to_list()
        else:
            return False

    def __ne__(self, other):
        return not self.__eq__(other)
```

A string representation of the `Sequence` is returned by the `to_string` method or the `string` property. The `to_list` method returns the sequence as a list of characters.

```
    def to_list(self):
        """Returns the sequence as a list of characters."""
        return [self._alphabet[value] for value in self._sequence]

    def to_string(self):
        """Returns the sequence as a string."""
        return ''.join(self.to_list())
```

6.4. Parsing

All `Sequence` classes can be initialized with a string, a list, or another `Sequence`.

`Sequence` also has four class methods that return a `Sequence` from a string or file: `from_fasta`, `sequences_from_fasta`, `from_fastq`, and `sequences_from_fastq`. These methods are implemented internally using the `Fasta` and `Fastq` classes in the `pybio.parse` module.

The file may be compressed with Zip, gzip, or bzip2. The compression type is deduced from examining the first few bytes of the file for a signature matching the format specification.

The `Fasta` and `Fastq` classes each inherit from a `namedtuple` with appropriate fields. The `Fasta` fields are `identifier`, `description`, and `sequence`; `Fastq` has these plus `quality_scores`.

The public methods of the classes are the same: the class method `parse_string` for parsing a string with one entry in FASTA or FASTQ format into a `Fasta` or `Fastq` object, the class method `parse_iterator` for parsing a file or string with multiple entries into an iterator of objects, and the `to_string` method, which returns the object as a string in FASTA or FASTQ format.

6.5. Transformations

`DnaSequence` and `RnaSequence` have several additional methods for transformations of the sequence.

6.5.1 Complement and Reverse Complement

Both `DnaSequence` and `RnaSequence` have the `complement` and `reverse_complement` methods. `complement` is implemented by creating a short numpy `ndarray` of type `np.int8` with the index for the conversion at each position. Then it broadcasts this index list over the entire sequence.

For example, in `DnaSequence`, the alphabet is `['A', 'C', 'G', 'T', 'N']`. In the sequence then, 'A' is 0, 'C' is 1, and so on. The complement is defined by a dictionary, `{'A':'T','T':'A','G':'C','C':'G'}`. The conversion index list is calculated to be [3 2 1 0 4], where each position lists the index that it will become after the conversion; 'A' at position 0 becomes 'T' at position 3. 'N' at 4 is unchanged.

`reverse_complement` simply computes the complement and then reverses the `Sequence` using a numpy slice.

6.5.2 Transcription and Reverse Transcription

`DnaSequence`s are converted to `RnaSequence`s with the `transcribe` method. `RnaSequence`s are converted to `DnaSequence`s with the `reverse_transcribe` method. Both of these pass through the sequence unchanged to the new `Sequence` object; only the alphabet is changed.

6.5.3 Translation

Both `DnaSequence` and `RnaSequence` implement a `translate` method. The `DnaSequence` `translate` method calls `transcribe` to create an `RnaSequence`, then calls the `translate` method on the `RnaSequence`.

The `RnaSequence` `translate` method works by looping through the sequence three characters at a time and using a dictionary to translate the codon triplet into an amino acid single letter code.

`translate` uses the Standard Genetic Code. I plan to expand `translate` to include additional codes in a future release.

Chapter 7: Entrez

The Entrez Programming Utilities, also called the E-utilities, are a set of server-side programs for accessing Entrez databases at NCBI through a web interface. There are nine separate utilities, covering search, record retrieval, and metadata (NCBI, 2014).

PyBio provides functions for accessing Entrez though the E-utilities in the `pybio.entrez` module. The module is oriented toward providing direct access to Entrez for a data scientist working at an IPython prompt. In this chapter, I describe the implementation of this module.

7.1. Entrez Usage Limits

NCBI asks user to limit their requests to fewer than three per second, and to run large jobs overnight and on weekends. PyBio does not police for large jobs, but it does automatically enforce the three requests per second limit, to be a good member of the community, and to simplify the job of PyBio users running scripts. PyBio takes care of the rate transparently for users.

All requests to NCBI are run through the `_entrez_get` and `_entrez_post` functions. These maintain a global `last_access` variable, and run `_ncbi_delay`. `_ncbi_delay` checks the time since the last call, and sleeps until one third of a second has passed since the last call.

This method avoids any delay if it has already been more than a third of a second since the last request to NCBI, and it shortens the delay for subsequent requests when other processing has taken time since the last request. This is in contrast to the system used in BioPython, which always imposes a third of a second delay.

The delay can be set manually with the `set_ncbi_delay` function. The function is not exposed in the main API, but can be reached through `pybio.entrez.entrez.set_ncbi_delay`.

7.2. `email` and `tool` values

All of the E-utilities can take `email` and `tool` parameters. These parameters help the NCBI to monitor usage of Entrez.

The NCBI may block ip addresses of users who abuse the usage limits (Sayers, 2010). If the `email` is included in these requests, NCBI can attempt to contact the user.

This email can be set for a session with the `set_email` function. PyBio does not require that the email be set in order to preserve complete generality in Entrez requests. However, a warning is logged if an Entrez request is made without running `set_email` or including an `email` argument in the request.

The `tool` parameter is used by NCBI to monitor tool usage. Tools can be registered with the NCBI to allow identifying and correcting excessive Entrez requests.

PyBio sends this parameter with every Entrez request. By default, the parameter is set to 'pybio'. A user can change this parameter with the `set_tool` function. While most users will never need to reset the `tool` parameter, it is potentially useful for the writers of applications and libraries that import PyBio's `entrez` module.

7.3. Low-level Wrappers for Entrez E-utilities

PyBio provides low-level wrappers for all nine of the Entrez E-utilities. By low-level, I mean that these are simple, general wrappers that give access to all of the functionality of the original tools from a Python command line without providing much additional help to the user. A user needs knowledge of the parameters of the underlying tools to make good use of these functions.

The wrappers do provide a few conveniences. It is no longer necessary to know how to handle http requests to use the Entrez E-utilities. The tools are wrapped in a documented Python function, with the parameters for the tool given as keyword arguments to that function. Required parameters are explicitly named as positional arguments in the function so that required parameters are manifest to the user. The wrappers enforce NCBI's request per second limit, and silently add the `email` and `tool` properties.

The functions have the same name as the corresponding Entrez E-utility, so, for example, `pybio.entrez.esummary` is the function corresponding to the Entrez ESummary utility. A short description of each tool is provided in the User Guide chapter.

These functions return a `Response` object from the Python `requests` module. This allows the user to check the url and status code of the request. The result itself is a string in the `text` property of the `Response` object.

The result format depends both on the tool and the parameters passed. Typically, this result is in XML or JSON, but other formats are possible; the EFetch tool can return sequences as text in FASTA format.

Using these tools involves knowing both the parameters to use in which tool, and how to parse the result text. Some tools are also subject to limits on the number of entries that can be returned.

These tools are the base API used by the rest of the module to access Entrez.

7.4. Id Search

The `esearch_id` function makes an Entrez ESearch query and returns the list of UIDs as a Python list. It is implemented using the `esearch` function. The resulting XML is parsed using `cElementTree` to extract the UIDs.

7.5. Entrez Pipelines

In *A General Introduction to the E-utilities* (Sayers, 2010), Eric Sayers notes that "The E-utilities... full potential is realized when successive E-utility URLs are combined to create a data pipeline." He lists several basic and advanced pipelines. PyBio implements two of these as functions.

ESearch$\rightarrow$ ESummary is implemented as `esearch_esummary`. This function performs an `esearch`, saves the resulting UIDs to the Entrez History server, then calls `esummary` on these UIDs and returns its `Response`. For the user, it means that a single call can return all summaries that match the search terms (up to the `retmax` parameter).

ESearch$\rightarrow$ EFetch is implemented as `esearch_efetch`. This function is implemented internally nearly identically to `esearch_esummary`, except that it uses `efetch` instead of `esummary`.

Both of the functions take advantage of the Entrez's history feature. The initial ESearch is called with the 'usehistory' parameter set to 'y', which instructs Entrez to save the resulting UIDs to the Entrez History server. A 'query_key' and 'WebEnv' are included within the XML returned by ESearch. These parameters are added to the following NCBI request, the `esummary` or `efetch` call, so that they use the saved UIDs rather than a set of UIDs passed as a parameter. This allows passing data between these two requests without returning the data to the local client, which cuts down transferred data and eliminates parsing of the ESearch result.

7.6. Sequences from Entrez

PyBio's highest level interface to Entrez is a set of functions for returning sequences from Entrez's 'nucleotide' and 'protein' databases as PyBio `Sequence` objects.

7.6.1 Sequences by Accession Number

Accession numbers are unique identifiers for biosequences with in a data store. They are often referenced in scientific articles.

`get_nucleotide_sequence` returns a `DnaSequence` object given an accession number. `get_protein` returns a `ProteinSequence` given an accession number.

Both of these functions are implemented using `esearch_efetch`, with the search term set to search for an accession number and to return the result in FASTA format. They raise an exception if there is an error in the search, or if nothing is found. The `Sequence.from_fasta` class method is used to create the returned `Sequence` object.

7.6.2 Sequences by Entrez ESearch

`search_nucleotide_sequences` and `search_proteins` return iterators over `Sequence` objects matching a search query.

The ESearch `term` parameter, which specifies the search criteria, is constructed from named arguments passed to the function. A colon is used to separate ranges. For example, `properties='biomol trna'` would return tRNA sequences and `sequence_length='1000:2000'` would return sequences with lengths between 1000 and 2000. The allowed keywords are listed in the function doc strings.

The iterator is implemented with a simple producer-consumer pattern. One thread makes requests to Entrez, using `esearch_efetch`, returning a set of entries as a string in FASTA format. A reference to this string is put on to a fixed-size queue, which is implemented using the `Queue` module. Requests are made, and the results are put on to the queue, until all of the requested records have been recieved, when a `None` object is placed on the queue.

Meanwhile, the main thread does a blocking `get` on the queue. Each FASTA formatted string from the queue is parsed into `Sequence` objects using the `Sequence.sequences_from_fasta` class method. The main thread continues pulling from the queue until it recieves `None`, indicating end of data.

This implementation has several advantages over simply making the request and parsing the resulting entries:

- The response time is shortened for the interactive user, allowing `Sequence` objects to be seen and processed within about a second, instead of waiting possibly several minutes for a response from the Entrez server.
- It gives an opportunity for a user in an IPython environment to see the live response of their program and make a decision to interrupt the function by hitting Control-C without returning every sequence from the server.
- There is a much smaller memory footprint.
- Throughput is increased for heavy data processing by not having processing wait on a long request time.

- Entrez limits on the number of entries returned from a single request are naturally accounted for, as each request is limited to size much smaller than the limit. This shifts the burden of accounting for size limits off of the user, and avoids having specialized code in the library for handling requests which may exceed the limit.

This implementation takes advantage of the Retrieval Parameters mentioned in the user guide. At each request to Entrez, the `retstart` parameter, which gives the sequential index of the first returned entry, is incremented by maximum number of entries requested per call.

The consumer-producer pattern is tuneable with the `queue_size` and `max_per_call` parameters. `queue_size` adjusts the size of the queue, limiting the number of outstanding Entrez requests yet to be processed. `max_per_call` adjust the maximum number of entries per request.

Chapter 8: Sequence Alignment

Sequence alignment is one of the most important sets of problems in bioinformatics algorithms. Many algorithms and variations of algorithms for sequence alignment have been implemented.

The `pybio.alignment` module provides an implementation of the Smith-Waterman algorithm and classes for working with sequence alignments. Along with its direct use, this implementation is meant to provide some general infrastructure for sequence alignments. The framework will help future expansions into other sequence alignments to maintain common interfaces and interoperability within the `pybio.alignment` module and throughout the library.

8.1. Design

The fundamental unit of organization in Python is the module rather than the class. Good style in Python encourages the use of plain functions in modules along with classes and associated methods for structuring data and data access.

Following this model, PyBio implements `smith_waterman` as a function on the `pybio.alignment` module. The function returns an `Alignment` object, which encapsulates an alignment between two sequences.

8.2. The `smith_waterman` function

The `smith_waterman` function performs the Smith-Waterman algorithm for local alignment between a query sequence and a target sequence, or between a query sequence and the sequences in a FASTA or FASTQ file.

Files are handled by calling the `pybio.Sequence.sequences_from_fasta` and `pybio.Sequence.sequences_from_fastq` methods that return iterators of `Sequence` objects from a compressed or uncompressed file. These methods are called on the type of the query `Sequence`, restricting the `Sequence`s returned by the iterator to match the query.

The `smith_waterman` function is implemented in Cython. The function ultimately calls into a module of the SSW Library, an optimized, open-source implementation of Smith-Waterman (Zhao et al., 2013).

8.2.1 SSW Library

The C code from the SSW library was written by Mengyao Zhao and released in 2013 (Zhao, 2016). Although the SSW library includes basic wrappers for C++, Python, and Java, PyBio only includes and makes use of a portion of the C code, with some small refactoring unique to PyBio.

The SSW library is a highly optimized, single-threaded implementation of Smith-Waterman, clocking in at about fifty times faster than a naive Smith-Waterman implementation in C, and on par with SSearch, using William Pearson's implementation of Smith-Waterman (Pearson & J, 1998), and Michael Farrar's original striped implementation (Farrar, 2006) (Zhao et al., 2013).

Like Farrar's implementation, the SSW library leverages single instruction multiple data (SIMD) on commodity x86 hardware, allowing processor level parallelization on each pairwise alignment. Unlike implementations based on field-programmable gate arrays (FPGAs), graphics processing units (GPUs), or parallelization on a cluster, the SSW implementation works on most common hardware.

Unlike Farrar's implementation, the SSW library returns information describing the alignment in addition to the alignment scores. This includes the beginning and end of the alignment on both the query and target sequences, and a CIGAR string descibing the alignment.

The SSW library is properly designed as a library rather than a monolithic application, making it easier to integrate into other C/C++ programs.

8.2.2 Cython Binding

PyBio integrates the SSW library with Python code using Cython. The `ssw.pxd` acts as a Cython header file, defining the functions and C structs available in Cython from the `ssw.h` header file of the SSW library. `smith_waterman.pyx` is the Cython source file that defines the `smith_waterman` function. Cython compiles this file into an intermediate C file, then into a shared object file or dynamic linked library, depending on the platform.

The Cython implementation of `smith_waterman` applies default values for the match matrix, the gap open penalty, and the gap extend penalty, based on the `Sequence` type of the query. Pointers to C arrays are extracted from the query sequence, the target sequence, and the match matrix. Using these pointers, the function then calls functions defined in `ssw.pxd` to perform the Smith-Waterman alignment. A CIGAR is extracted from the returned alignment, and an `Alignment` object is constructed and returned.

The SSW library represents the sequences as an array of eight-bit `ints`. This matches the internal representation of the NumPy arrays that are the internal representation of the PyBio `Sequence` object; the sequence inputs to the `ssw_init` and `ssw_align` functions from the SSW library used by PyBio can be found by calling the `data` method on the underlying NumPy sequence.

Both the `ssw_init` and `ssw_align` functions return a pointer to a struct. These must both be destroyed after extracting the CIGAR and `Alignment` using the `ssw.init_destroy` and `ssw.align_destroy` functions.

8.3. Classes

There are three classes in PyBio that provide infrastructure for alignments: `Cigar`, `CigarSegment`, and `Alignment`.

8.3.1 `Cigar` and `CigarSegment`

The CIGAR is a string that encodes the structure of a sequence alignment. The string format is defined in the Sequence Alignment / Map (SAM) Format specification (Heng et al., 2009).

A CIGAR string is composed of segments with an integer length and an op code: ‘M’ for match or mismatch, ‘I’ for an insertion, ‘D’ for deletion, ‘N’ for skipped bases, ‘S’ for soft clipping, ‘H’ for hard clipping, and ‘P’ for padding.

```
    taaAACGTTCAG-AGCaagggcacag
       |  || ||| | |
aattcggA--GTACAGAATCggacacgg
```

Figure 8.1: If the top string is the reference, the CIGAR describing this local alignment is 1M2D6M1I3M.

The individual segments of the CIGAR are represented in PyBio by the `CigarSegment` class, a `namedtuple` with a `length` and an `op`. The CIGAR itself is represented by the `Cigar` class, a `list` of `CigarSegments`.

8.3.2 `Alignment`

The `Alignment` class stores a reference to the query and target `Sequence`s, optimal and suboptimal alignment scores, the beginning and end of the alignment on the query and the target, and the CIGAR describing the alignment.

Chapter 9: Comparisons with Existing Systems

In this chapter, I present a comparison of PyBio with the two leading Python bioinformatics libraries, Biopython and scikit-bio.

9.1. Feature Comparison

All three libraries have similar features for working with biosequences and FASTA and FASTQ file formats. All three also include an implementation of an algorithm for optimal alignment. From there, the libraries go in different directions.

Biopython is the oldest and most comprehensive of the three libraries. Along with a comprehensive set of parsers for different biological formats and wrappers for third-party command line tools (installed separately), Biopython has support for phylogenetic trees, cluster analysis, sequence motifs, and access to NCBI's Entrez data retrieval system.

scikit-bio also has support for phylogenetic trees. It also includes some statistics libraries. Some of these library functions relate directly to biology while others are more general packages.

PyBio is more similar in construction to scikit-bio. Rather than require the user to locally install command line tools that are then wrapped by PyBio, it endeavors to provide complete solutions accessible directly after installing the library. Future expansion of PyBio will keep the focus on biology, excluding tools for statistics and machine learning that do not directly relate to biology.

	PyBio	scikit-bio	Biopython
Biosequence	✓	✓	✓
File formats	✓	✓	✓
Optimal alignment	Smith-Waterman in C	Smith-Waterman in C	Smith-Waterman in Python
Entrez	✓		✓
BLAST			wrappers
Multiple sequence alignment			ClustalW and MUSCLE wrappers
Phylogenetic trees		✓	✓
Biological diversity		✓	
Statistics package		✓	
Cluster analysis			✓
Sequence motifs		✓	✓

Table 9.1: A Partial Comparison of Features Among Python Bioinformatics Libraries

9.2. Supported File Formats

Biopython and scikit-bio both have support for many biological file formats. While many of these are simple wrappers that just parse a file or entry and store the various values, some are more involved. For example, the PDB file format (used to store three-dimensional structure data of biological molecules) support in Biopython allows not only simple parsing, but also several modes of iteration over the structure and functions to determine the distance between atoms or the nearest neighboring atoms.

At present, PyBio only supports FASTA and FASTQ formats.

9.3. Sequence Implementations

All three libraries have an object representing a biosequence. In Biopython, the sequence is implemented as a Python string.

In PyBio and scikit-bio (since version 0.4.0, released in July 2015), the sequence object is implemented internally as a numpy array. This implementation has major advantages over the Biopython implementation for speed and simplicity of code.

Use of sequences is fairly similar between the different packages. In all three libraries, sequences act like strings; they can sliced and iterated over.

9.3.1 scikit-bio Sequences

Like PyBio, scikit-bio has a `Sequence` class for general sequence data. There is a `GrammaredSequence` class that inherits from `Sequence` that stores a sequence with a character set. Then there are three classes that inherit from `GrammaredSequence` that represent the big three sequences of bioinformatics: `DNA`, `RNA`, and `Protein`.

The `DNA` class has three methods for transformations `transcribe`, `translate`,

	PyBio	scikit-bio	Biopython
FASTA	✓	✓	✓
FASTQ	✓	✓	✓
BLAST		✓	✓
Clustal		✓	✓
Newick		✓	✓
PHYLIP		✓	✓
QSeq		✓	✓
ABI			✓
ACE			✓
EMBL			✓
GenBank			✓
IntelliGenetics			✓
NEXUS			✓
PDB			✓
PHD			✓
PIR			✓
seqxml			✓
SFF			✓
Stockholm PFAM			✓
Swiss-Prot			✓
QUAL			✓
EMBOSS			✓
MAF			✓

Table 9.2: A Comparison of Supported File Formats Among Python Bioinformatics Libraries

and `translate_six_frames`, which translates across the six possible reading frames. There are no methods equivalent to PyBio's `complement` and `reverse_complement`.

The `RNA` class has three methods for transformations `reverse_transcribe`, `translate`, and `translate_six_frames`, which translates across the six possible reading frames. Again, there are no methods equivalent to PyBio's `complement` and `reverse_complement`.

9.3.2 Biopython Sequences

The root class for Biopython sequences is the `Seq` class. All `Seq` objects must have an alphabet.

Biopython does not subclass for DNA, RNA, and Proteins. Instead, an alphabet object from `Bio.Alphabet` is passed to the `Seq` constructor. For example, a DNA sequence is created by passing a dna alphabet.

```
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC
dna_seq = Seq("GATCGATGGGCCTATATAGGATCGAAAATCGC", IUPAC.unambiguous_dna)
```

Biopython sequence transformations are similar to PyBio. They are implemented as methods on the `Seq` class: `transcribe`, `translate`, `complement`, and `reverse_complement`. There is no `reverse_transcribe`.

9.4. Optimal Alignment Implementations

scikit-bio and PyBio both implement the Smith-Waterman algorithm by using Cython to interface efficiently with the SSW library. The details of the interface are quite different; the PyBio interface with the underlying C code is more simple and direct while providing more options and better integration with the library.

scikit-bio implements Smith-Waterman in the `skbio.alignment` module, using the `StripedSmithWaterman` class and the `query` function. The function returns an `AlignmentStructure` object, which wraps the results from the SSW library call.

```
from skbio.alignment import StripedSmithWaterman
query = StripedSmithWaterman("ACTAAGGCTCTCTACCCCTCTCAGAGA")
alignment = query("AAAACTCTAAACTCACTAAGGCTCTCTACCCCTCTTCAGAGAAGTCGA")
print alignment
ACTAAGGCTC...
ACTAAGGCTC...
Score: 49
Length: 28
```

The major differences between the two implementations are in the integration with the library. In scikit-bio, the Smith-Waterman implementation is completely independent of the scikit-bio `Sequence` class, requiring that a `Sequence` first be converted to a string, then converted internally into a C array implementation internally. The `AlignmentStructure` class is not aware of the scikit-bio `Sequence` class.

The PyBio `smith_waterman` function takes PyBio `Sequence` objects as arguments, and returns an `Alignment` object that references the query and target `Sequence`s directly. PyBio can skip the internal conversion, because the implementation of the Cython interface code in the `smith_waterman` function uses the same internal representation as the `Sequence` class.

Biopython also has an implementation of Smith-Waterman, under the `Bio.pairwise2` module. This is a pure Python implementation. Instead of using this implementation, the documentation suggests using Biopython's wrappers for the EMBOSS tools `water` and `needle` for Smith-Waterman and Needleman-Wunsch. This requires a separate local installation of EMBOSS and setting the system PATH.

Chapter 10: Summary and Conclusions

PyBio is the reflection of a set of software design principles, and a concrete tool that supports a small set of bioinformatics functionality. PyBio's range of functionality will have to grow for it to become a valuable tool, but it is today useful for a variety of bioinformatics tasks.

10.1. Lessons Learned

When I began this project, I focused on implementing bioinformatics algorithms. As the project advanced, I realized that I would need to focus more on usability, both for the user at an IPython prompt, and contributors writing extensions.

In particular, Entrez was not a part of the original proposal. I found that I often needed real biosequences to develop and validate functionality. These biosequences were a minor hassle to acquire; I had to search online for them, and often download very large files. I realized that not only would access to Entrez through Python be useful for me, but that acquiring sequences was a barrier to flow and the spirit of discovery that I wanted for PyBio.

That focus on usability also led to a shift toward providing a platform for integrating existing tools and encouraging the writing of new tools, rather than emphasizing developing the tools themselves.

That perspective also led to an interesting set of questions that PyBio will continue to answer as it matures. What is the best implementation of a particular algorithm? How can it be integrated with other tools in the PyBio environment? Can this implementation be improved? Can we write a better implementation - faster, cleaner code, a better interface? Answering these questions is part of PyBio's future.

10.2. Next Steps

The next major goal is to make PyBio accessible to outside contributors. The most important part here is to complete all of the existing pieces: better documentation, finalizing the code organization, ensuring that the setup is complete and general enough to accomodate new code. There will also have to be documentation for contributors to get them started with PyBio, and a set of issues and features to be done will need to be created on PyBio's github. And a certain amount of social engineering is required: putting up web pages, attending conferences, and giving talks.

Expanding functionality on `Sequence`s will be one of the major themes. Some of the obvious functionality to add are ORF finding, gene prediction, BLAST, and multiple alignment. Visual, interactive tools for the IPython Notebook are potentially powerful ways to interact with data, and they add to the accessibility of PyBio, lowering the programming sophistication and PyBio experience needed to get results from PyBio.

I would also like to expand PyBio's functionality in directions I find personally interesting, like *ab initio* protein and RNA structure prediction, rational protein design, and molecular dynamics simulation.

Maybe other contributors will do the same for their interests and expertise.

10.3. Final Thoughts

My hope for PyBio is that it be adopted by a community of contributors who will improve and expand it into a useful tool for the biology and bioinformatics communities.

New contributions could eventually give PyBio a role in standardization, with PyBio providing a standard set of tools that are recognized, understood, and univer-ally available.

PyBio's integration with NumPy and Cython may encourage programmers to develop bioinformatics tools largely in Python rather than C, so that the code and algorithms bioinformaticians use are accessible to people with less programming expertise, fostering tighter collaborations between programmers and biologists.

Whether or not PyBio is adopted beyond this thesis, writing PyBio has been a rewarding experience, and the lessons and design will continue to inform my work and view of bioinformatics and programming in general.

References

Altschul, S., Gish, W., Miller, W., Myers, E., & Lipman, D. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3), 403–410.

Bradshaw, R. & Behnel, S. (2015). Cython - c extensions for python. "`http://cython.org/`'.

Cock, P., Antao, T., Chang, J., Chapman, B., Cox, C., Dalke, A., Friedberg, I., Hamelryck, T., Kauff, F., Wilczynski, B., & de Hoon (2009). Biopython: freely available python tools for computational molecular biology and bioinformatics. *Bioinformatics*, 25(11), 1422–1423.

Cock, P., Fields, C., Goto, N., Heuer, M., & Rice, P. (2010). The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic Acids Research*, 38(6), 1767–1771.

Dumbill, E. (2005). On the success of ruby on rails. Retrieved from `http://www.oreillynet.com/network/2005/08/30/ruby-rails-david-heinemeier-hansson.html`.

Farrar, M. (2006). Striped smith-waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2), 156–161.

Gehring, W. J., Affolter, M., & Burglin, T. (1994). Homeodomain proteins. *Annual Review of Biochemistry*, 63, 487–526.

Gibas, C. & Jambeck, P. (2001). *Developing Bioinformatics Computer Skills*. Sebastopol, CA: O'Reilly.

Gorelick, M. & Ozsvald, I. (2014). *High Performance Python: Practical Performant Programming for Humans*. Sebastopol, CA: O'Reilly.

Heng, L., Handsaker, B., Wysoker, A., Fennel, T., Ruan, J., Homer, N., Marth, G., Abecasis, G., & Durbin, R. (2009). The sequence alignment/map format and samtools. *Bioinformatics*, 25(16), 2078–2079.

iMatix Corporation and Contributors (2014). Code connected - zeromq. "`http://zeromq.org/`'.

IUPAC-IUB Joint Commission on Biochemical Nomenclature (1984). Nomenclature and symbolism for amino acids and peptides. recommendations 1983. *Biochemical Journal*, 2, 345–373.

Johansson, R. (2015). *Numerical Python*. New York, New York: Appress.

Jones, N. & Pevzner, P. (2004). *An Introduction to Bioinformatics Algorithms*. Cambridge, Massachusetts: MIT Press.

Jupyter Project (2015). Jupyter project. "`https://jupyter.org/`'.

Kernighan, B. & Ritchie, D. (1988). *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice Hall, 2 edition.

Kuchling, A. (2005). Pep 206 – python advanced library. "`https://www.python.org/dev/peps/pep-0206/`".

Lipman, D. & Pearson, W. (1985). Rapid and sensitive protein similarity searches. *Science*, 227(4693), 1435–41.

Lutz, M. (2010). *Programming Python*. Sebastopol, CA: O'Reilly.

Lutz, M. (2013). *Learning Python*. Sebastopol, CA: O'Reilly.

McKinney, W. (2012). *Python for Data Analysis*. Sebastopol, CA: O'Reilly.

Mount, D. (2004). *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor, New York: Cold Spring Harbor Laboratory Press.

NCBI (2014). Entrez help [internet]. `"http://www.ncbi.nlm.nih.gov/books/NBK3837/"`.

Needleman, S. B. & Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3), 443–53.

NumPy Developers (2013). Numpy. `"http://www.numpy.org/`'.

Pearson, W. & J, L. (1998). Improved tools for biological sequence comparison. *Proceedings of the Natural Academy of Sciences of the United States of America*, 85(8), 2444–2448.

Pérez, F. & Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3), 21–29.

Perkel, J. (2015). Programming: Pick up python. *Nature.*, 518, 125–126.

Python Software Foundation (2015). Welcome to python. `"https://www.python.org/"`.

Ritchie, D. (1993). The development of the c language. In T. Bergin & R. Gibson (Eds.), *History of Programming Languages-II* New York: ACM Press.

Sayers, E. (2010). A general introduction to the e-utilities. in: Entrez programming utilities help [internet]. "`http://www.ncbi.nlm.nih.gov/books/NBK25497/`".

Sayers, E. (2015). The e-utilities in-depth: Parameters, syntax and more. "`http://www.ncbi.nlm.nih.gov/books/NBK25499/`".

scikit-bio developers (2015). scikit-bio. "`http://scikit-bio.org`".

Scopatz, A. & Huff, K. (2015). *Effective Computation in Physics*. Sebastopol, CA: O'Reilly.

Smith, T. F. & Waterman, M. S. (1981). Identification of common molecular subsequences. *Journal of Molecular Biology*, 147, 195–197.

Vanjari, S., Chimandare, N., & Gandhi, S. (2012). A review on in silico approach in pharmacology. *Advance Research in Pharmaceuticals and Biologics*, 2(2), 129–141.

Zhao, M. (2016). Ssw library: An simd smith-waterman c/c++/python/java library for use in genomic applications. "`https://github.com/mengyao/Complete-Striped-Smith-Waterman-Library`".

Zhao, M., Wan-Ping, L., Garrison, E., & Marth, G. (2013). Ssw library: An simd smith-waterman c/c++ library for use in genomic applications. *PLOS One*.

Ziade, T. (2008). *Expert Python Programming*. Olton, United Kingdom: Packt Publishing Ltd.