bio_utils Documentation

Release 1.1.2

Alex Hyer

Sep 07, 2018

Contents

Authors Alex Hyer, William Brazelton, Christopher Thornton

Date Sep 07, 2018

Version 1.0

Software library containing common bioinformatic functions

CHAPTER 1

Summary

bio_utils is a library of Python modules performing routine functions in bioinformatic scripts.

CHAPTER 2

Introduction

Many bioinformatic scripts perform similar or identical tasks internally during the execution of a program. Such tasks include reading FASTA files or filtering BLAST+ results. bio_utils contains numerous functions that quickly and simply handle simple, mundane, everyday tasks in a streamlined and simple fashion to save developers time. bio_utils aims to be as simple as possible, providing functionality without adding in unnecessary features, i.e. bio_utils is as vanilla as reasonable. This both increases the speed of most functions and greatly simplifies APIs.

Many libraries, such as SCREED and Biopython, already provide importable functions that execute these simple tasks. SCREED maintains a fairly simple API but is fairly slow and no one has updated the original repo since 2012-06-17. Numerous developers actively maintain Biopython and it is quite a bit faster, in some regards, than SCREED. However, Biopython stocks their functions with an enormous number of features that, while useful in a Python interpreter, are often ignored by programs or can be accomplished more quickly using built-in Python features, i.e. Biopython is bloatware to many developers. As aforementioned, bio_utils' vanilla design overcomes both these libraries issues by providing both simplicity and speed.

At this point in time, bio_utils is quite small and its scope limited. The authors intend to slowly but surely increase this library's repertoire over time. We welcome any and all contributions to our project.

CHAPTER 3

Installation

```
pip install bio_utils
```

CHAPTER 4

Contents

4.1 Classes

4.1.1 Introduction

bio_utils' offers many classes that house biological data from many different file formats. Each instance contains an attribute per field of a given file format as well as a `write()` method that returns the original entry properly formatted and followed by a newline character. Each section below includes a simple description of what each format contains and a link to a detailed description of said format.

These classes are currently contained in the iterators subpackage but will constitute their own package later, see Roadmap for details.

4.1.2 B6Entry

B6 files contain various data detailing the length and quality of an alignment between nucleotide or protein sequences. This file format is used by NCBI BLAST+ as output format 6, hence B6 (Blast+ 6). B6 was referred to as M8 in NCBI BLAST. drive5 contains a good, succinct description of this format. This class only supports the default B6 format and does not accept arbitrary fields.

4.1.3 FastaEntry

FASTA files contain nucleotide and protein sequences differentiated by unique identifiers. Wikipedia provides both the history of the FASTA format and format specifications.

4.1.4 FastqEntry

FASTQ files contain nucleotide and protein sequences differentiated by unique identifiers, like *FASTA* files. FASTQ files also contain quality scores indicating the confidence of each base or residue declaration. Wikipedia provides a description of the FASTQ format and the meaning of the various quality scores.

4.1.5 GFF3Entry

General Feature Format 3 (GFF3) contain various data on the location, type, and quality of a nucleotide or protein sequence annotation. As opposed to previous versions, GFF3 support an arbitrary number of hierarchical annotation levels. GMOD gives a very detailed walkthrough of this format.

4.1.6 SamEntry

Sequence Alignment/Map (SAM) files contains details on the location and quality of an alignment. Many alignment programs produced SAM files as their default output. GitHub host a painfully detailed description of the SAM file format.

4.2 Iterators

4.2.1 Introduction

The bio_utils' iterators subpackage contains numerous iterators for a variety of biology-relevant file types. The contained iterators share numerous features listed in the next section. The subsequent sections detail iterator-specific elements. Each iterator returns file-specific Python classes. This page only details how to use the iterators not the instances they return, see Classes for details on each class in bio_utils.

4.2.2 Common Features

Abilities shared by all iterators in this package include:

- Return file-specific instances, see Classes for more details
- Support the `next()` and `__next__()` methods
- Usable in *for* loops
- Accept any iterator: `iter()` objects, file handles, memory files, etc.
- Begin iteration at arbitrary lines using *start_line* or *header* argument

All iterators in this package are quite fast; the following table compares SCREED's, Biopython's, and bio_utils' FASTA iteration speed in seconds on a 3.2 GB FASTA file containing 1,614,108 sequence entries averaging 158 bases each.

FASTA File Iteration Speed for Different Libraries (seconds)						
Library	Trial#1	Trial#2	Trial#3	Trial#4	Trial#5	Average
SCREED	974.812	971.185	953.027	961.247	971.924	966.439
Biopython	581.148	575.677	585.322	548.876	516.235	561.452
bio_utils	290.876	298.047	291.625	291.966	289.635	292.430

4.2.3 b6_iter

Iterates over B6 alignment files and returns each line as an instance of *B6Entry*. This iterator does not support arbitrary table modifications as does BLAST+.

4.2.4 fasta_iter

Iterates over a FASTA file and returns each entry as an instance of *FastaEntry*. This iterator can handle sequences spanning multiple lines.

4.2.5 fastq_iter

Iterates over a FASTQ file and returns each entry as an instance of *FastqEntry*. This iterator can handle sequences and quality score spanning multiple lines.

4.2.6 gff3_iter

Iterates over a GFF3 file and returns each line as an instance of *GFF3Entry*.

4.2.7 sam_iter

Iterates over a SAM file and returns each as line as an instance of *SamEntry*.

4.3 Verifiers

4.3.1 Introduction

The bio_util's verifiers subpackage contains numerous functions that verify the data of a biological file format, i.e. they ensure a given file is properly formatted. These function check file entries against a regex matching a given file format. If the match fails, the verifier will subdivide the entry and determine what part of the entry fails the regex. This investigation of the entry permits the verifiers to return detailed error messages on what and where the file failed. Each verifier except *entry_verifier* is also a program with the simple syntax

[file]_verifier <file>

which simply reads through a file and prints whether ot not it is valid.

4.3.2 entry_verifier

The guts of the verifiers package, this versatile function matches a string to a regex. If the match fails, `entry_verifier()` splits both the regex and string by a given delimiter and matches each regex fragment to its corresponding string fragment. When a string fragment fails, a custom `FormatError` containing details on the failure is raised.

4.3.3 b6_verifier

Verifies the validity of a list of *B6Entry*.

4.3.4 binary_guesser

Heuristically guess whether a file is binary or text. While not technically a "verifier", this function fits in this subpackage well as it helps confirm a generic property of the file before use by a program.

4.3.5 fasta_verifier

Verifies the validity of a list of *FastaEntry*.

4.3.6 fastq_verifier

Verifies the validity of a list of *FastqEntry*.

4.3.7 gff3_verifier

Verifies the validity of a list of *GFF3Entry*.

4.3.8 sam_verifier

Verifies the validity of a list of *SamEntry*.

4.4 Blast Tools

4.4.1 Introduction

The bio_utils' blast_tools subpackage contains a few tools for making BLAST output easier to work with. They are from extensive but what they do provide may be useful to some developers.

4.4.2 blast_to_cigar

A simple function that converts the query sequence, subject sequence, and midline fields of BLAST+ XML output (M7) to a CIGAR string. This function supports both the newer and older versions of CIGAR strings.

4.4.3 b6_evalue_filter

This function iterates through any iterator yielding lines of a *B6/M8* file. This iterator only returns the lines above an E-value threshold as *B6Entry*.

4.4.4 query_sequence_retriever

Retrieve the aligned query sequence for each alignment in a B6 file above an E-value threshold.

4.4.5 subject_sequence_retriever

Identical to *query_sequence_retriever* except it returns subject sequences.

4.5 Contributing

4.5.1 Introduction

Due to the *potential* size of this project and contributors thereto, it is helpful to have an underlying, general philosophy concerning what should be included in bio_utils, how it should be coded, and where it should go within the library. This ensures that everything is structured logically and that the library is both intuitive to use and internally consistent. This document explains these philosophies—and their implementations—and should be read by anyone looking to contribute to the library on whether or not their script belongs in bio_utils, where it belongs, and how to structure their script.

4.5.2 How to Contribute

This section is placed near the top for your convenience, if this is your first time contributing, please read the rest of this document first.

Start-Up Preparation

Follow these steps if you have never contributed to bio_utils before:

1. Install the following software:
    - Git
    - Sphinx
    - pytest
    - gitchangelog
2. Get a GitHub account
3. Fork the repo ('fork' button)
4. Clone your copy of bio_utils:

```
git clone https://github.com/<your GitHub account username>/bio_utils.git
```

5. Connect the "official" bio_utils repo to your fork:

```
git remote add braz https://github.com/Brazelton-Lab/bio_utils.git
```

Actually Contributing

Before editing anything, run the following command to make sure your copy of bio_utils is up-to-date with the "official" branch:

```
git pull braz master
```

After editing or adding files, perform the following steps:

1. Write a unit test if you wrote a new script.
2. Run unit tests and do not continue with this workflow unless they pass:

```
cd <main directory of git repo>
py.test
```

3. Add files to your git commit:

```
git add <file>
```

4. Update ChangeLog.rst:

```
gitchangelog > ChangeLog.rst
```

5. Commit your changes, add a detailed message in whatever text editor appears:

```
git commit
```

6. Push your commit to GitHub:

```
git push origin
```

7. Repeat steps 1-6 until you want to merge your changes to the "official" repo
8. To merge your changes, go to you GitHub copy of bio_utils and click 'Compare & pull request'
9. In your pull request mention @TheOneHyer for review

4.5.3 Core Principles

Simple-to-Use/Intuitive

A software library is analogous to a toolkit. Like a real-world toolkit, it should be organized so that any single tool can be found without any real effort. Once a tool is found, it's function should be obvious based on it's name, e.g. a screwdriver drives a screw through material. To translate this analogy to computer terms, each script should be a single tool whose name reflects its function.

ASAP (As Simple As Possible)

A screwdriver drives a screw, a hammer applies normal force, and a wrench applies torque. Each of these tools' use is fairly singular and unique. Each script in a library should have a single, unique function. To say that in more confusing terms: each tool should do one thing and one thing only, and said thing shall not have already been done. Additionally, each script should be written in a straightforward manner that is easy to read and understand; i.e. each script should be Pythonic.

Heavily Documented

A library is used by developers writing a program. This obvious statement should lead to an obvious conclusion: a library is of no use if a programmer doesn't know/can't find out what any given tool does. Thus, each script should be very documented heavily. While, it is better to document more than less, there is such a thing as too much documentation. There comes a point where documentation is too repetitive and confuses the reader more than helping them; avoid this.

Logically Organized

This point was basically made with the first Core Principle but will be re-iterated/elaborated on here. When opening a toolkit, it should be easy to figure out where a tool is because "tools of a feather should be stored together". Therefore, a library should have a hierarchical structure grouping similar tools. All tools fit into at least some category so there should never be a "miscellaneous" category.

Up-to-Date

An outdated library is not particularly useful. What constitutes an outdated library is quite subjective as some tools may never need updating after they're initially written. However, this is not normally the case as new functions become desirable and various standards change. An up-to-date library should always support the latest file conventions and adhere to current coding standards withing both the library and the library's coding language.

Fast

The whole point of having a library is to save time and effort by pre-writing tools and making them easily accessible. If the tools are slow for their given task, they are essentially useless. Scripts should be written so as to minimize resource usage and maximize speed. Occasionally, a script that runs fast is not ASAP and vice versa. If such a conflict arises, attempt to find an optimum intermediate favoring ASAP over speed.

4.5.4 Script Requirements

This section is as a practical coding guide to implement the above principles in bio_utils' scripts.

Follow PEP!

Python already has multiple cannon code style guides, versioning systems, etc.; follow them! There are many PEPs so it can be hard to keep track of them all. IDEs such as PyCharm will take care of PEP formatting for you. Several critical PEPs are also linked below:

- PEP 0007 - C Code Style Guide
- PEP 0008 - Python Code Style

Guide * PEP 0440 - Versioning and Specifying Dependencies

Only One or a Few Related Importable Functions per Script

Each script should only contain a few related functions at most. Most scripts should only have a single function. This helps keep everything logically and hierarchically organized. Let's go over a simple example to demonstrate the benefits of this approach:

One could write a script called fasta.py that contains all library functions dealing with FASTA files. This seems convenient because if a developer wants to do something with a FASTA file, s/he only needs to look at one script to see if the functionality they want exists. However, they have to open and look through the whole file to see if what they want exists. Also, they can't get any information on categorical functions such as what iterators available in bio_utils. By creating a sub-package called "iterators" in bio_utils and placing a fasta.py script containing a single function (iterating through a FASTA file), a developer can see what iterators are available and understand the function of fasta.py at the same time without needing to open a file! Also, by placing a fasta.py in each appropriate sub-package (with the package- corresponding functionality), a developer can simply search for files named fasta.py to

glean everything they can do with a FASTA file in bio_utils. This also makes imports more obvious and clear to a reader.

If multiple functions are all related to a single "thing" within the same sub-package, then it is appropriate to include multiple functions in a single script. Doing so is simply a best judgement call.

No "End" Functions

As aforementioned, a software library is analogous to a toolbox. To that end, each script should perform and return data but never execute an ultimatum. The developer needs to have maximal control over their script; they should not have to worry about tools manipulating the flow of their program. As an example, all functions in the verifiers sub-package used to exit the program if a file could not be validated because it assumed that if a file was incorrectly formatted, the program calling it would crash downstream. This assumption is not always valid and such a drastic change in program flow should never be assumed. Now all the verifiers raise a `FormatError` if a file is invalid. This allows programmers let a script crash with the error or catch and continue.

Since each script or function must act as a means and not an end, they **MUST** return something. There is no such thing as a silent function call in bio_utils.

In summary, scripts in bio_utils should never print to screen, exit the program, or elsewise do anything a developer cannot control and must return something. Scripts can raise errors.

No Command-Line Programs

This section is somewhat related to the last, i.e. bio_utils is a library and not an end product. As such, there are no standalone programs as that would constitute an end goal. There is, however, one exception: if the function in the script can be logically and simply transformed into a standalone program, then it should be made into one. As an example, each of the verifiers double as command-line programs that take a single file as their only argument and print whether or not the file is properly formatted. When a script in bio_utils doubles as a program, it should:

1. Simply call it's own importable function
2. **The program should support the following (if applicable):**
    - Reading and writing compressed files
    - Piping
    - One or zero positional arguments

Docstrings for Each Script, Class, AND Function

Each individual document in bio_utils should be documented with docstrings and inline comments as appropriate. More specifically, each docstring should have a synopsis line, document arguments, and returns as per Google Function Definitions . If appropriate, the docstrings should also include a more thorough description of the function. Each script, *even those only containing a single function or class*, should also have docstrings. If the script contains one function or class, the docstring can simply be a one-liner about the function and copyright information. If the script has multiple functions or classes, the docstrings should include a synopsis of what the script offers and a one-liner about each function. Full API should also be described in our Sphinx

Metadata and Copyright

All scripts must start with the following code:

```
#! /usr/bin/env python

# from __future__ imports go here

"""<one-liner describing software>

<whatever you want here>

Copyright:

    <program name> <one-liner describing software>
    Copyright (C) 2015  William Brazelton, Alex Hyer

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>.
"""

__author__ = '<authors>'
__email__ = '<email of lead author or maintainer>'
__license__ = 'GPLv3'
__maintainer__ = '<maintainer of script>'
__status__ = '<production level of script>'
__version__ = '<script version>'
__credits__ = '<credit for legally borrowed code if appropriate>'

# imports, then rest of script
```

4.5.5 Sub-Package Requirements

This section details what sub-packages in bio_utils contain and when it is appropriate to start a new one.

2+ Scripts per Package

While you can have a package with just a single script, try for at least two scripts per package. The reasoning behind this is simple, a package with a single script feels like an unnecessary "package" in the import statement and chances are you can think of a second script useful to the concept driving the sub-package. If a script truly doesn't fit in any other sub-package, try to think of a second function fitting the schema and code it. bio_utils will never have a "misc" package.

Import at Package Level

Since each script should have only one (or a few functions), they should be imported at the package level—in the "__init__.py" file—so that a programmer doesn't have to write redundant words in import statements. For example:

from bio_utils.iterators import sam_iter (package level = better)

from bio_utils.iterators.sam import sam_iter (file level = worse)

Own Documentation Page

Each sub-package must have its own web page in the documentation following this format:

```
=====
Title
=====

.. automodule:: <module>

Introduction
------------

<what package contains and any globally relevant information>

<optional sections>

<first function>
----------------

<short function description, should be longer/give more info than function
one-liner>

.. autofunction:: <function>
```

4.6 Roadmap

4.6.1 Introduction

This page details planned updates and features for future versions of bio_utils. There is not a timetable for any of these upgrades. The following changes may themselves change frequently and drastically.

4.6.2 V1.1

- Give all iterators an argument to verify each entry before returning
- Added iterator for forward and reverse reads

4.6.3 V1.2

- Create a new package for file tools
- Add versatile compression reader based on http://stackoverflow.com/a/13044946/1585509
- Add versatile compression writer

4.6.4 V2.0

This major version change will include many changes that make bio_utils API more intuitive and better organized. While some changes seem smaller, like renaming packages, they constitute a major as they will break current API and not be backwards compatible.

- Rename iterators package to parsers, many doc changes
- Moving classes in iterators to a new "data storage structures" package

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

CHAPTER 6

Copyright

bio_utils' operates under the `GPLv3 License` and may be edited and redistributed as per that license.

Python Module Index

b

Index

B