pybedtools Documentation

Release 0.2.3dev

Ryan Dale

May 20, 2011

CONTENTS

CHAPTER ONE

OVERVIEW

pybedtools is a Python wrapper for Aaron Quinlan's BEDtools and is designed to leverage the "genome algebra" power of BEDtools from within Python scripts.

This documentation is written assuming you know how to use BEDTools and Python.

See full online documentation, including installation instructions, at http://pybedtools.genomicnorth.com.

Note: `pybedtools` is still very much in progress. Please keep that in mind when assesing whether to use this package in production code.

CHAPTER

TWO

CONTENTS:

2.1 Installation

2.1.1 Requirements

- argparse (unless you're running Python 2.7, which comes with `argparse` already)
- Cython - part of `pybedtools` is written in Cython for speed
- BEDTools

Both argparse and Cython can be installed with pip:

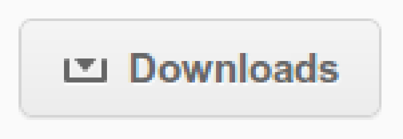
```
pip install cython argparse
```

To use `pybedtools` you'll need the latest version of the package and the latest version of BEDTools.

2.1.2 Installing `pybedtools`

To install the latest version of `pybedtools` you have 2 options:

Option 1: install from source

- go to http://github.com/daler/pybedtools
- click the Downloads link ()

- choose either a `.tar.gz` or a `.zip` file, whatever you're comfortable with
- unzip into a temporary directory
- from the command line, run:

  ```
  python setup.py install
  ```

  (you may need admin rights to do this)

Option 2: use pip to automatically download the latest stable version from the Python Package Index:

```
sudo pip install --upgrade pybedtools
```

You can run the tests with:

```
sh test.sh
```

Note, however, that you will need to have `nosetests` installed in order to run the tests, e.g.,:

```
pip install nosetests
```

2.1.3 Installing BEDTools

To install BEDTools,

- follow the instructions at https://github.com/arq5x/bedtools to install
- make sure all its programs are on your path

2.2 Three brief examples

Here are three examples to show typical usage of `pybedtools`. More info can be found in the docstrings of `pybedtools` methods and in the *Tutorial Contents*. Before running the examples, you need to import `pybedtools`:

```
>>> from pybedtools import BedTool, cleanup
```

After running the examples, clean up any intermediate temporary files with:

```
>>> cleanup()
```

2.2.1 Example 1: Save a BED file of intersections, with track line

This example saves a new BED file of intersections between `a.bed` and `b.bed`, adding a track line to the output:

```
>>> a = BedTool('a.bed')
>>> a.intersect('b.bed').saveas('a-and-b.bed', trackline="track name='a and b' color=128,0,0")
```

2.2.2 Example 2: Intersections for a 3-way Venn diagram

This example gets values for a 3-way Venn diagram of overlaps. This demonstrates operator overloading of `bedtool` objects:

```
>>> # set up 3 different bedtools
>>> a = bedtool('a.bed')
>>> b = bedtool('b.bed')
>>> c = bedtool('c.bed')

>>> (a-b-c).count()   # unique to a
>>> (a+b-c).count()   # in a and b, not c
>>> (a+b+c).count()   # common to all
>>> # ... and so on, for all the combinations.
```

2.2.3 Example 3: Flanking sequences

This example gets the genomic sequences of the 100 bp on either side of features.

The `bedtool.slop()` method automatically downloads the `chromSizes` table from UCSC for the dm3 genome, but you can pass your own file using the standard BEDTools `slop` argument of `g`. Note that this example assumes you have a local copy of the entire dm3 genome saved as `dm3.fa`.

```
>>> # set up bedtool
>>> mybed = bedtool(’in.bed’)

>>> # add 100 bp of "slop" to either side.  genome=’dm3’ tells
>>> # the slop() method to download the dm3 chromSizes table from
>>> # UCSC.
>>> extended_by_100 = mybed.slop(genome=’dm3’, l=100, r=100)

>>> # Delete the middle of the now-200-bp-bigger features so
>>> # all we’re left with is the flanking region
>>> flanking_features = extended_by_100.subtract(’in.bed’)

>>> # Assuming you have the dm3 genome on disk as ’dm3.fa’, save the
>>> # sequences as a new file ’flanking.fa’
>>> seqs = flanking_features.sequence(fi=’dm3.fa’).save_seqs(’flanking.fa’)

>>> # We could have done this all in one line
>>> # (this demonstrates "chaining" of bedtool objects)
>>> bedtool(’in.bed’).slop(genome=’dm3’,l=100,r=100).subtract(’in.bed’).flanking_features.sequence(fi
```

For more, continue on to the *Tutorial Contents*, and then check out the *Topical Documentation*.

2.3 Tutorial Contents

2.3.1 Intro

This tutorial assumes that

1. You know how to use BEDTools (if not, check out the BEDTools documentation)
2. You know how to use Python (if not, check out some tutorials like Learn Python the Hard Way)

A brief note on conventions

Throughout this documentation I’ve tried to use consistent typography, as follows:

- Python variables and arguments, as well as filenames look like this: `s=True`
- Methods, which are often linked to documentation look like this: `BedTool.merge()`.
- Arguments that are passed to BEDTools programs, as if you were on the command line, look like this: `-d`.
- The “>>>” in the examples below indicates a Python interpreter prompt and means to type the code into an interactive Python interpreter like IPython or in a script. (don’t type the >>>)

Onward!

2.3.2 Create a BedTool

First, follow the *Installation* instructions if you haven't already done so to install both BEDTools and pybedtools.

Then import the pybedtools module:

```
>>> import pybedtools
```

This documentation will use example files that ship with pybedtools, but you may find it more useful to use your own files. To create a BedTool you need to specify a filename. To get a list of example files that ship with pybedtools:

```
>>> pybedtools.list_example_files()
['a.bed', 'b.bed', 'c.gff', 'd.gff', 'dm3-chr2L-5M-invalid.gff.gz', 'dm3-chr2L-5M.gff.gz', 'dmel-all-
```

Note that there are BED and GFF files, some of which are compressed. All files supported by BEDTools are supported by pybedtools.

Using one of these filenames, get the full path to the file (which will depend on your operating system and how you installed the package) with:

```
>>> full_path = pybedtools.example_filename('a.bed')
```

Once you have a filename (whether it's an example file or your own file), creating a BedTool is easy:

```
>>> # create a new BedTool using that filename
>>> a = pybedtools.BedTool(full_path)
```

Let's set up a second BedTool so we can do intersections and subtractions. This time, we'll use a convenience function for creating BedTool instances out of example files (if you're using your own files, just make another one the same way a was made above).

```
>>> # create another BedTool to play around with
>>> b = pybedtools.example_bedtool('b.bed')
```

See *Creating a BedTool* for more information, including making BedTool objects directly from strings and iterators.

2.3.3 Intersections

One common use of **BEDTools_** and pybedtools is to perform intersections.

First, create some example BEDTool instances if you haven't already done so:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> b = pybedtools.example_bedtool('b.bed')
```

Then do the intersection with the BedTool.intersect() method:

```
>>> a_and_b = a.intersect(b)
```

a_and_b is a new BedTool instance. It now points to a temp file on disk, which is stored in the attribute a_and_b.fn; this temp file contains the intersection of a and b.

We can either print the new BedTool (which will show ALL features – use with caution if you have huge files!) or use the BedTool.head() method to show up to the first N lines (10 by default). Here's what a, b, and a_and_b look like:

```
>>> a.head()
chr1    1   100 feature1    0   +
chr1    100 200 feature2    0   +
chr1    150 500 feature3    0   -
chr1    900 950 feature4    0   +

>>> b.head()
chr1    155 200 feature5    0   -
chr1    800 901 feature6    0   +

>>> a_and_b.head()
chr1    155 200 feature2    0   +
chr1    155 200 feature3    0   -
chr1    900 901 feature4    0   +
```

The `BedTool.intersect()` method simply wraps the **BEDTools_** program `intersectBed`. This means that we can pass `BedTool.intersect()` any arguments that `intersectBed` accepts. For example, if we want to use the `intersectBed` switch `-u` (which acts as a True/False switch to indicate that we want to see the features in `a` that overlapped something in `b`), then we can use the keyword argument `u=True`, like this:

```
>>> # Intersection using the -u switch
>>> a_with_b = a.intersect(b, u=True)
>>> a_with_b.head()
chr1    100 200 feature2    0   +
chr1    150 500 feature3    0   -
chr1    900 950 feature4    0   +
```

This time, `a_with_b` is another `BedTool` object that points to a different temp file whose name is stored in `a_with_b.fn`. You can read more about the use of temp files in *temp principle*. More on arguments that you can pass to `BedTool` objects in a moment, but first, some info about saving files.

2.3.4 Saving the results

If you want to save the results as a meaningful filename for later use, use the `BedTool.saveas()` method. This also lets you optionally specify a trackline for directly uploading to the UCSC Genome Browser, instead of opening up the files afterward and manually adding a trackline:

```
>>> c = a_with_b.saveas('intersection-of-a-and-b.bed', trackline='track name="a and b"')
>>> print c.fn
intersection-of-a-and-b.bed

>>> print c
track name="a and b"
chr1        155     200     feature2        0       +
chr1        155     200     feature3        0       -
chr1        900     901     feature4        0       +
```

Note that the `BedTool.saveas()` method returns a new `BedTool` object which points to the newly created file on disk. This allows you to insert a `BedTool.saveas()` call in the middle of a chain of commands (described in another section below).

2.3.5 Default arguments

Recall that we passed the `u=True` argument to `a.intersect()`:

```
>>> a_with_b = a.intersect(b, u=True)
```

While we're on the subject of arguments, note that we didn't have to specify `-a` or `-b` arguments, like you would need if calling `intersectBed` from the command line. That's because `BedTool` objects make some assumptions for convenience.

We could have supplied the arguments `a=a.fn` and `b=b.fn`:

```
>>> another_way = a.intersect(a=a.fn, b=b.fn, u=True)
>>> assert another_way == a_with_b
```

But since we're calling a method on `a`, `pybedtools` assumes that the file `a` points to (specifically, `a.fn`) is the one we want to use as input. So by default, we don't need to explicitly give the keyword argument `a=a.fn` because the `a.intersect()` method does so automatically.

We're also calling a method that takes a second bed file as input – other such methods include `BedTool.subtract()` and `BedTool.closest()`. In these cases, `pybedtools` assumes the first unnamed argument to these methods are the second file you want to operate on (and if you pass a `BedTool`, it'll automatically use the file in the `fn` attribute of that `BedTool`). So `a.intersect(b)` is just a more convenient form of `a.intersect(a=a.fn, b=b.fn)`, which does the same thing.

If you're ever unsure, the docstring for these methods indicates which, if any, arguments are used as default. For example, in the `BedTool.intersect()` help, it says:

```
.. note::

For convenience, the file this bedtool object points to is
passed as "-a"
```

OK, enough about arguments for now, but you can read more about them in *similarity principle*, *default args principle* and *non defaults principle*.

2.3.6 Chaining methods together (pipe)

One useful thing about `BedTool` methods is that they often return a new `BedTool`. In practice, this means that we can chain together multiple method calls all in one line, similar to piping on the command line.

For example, this intersect and merge can be combined into one command:

```
>>> # These two lines...
>>> a_with_b = a.intersect(b, u=True)
>>> a_with_b_merged_1 = a_with_b.merge()

>>> # Could be combined into one line:
>>> a_with_b_merged_2 = a.intersect(b, u=True).merge()

>>> a_with_b_merged_1 == a_with_b_merged_2
True
```

In general, methods that return `BedTool` objects have the following text in their docstring to indicate this:

```
.. note::

    This method returns a new BedTool instance
```

A rule of thumb is that all methods that wrap BEDTools programs return `BedTool` objects, so you can chain these together. Other `pybedtools`-unique methods return `BedTool` objects too, just check the docs (according to *good docs principle*). For example, as we saw in one of the examples above, the `BedTool.saveas()` method returns a `BedTool` object. That means we can sprinkle those commands within the example above to save the intermediate steps as meaningful filenames for later use. For example:

```
>>> a_with_b_merged_3 = a.intersect(b, u=True).saveas('a-with-b.bed').merge().saveas('a-with-b-merge
```

Now we have new files in the current directory called `a-with-b.bed` and `a-with-b-merged.bed`. Since `BedTool.saveas()` returns a `BedTool` object, `x2` points to the `a-with-b-merged.bed` file.

There's an even easier way to chain together commands.

I found myself doing intersections so much that I thought it would be useful to overload the + and - operators to do intersections. To illustrate, these two example commands do the same thing:

```
>>> a_with_b_1 = a.intersect(b, u=True)
>>> a_with_b_2 = a+b

>>> a_with_b_1 == a_with_b_2
True

>>> a_without_b_1 = a.intersect(b, v=True)
>>> a_without_b_2 = a-b

>>> a_without_b_1 == a_without_b_2
True
```

Note that the + operator assumes the `-u` option and the - operator assumes `intersectBed`'s `-v` option:

If you want to operating on the resulting `BedTool` that is returned by an addition or subtraction, you'll need to wrap the operation in parentheses:

```
>>> a_with_b_merged_4 = (a+b).merge()
```

And to double-check that all these methods return the same thing:

```
>>> a_with_b_merged_1 == a_with_b_merged_2 == a_with_b_merged_3 == a_with_b_merged_4
True
```

You can learn more about chaining in *chaining principle*.

2.3.7 Intervals

An `Interval` object is how `pybedtools` represents a line in a BED, GFF, GTF, or VCF file in a uniform fashion. This section will describe some useful features of `Interval` objects.

First, let's get a `BedTool` to work with:

```
>>> a = pybedtools.example_bedtool('a.bed')
```

We can access the `Intervals` of `a` several different ways. The most common use is probably by using the `BedTool` `a` as an iterator. For now, let's look at a single `Interval`:

```
>>> feature = iter(a).next()
```

Common `Interval` attributes

Printing a feature converts it into the original line from the file:

```
>>> print feature
chr1        1       100     feature1        0       +
```

All features have `chrom`, `start`, `stop`, `name`, `score`, and `strand` attributes. Note that `start` and `stop` are long integers, while everything else (including `score`) is a string.

```
>>> feature.chrom
'chr1'

>>> feature.start
1L

>>> feature.stop
100L

>>> feature.name
'feature1'

>>> feature.score
'0'

>>> feature.strand
'+'
```

Let's make another feature that only has chrom, start, and stop to see how `pybedtools` deals with missing attributes:

```
>>> feature2 = iter(pybedtools.BedTool('chrX 500 1000', from_string=True)).next()

>>> print feature2
chrX        500     1000


>>> feature2.chrom
'chrX'

>>> feature2.start
500L

>>> feature2.stop
1000L

>>> feature2.name
''

>>> feature2.score
''

>>> feature2.strand
''
```

This illustrates that default values are empty strings.

Indexing into `Interval` objects

`Interval` objects can be indexed by position into the original line (like a list) or indexed by name of attribute (like a dictionary).

```
>>> print feature
chr1        1       100     feature1        0       +


>>> feature[0]
'chr1'
```

```
>>> feature['chrom']
'chr1'

>>> feature[1]
'1'

>>> feature['start']
1L
```

BED is 0-based, others are 1-based

One troublesome part about working with multiple formats is that BED files have a different coordinate system than GFF/GTF/VCF/ files.

BED files are 0-based (the first base of the chromosome is considered position 0) and the feature does not include the stop position.

GFF, GTF, and VCF files are 1-based (the first base of the chromosome is considered position 1) and the feature includes the stop position.

Note: `pybedtools` follows the following conventions:

- The value in `Interval.start` will **always** contain the 0-based start position, even if it came from a GFF or other 1-based feature.
- Getting the `len()` of an `Interval` will always return `Interval.stop - Interval.start`, so no matter what format the original file was in, the length will be correct.
- The contents of `Interval.fields` will **always** be strings, which in turn always represent the original line in the file. This means that for a GFF feature, `Interval.fields[3]`, which is 1-based according to the file format, will always be one bp larger than `Interval.start`, which always contains the 0-based start position.

To illustrate and confirm, let's create a GFF feature and a BED feature from scratch and compare them:

```
>>> # GFF Interval from scratch
>>> gff = ["chr1",
...        "fake",
...        "mRNA",
...        "51",    # (1 greater than the BED start below)
...        "300",
...        ".",
...        "+",
...        ".",
...        "ID=mRNA1;Parent=gene1;"]
>>> gff = pybedtools.create_interval_from_list(gff)
>>> print gff
chr1        fake    mRNA    51      300     .       +       .       ID=mRNA1;Parent=gene1;

>>> # BED Interval from scratch
>>> bed = ["chr1",
...        "50",
...        "300",
...        "mRNA1",
...        ".",
...        "+"]
>>> bed = pybedtools.create_interval_from_list(bed)
>>> print bed
chr1        50      300     mRNA1   .       +


>>> # confirm they are recognized as the right type
>>> gff.file_type
'gff'
>>> bed.file_type
'bed'

>>> # Start attributes should be identical
>>> bed.start == gff.start == 50
True

>>> bed.start
50L
>>> bed[1]
'50'

>>> # GFF .start is 1 less than the string value stored at index 3
>>> gff.start
50L
>>> gff[3]
'51'

>>> len(bed) == len(gff) == 250
True
```

GFF features have access to attributes

GFF and GTF files have lots of useful information in their attributes field (the last field in each line). These attributes can be accessed with the `Interval.attrs` attribute, which acts like a dictionary. For speed, the attributes are lazy – they are only parsed when you ask for them. BED files, which do not have an attributes field, will return an empty dictionary.

```
>>> print gff
chr1        fake    mRNA    51      300     .       +       .       ID=mRNA1;Parent=gene1;

>>> gff.attrs
{'ID': 'mRNA1', 'Parent': 'gene1'}

>>> gff.attrs['Awesomeness'] = 99
>>> gff.attrs['ID'] = 'transcript1'

>>> # Changes in attributes are propagated to the printable feature
>>> print gff
chr1        fake    mRNA    51      300     .       +       .       Awesomeness=99;ID=transcript1;Pa
```

Understanding `Interval` objects is important for using the powerful filtering and mapping facilities of `BedTool` objects, as described in the next section.

2.3.8 Filtering

The `filter()` method lets you pass in a function that accepts an `Interval` as its first argument and returns True for False. For example, here's how to get a new `BedTool` containing features from `a` that are more than 100 bp long:

```
>>> a = pybedtools.example_bedtool(’a.bed’)
>>> b = a.filter(lambda x: len(x) > 100)
>>> print b
chr1        150     500     feature3        0       -
```

The `filter()` method will pass its `*args` and `**kwargs` to the function provided. So here is a more generic case would be the following, where the function is defined once and different arguments are passed in for filtering on different lengths:

```
>>> def len_filter(feature, L):
...     "Returns True if feature is longer than L"
...     return len(feature) > L
```

Now we can pass different lengths without defining a new function for each length of interest:

```
>>> a = pybedtools.example_bedtool(’a.bed’)

>>> print a.filter(len_filter, L=10)
chr1        1       100     feature1        0       +
chr1        100     200     feature2        0       +
chr1        150     500     feature3        0       -
chr1        900     950     feature4        0       +


>>> print a.filter(len_filter, L=99)
chr1        100     200     feature2        0       +
chr1        150     500     feature3        0       -


>>> print a.filter(len_filter, L=200)
chr1        150     500     feature3        0       -
```

See **:ref:‘BedTools as iterators‘_** for more advanced and space-efficient usage of `filter()` using iterators.

Fast filtering functions in Cython

The `featurefuncs` module contains some ready-made functions written in Cython that will be faster than pure Python equivalents. For example, there are `greater_than()` and `less_than()` functions, which are about 70% faster. In IPython:

```
>>> len(a)
310456

>>> def L(x,width=100):
...     return len(x) > 100

>>> %timeit a.filter(greater_than, 100)
1 loops, best of 3: 1.74 s per loop

>>> %timeit a.filter(L, 100)
1 loops, best of 3: 2.96 s per loop
```

2.3.9 Each

Similar to `BedTool.filter()`, which applies a function to return True or False given an `Interval`, the `BedTool.each()` method applies a function to return a new, possibly modified `Interval`.

The `BedTool.each()` method applies a function to every feature. Like `BedTool.filter()`, you can use your own function or some pre-defined ones in the `featurefuncs` module. Also like `filter()`, `*args` and `**kwargs` are sent to the function.

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> b = pybedtools.example_bedtool('b.bed')

>>> # The results of an "intersect" with c=True will return features
>>> # with an additional field representing the counts.
>>> with_counts = a.intersect(b, c=True)
```

The `featurefuncs.normalized_to_length()` function divides the value at position `N` by the length. Here we specify `N=-1`, which refers to the count from the previous step:

```
>>> from pybedtools import featurefuncs

>>> normalized = with_counts.each(featurefuncs.normalized_to_length, -1)

>>> print normalized
chr1        1       100     feature1        0       +       0.0
chr1        100     200     feature2        0       +       1.0000000475e-05
chr1        150     500     feature3        0       -       2.85714299285e-06
chr1        900     950     feature4        0       +       2.00000009499e-05
```

2.3.10 Using the history and tags

BEDTools makes it very easy to do rather complex genomic algebra. Sometimes when you're doing some exploratory work, you'd like to rewind back to a previous step, or clean up temporary files that have been left on disk over the course of some experimentation.

To assist this sort of workflow, `BedTool` instances keep track of their history in the `BedTool.history` attribute. Let's make an example `BedTool`, *c*, that has some history:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> b = pybedtools.example_bedtool('b.bed')
>>> c = a.intersect(b, u=True)
```

c now has a history which tells you all sorts of useful things (described in more detail below):

```
>>> print c.history
[<HistoryStep> bedtool("/home/ryan/pybedtools/pybedtools/test/a.bed").intersect("/home/ryan/pybedtool
```

There are several things to note here. First, the history describes the full commands, including all the names of the temp files and all the arguments that you would need to run in order to re-create it. Since `BedTool` objects are fundamentally file-based, the command refers to the underlying filenames (i.e., `a.bed` and `b.bed`) instead of the `BedTool` instances (i.e., *a* and *b*). A simple copy-paste of the command will be enough re-run the command. While this may be useful in some situations, be aware that if you do run the command again you'll get *another* temp file that has the same contents as *c*'s temp file.

To avoid such cluttering of your temp dir, the history also reports **tags**. `BedTool` objects, when created, get a random tag assigned to them. You can get get the `BedTool` associated with tag with the `pybedtools.find_tagged()` function. These tags are used to keep track of instances during this session.

So in this case, we could get a reference to the *a* instance with:

```
>>> should_be_a = pybedtools.find_tagged('klkreuay')
```

Here's confirmation that the parent of the first step of *c*'s history is *a* (note that `HistoryStep` objects have a `HistoryStep.parent_tag` and `HistoryStep.result_tag`):

```
>>> pybedtools.find_tagged(c.history[0].parent_tag) == a
True
```

Let's make something with a more complicated history:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> b = pybedtools.example_bedtool('b.bed')
>>> c = a.intersect(b)
>>> d = c.slop(g=pybedtools.chromsizes('hg19'), b=1)
>>> e = d.merge()

>>> # this step adds complexity!
>>> f = e.subtract(b)
```

Let's see what the history of *f* (the last `BedTool` created) looks like . . . note that here I'm formatting the results to make it easier to see:

```
>>> print f.history
[
|   [
|   |   [
|   |   |   [
|   |   |   |<HistoryStep> BedTool("/home/ryan/pybedtools/pybedtools/test/a.bed").intersect(
|   |   |   |                      "/home/ryan/pybedtools/pybedtools/test/b.bed",
|   |   |   |                      ),
|   |   |   |                      parent tag: rzrztxlw,
|   |   |   |                      result tag: ifbsanqk
|   |   |   ],
|   |   |
|   |   |<HistoryStep> BedTool("/tmp/pybedtools.BgULVj.tmp").slop(
|   |   |                      b=1,genome="hg19"
|   |   |                      ),
|   |   |                      parent tag: ifbsanqk,
|   |   |                      result tag: omfrkwjp
|   |   ],
|   |
|   |<HistoryStep> BedTool("/tmp/pybedtools.SFmbYc.tmp").merge(),
|   |                      parent tag: omfrkwjp,
|   |                      result tag: zlwqblvk
|   ],
|
|<HistoryStep> BedTool("/tmp/pybedtools.wlBiMo.tmp").subtract(
|                      "/home/ryan/pybedtools/pybedtools/test/b.bed",
|                      ),
|                      parent tag: zlwqblvk,
|                      result tag: reztxhen
]
```

Those first three history steps correspond to *c*, *d*, and *e* respectively, as we can see by comparing the code snippet above with the commands in each history step. In other words, *e* can be described by the sequence of 3 commands in the first three history steps. In fact, if we checked *e.history*, we'd see exactly those same 3 steps.

When *f* was created above, it operated both on *e*, which had its own history, as well as *b* – note the nesting of the list. You can do arbitrarily complex "genome algebra" operations, and the history of the `BEDTools` will keep track of this. It may not be useful in every situtation, but the ability to backtrack and have a record of what you've done can sometimes be helpful.

2.3.11 Deleting temp files specific to a single BedTool

You can delete temp files that have been created over the history of a BedTool with BedTool.delete_temporary_history(). This method will inspect the history, figure out which items point to files in the temp dir (which you can see with get_tempdir()), and prompt you for their deletion:

```
>>> f.delete_temporary_history()
Delete these files?
    /tmp/pybedtools..BgULVj.tmp
    /tmp/pybedtools.SFmbYc.tmp
    /tmp/pybedtools.wlBiMo.tmp
(y/N) y
```

Note that the file that *f* points to is left alone. To clarify, the BedTool.delete_temporary_history() will only delete temp files that match the pattern <TEMP_DIR>/pybedtools.*.tmp from the history of *f*, up to but not including the file for *f* itself. Any BedTool instances that do not match the pattern are left alone. Use the kwarg ask=False to disable the prompt.

2.3.12 BedTools as iterators

The filter() method uses a file-based format, where the new BedTool object refers to a new temp file. You can use a generator function to create a new BedTool if you want to save disk space:

```
>>> def len_filter(feature, L):
...     "Returns True if feature is longer than L"
...     return len(feature) > L

>>> a = pybedtools.example_bedtool('a.bed')

>>> b = pybedtools.BedTool( (i for i in a if len_filter(i, L=200)))

>>> print b
chr1        150     500     feature3        0       -
```

However, keep in mind that printing b, which was created using a generator expression, has now been consumed – so printing b again will do nothing:

```
>>> print b
```

If you create a BedTool with a generator expression, you can always save it as a file for later use. This is what filter() is doing:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> b = pybedtools.BedTool( (i for i in a if len_filter(i, L=200))).saveas('len-filtered-b.bed')
>>> print b
chr1        150     500     feature3        0       -


>>> print b
chr1        150     500     feature3        0       -
```

2.3.13 More documentation

For more info, see the *Topical Documentation*.

2.4 Topical Documentation

2.4.1 Using BedTool objects as iterators/generators

Typically, `BedTool` objects are used somewhat like handles to individual files on disk that contain BED lines. To save disk space, `BedTool` objects also have the ability to “stream”, much like piping in Unix.

You’ll need to be careful when using `BedTool` objects as generators, since any operation that reads all the features of a `BedTool` will consume the iterable.

To get a streaming BedTool, use the `stream=True` kwarg:

```
>>> a = pybedtools.example_bedtool(’a.bed’)
>>> b = pybedtools.example_bedtool(’b.bed’)
>>> c = a.intersect(b, stream=True)

>>> # checking the length consumes the iterator
>>> len(c)
3

>>> # nothing left, so checking length again returns 0
>>> len(c)
0
```

In some cases, a stream may be “rendered” to a temp file. This is because BEDTools programs can only accept one input file as `stdin`. This is typically the first input (`-i` or `-a`), while the other input (`-b`) must be a file. Consider this example, where the second intersection needs to convert the streaming BedTool to a file before sending to `intersectBed`:

```
>>> a = pybedtools.example_bedtool(’a.bed’)
>>> b = pybedtools.example_bedtool(’b.bed’)

>>> # first we set up a streaming BedTool:
>>> c = a.intersect(b, stream=True)

>>> # But supplying a streaming BedTool as the first unnamed argument
>>> # means it is being passed as -b to intersectBed, and so must be a file.
>>> # In this case, ‘c‘ is rendered to a tempfile before being passed.
>>> d = a.intersect(c, stream=True)
```

Creating a `BedTool` from an iterable

You can create a `BedTool` on the fly from a generator or iterator – in fact, this is what the `BedTool.filter()` method does for you:

```
>>> a = pybedtools.example_bedtool(’a.bed’)
>>> print a
chr1        1       100     feature1        0       +
chr1        100     200     feature2        0       +
chr1        150     500     feature3        0       -
chr1        900     950     feature4        0       +


>>> b = pybedtools.BedTool(f for f in a if f.start > 200)

>>> # this is the same as using filter:
>>> c = a.filter(lambda x: x.start > 200)
```

We need to "render" these BedTools to string before we can check equality since they are both iterables for which == is not defined:

```
>>> b == c
Traceback (most recent call last):
    ...
NotImplementedError: Testing equality only supported for BedTools that point to a file

>>> str(b) == str(c)
True
```

2.4.2 Working with BAM files

Some BEDTools programs support BAM files as input; for example `intersectBed`, `windowBed`, and others accept a `-abam` argument instead of `-a` for the first input file.

This section describes the workflow for working with BAM files within `pybedtools`.

As an example, let's intersect a BAM file of reads with annotations using files that ship with `pybedtools`. First, we create the `BedTool` objects:

```
>>> a = pybedtools.example_bedtool('x.bam')
>>> b = pybedtools.example_bedtool('dm3-chr2L-5M.gff.gz')
```

If `a` referred to a BED file like `a.bed`, we could just do `a.intersect(b)` because `a.bed` would be implictly passed as `-a` and the gzipped GFF file would be passed as `-b`. In order to use a BAM file, however, we need to explicitly specify an `abam` kwarg. In addition, since Python doesn't allow non-keyword arguments after keyword arguments, we need to explicitly specify a `b` kwarg. This should be much clearer with a simple example:

```
>>> c = a.intersect(abam=a.fn, b=b)
```

Now `c` points to a new BAM file on disk. Keep in mind that there is not yet iterable BAM support in `pybedtools`, so things like `c.count()` or iterating over `c` with a `for feature in c:  ...` won't work. For now, consider using a package like HTSeq for access to individual reads in BAM format.

Alternatively, we can specify the `bed=True` kwarg to convert the intersected BAM results to BED format, and use those like a normal BED file:

```
>>> d = a.intersect(abam=a.fn, b=b, bed=True)
```

The resulting BedTool `d` refers to a BED file and can be used like any other:

```
>>> d.count()
341324


>>> print iter(d).next()
chr2L       9329    9365    HWUSI-NAME:2:69:512:1017#0      3       -
```

2.4.3 Wrapping new tools

This section serves as a reference for wrapping new tools as they are added to BEDTools.

Let's assume we would like to wrap a new program, appropriately named `newProgramBed`. Its signature from the command line is `newProgramBed -a <infile> -b <other file> [options]`, and it accepts `-a stdin` to indicate data is being piped to it.

Method name

Generally, I've tried to keep method names as similar as possible to BEDTools programs while still being PEP8-compliant. The trailing 'Bed' is usually removed from the program name. So here the name would probably be `new_program`.

Method signature (args and kwargs)

If a BEDTools program accepts a `-b` argument (which is the case for this example) then the signature and first line should look like this:

```
def new_program(self, b=None, **kwargs):
    kwargs['b'] = b
    ...
```

That is, we specify a kwarg for `b` in the signature so that we have the option calling this method either with a regular arg or a kwarg, and then we ensure that `b` makes it into the `**kwargs` dictionary that will be passed around later.

This is to allow another BedTool (or file, or stream) be passed as the first non-keyword argument; otherwise, the user would always have to do:

```
a.intersect(b=b)
```

instead of:

```
a.intersect(b)
```

It's a minor thing, but it's convenient.

If the program you're wrapping doesn't accept another BED-like file as `-b`, then the method should only accept `self` and `**kwargs`:

```
def new_program(self, **kwargs):
    pass
```

Setting the default

Generally, the next thing to do is to set the default, or implicit kwarg. This is generally the `-i` or `-a` kwarg.

This is done by checking to see if the implicit kwarg was specified already (which allows the user to override the implicit assumption) and if not, add it to the kwargs dict.

For example:

```
if 'a' not in kwargs:
    kwargs['a'] = self.fn
```

For BEDTools programs that optionally take an `-abam` argument, we need to check to make sure that wasn't specified either:

```
if ('abam' not in kwargs) and ('a' not in kwargs):
        kwargs['a'] = self.fn
```

So to continue the example, our method now looks like this:

```
def new_program(self, b=None, **kwargs):

    if 'a' not in kwargs:
        kwargs['a'] = self.fn
```

Allowing input streams

BEDTools programs that can accept stdin as their first input need to be registered in the BedTool.handle_kwargs() method, in the implicit_instream1 dictionary.

This dictionary specifies which keyword argument to look for an object that could be used as the first input – this could be a filename, an iterator of features, or an open file. Depending on what it finds there, it will call the BEDTools program appropriately.

For example, if BedTool.handle_kwargs() finds an open file in kwargs['a'] for intersectBed, then it will tell intersectBed that -a should be stdin and the open file will eventually be passed to the subprocess.Popen call. But if kwargs['a'] is a filename, then it will just tell intersectBed that -a should be the filename.

If there is a second argument that has the potential to be a stream (like the b kwarg for BedTool.intersect(), which can be a BedTool, filename, IntervalIterator, or stream), then this kwarg should be added to the implicit_instream2 dictionary.

This second dictionary specifies which kwargs to check to see if they contain an iterable. If so, then it "renders" the iterable to a temp file and will pass that filename to the BEDTools program.

In the example case of BedTool.intersect(), if b is a stream or an iterable, the handle_kwargs method will convert that stream or iterable to a tempfile (say, tmp001) and then will eventually send intersectBed the option -b tmp001.

The BedTool.handle_kwargs() method also decides whether to create a new temp file (if stream=False) or not, and also converts kwargs like a to -a. Finally, it creates a list of commands ready for call_bedtools() to run.

To illustrate, we add a to the implicit_instream1 dict:

```
implicit_instream1 = {'intersectBed': 'a',
                      'subtractBed': 'a',
                       'closestBed': 'a',
                        'windowBed': 'a',
                          'slopBed': 'i',
                         'mergeBed': 'i',
                          'sortBed': 'i',
                      'bed12ToBed6': 'i',
                       'shuffleBed': 'i',
                      'annotateBed': 'i',
                         'flankBed': 'i',
                     'fastaFromBed': 'bed',
                 'maskFastaFromBed': 'bed',
                      'coverageBed': 'a',
                      'newthingBed': 'a', # here's the new wrapped program
                      }
```

And back in the method body, we call BedTool.handle_kwargs() so that our method now looks like this:

```
def new_program(self, b=None, **kwargs):

    if 'a' not in kwargs:
        kwargs['a'] = self.fn

    cmds, tmp, stdin = self.handle_kwargs(prog='newthingBed', **kwargs)
```

Call BEDTools, and return result

So now we have cmds (the list of commands ready to be called by subprocess.Popen), tmp (the new tempfile ready to accept results, or None if we will be streaming the output), and stdin (None if it’s a filename, or the open file that will be send to subprocess.stdin). These are passed to the call_bedtools() function, which does all the subprocess business and returns a “stream”, which could be any of the things that a BedTool can accept (filename, open file, IntervalIterator). Finally, we return a new BedTool made from this stream.

Now our method looks like this:

```
def new_program(self, b=None, **kwargs):

    if ’a’ not in kwargs:
        kwargs[’a’] = self.fn

    cmds, tmp, stdin = self.handle_kwargs(prog=’newthingBed’, **kwargs)
    stream = call_bedtools(cmds, tmp, stdin=stdin)
    return BedTool(stream)
```

Add decorators

Some decorators are used to add text to the method’s docstring, like _implicit and _file_or_bedtool, and _returns_bedtool. More useful is _help, which adds the full text of the BEDTools program to the end of the docstring, so all information is available from the Python interpreter (especially useful when using IPython). The _log_to_history decorator will register the calling of this method in the BedTool’s history.

The final wrapped method, with all the decorators to add relevant text to the docstring, then is simply:

```
@_returns_bedtool()
@_file_or_bedtool()
@_help(’newProgramBed’)
@_log_to_history
@_implicit(’-a’)
def new_program(self, b=None, **kwargs):

    if ’a’ not in kwargs:
        kwargs[’a’] = self.fn

    cmds, tmp, stdin = self.handle_kwargs(prog=’newthingBed’, **kwargs)
    stream = call_bedtools(cmds, tmp, stdin=stdin)
    return BedTool(stream)
```

Write tests!

The only way to know for sure if your new wrapped method works is to write good tests for it. This can either be done in the docstring or in the test suite. See the source for how this is done; also check out the test.sh script in the top level of the repository.

Send a pull request

If you’ve made something that you think would be useful to others, please send a github pull request so that your newly created and tested code can be distributed to others. Any contributions are much appreciated.

2.4.4 Specifying genomes

This section illustrates the use of genome files for use with BEDTools programs that need to know chromosome limits to prevent out-of-range coordinates.

Using BEDTools programs like `slopBed` or `shuffleBed` from the command line requires “genome” or “chromsizes” files. `pybedtools` comes with common genome assemblies already set up as a dictionary with chromosomes as keys and zero-based (start, stop) tuples as values:

```
>>> from pybedtools import genome_registry
>>> genome_registry.dm3[’chr2L’]
(0, 23011544)
```

The rules for specifying a genome for methods that require a genome are as follows (use whatever is most convenient):

- Use `g` to specify either a filename or a dictionary
- Use `genome` to specify either an assembly name or a dictionary

Below are examples of each.

As a file

This is the typical way of using BEDTools programs, by specifying an existing genome file with `g`:

```
>>> a = pybedtools.example_bedtool(’a.bed’)
>>> b = a.slop(b=100, g=’hg19.genome’)
```

As a string

This is probably the most convenient way of specifying a genome. If the genome exists in the genome registry it will be used directly; otherwise it will automatically be downloaded from UCSC. You must use the `genome` kwarg for this; if you use `g` a string will be interpreted as a filename:

```
>>> c = a.slop(b=100, genome=’hg19’)
```

As a dictionary

This is a good way of providing custom coordinates; either `g` or `genome` will accept a dictionary:

```
>>> d = a.slop(b=100, g={’chr1’:(1, 10000)})
>>> e = a.slop(b=100, genome={’chr1’:(1,100000)})
```

Make sure that all these different methods return the same results

```
>>> b == c == d == e
True
```

2.4.5 Randomization

`pybedtools` provides some basic functionality for assigning some significance value to the overlap between two BEDfiles.

The strategy is to randomly shuffle a file many times, each time doing an intersection with another file of interest and counting the number of intersections. Upon doing this many times, an empirical distribution is constructed, and the number of intersections between the original, un-shuffled file is compared to this empirical distribution to obtain a p-value, or compared to the median of the distribution to get a score.

There are two methods, `pybedtools.BedTool.randomintersection()` which does the brute force randomizations, and `BedTool.randomstats()` which compiles and reports the results from the former method.

Example workflow

As a somewhat trivial example, we'll intersect the example `a.bed` with `b.bed`, taking care to set some options that will let it run in a determinisitic way so that these tests will run.

We will be shuffling `a.bed`, so we'll need to specify the limits of its chromosomes with `BedTool.set_chromsizes()`. Here, we set it to an artifically small chromosome size so that we can get some meaningful results in reasonable time. In practice, you would either supply your own dictionary or use a string assembly name (e.g., `'hg19'`, `'mm9'`, `'dm3'`, etc). The genome-handling code will find the chromsizes we've set, so there's no need to tell `shuffleBed` which genome file to use each time.

```
>>> chromsizes = {'chr1': (0, 1000)}
>>> a = pybedtools.example_bedtool('a.bed').set_chromsizes(chromsizes)
>>> b = pybedtools.example_bedtool('b.bed')
```

We have the option of specifying what kwargs to provide `BedTool.shuffle()` and `BedTool.intersect()`, which will be called each iteration. In this example, we'll tell `shuffleBed` to only shuffle within the chromsome just to illustrate the kwargs passing. We also need to specify how many iterations to perform. In practice, 1000 or 10000 are good numbers, but for the sake of this example we'll only do 100.

Last, setting `debug=True` means that the random seed will be set in a predictable manner so that we'll always get the same results for testing. In practice, make sure you use `debug=False` (the default) to ensure random results.

```
>>> results = a.randomintersection(b, iterations=100, shuffle_kwargs={'chrom': True}, debug=True)
```

`results` is a generator of intersection counts where each number is the number of times the shuffled `a` intersected with `b`. We need to convert it to a list in order to look at it:

```
>>> results = list(results)
>>> len(results)
100

>>> print results[:10]
[1, 1, 2, 2, 1, 2, 1, 0, 2, 3]
```

Running thousands of iterations on files with many features will of course result in more complex results. We could then take these results and plot them in matplotlib, or get some statistics on them.

The method `BedTool.randomstats()` does this for us, but requires NumPy and SciPy to be installed. This method also calls `BedTool.randomintersection()` for us, returning the summarized results in a dictionary.

`BedTool.randomstats()` takes the same arguments as `BedTool.randomintersection()`:

```
>>> results_dict = a.randomstats(b, iterations=100, shuffle_kwargs={'chrom': True}, debug=True)
```

The keys to this results dictionary are as follows (some are redundant, I've found these keys useful for writing out to file):

iterations the number of iterations we specified

actual the number of intersections between then un-shuffled `a` and `b`

file_a the filename of `a`

file_b the filename of `b`

<file_a> the key is actully the filename of a, and the value is the number of features in a

<file_b> the key is actually the filename of b and the value is the number of features in b

self number of features in a (or "self"; same value as for <file_a>)

other number of features in b (or "other"; same value as for <file_b>)

frac randomized above actual fraction of iterations that had counts above the actual count

frac randomized below actual fraction of iterations that had counts below the actual count

median randomized the median of the distribution of randomized intersections

normalized the actual count divided by the median; can be considered as a score

percentile the percentile of actual within the distribution of randomized intersections; can be considered an empirical p-value

upper 97.5th the 97.5th percentile of the randomized distribution

lower 2.5th the 2.5th percentile of the randomized distribution

For example:

```
>>> keys = ['self', 'other', 'actual', 'median randomized', 'normalized', 'percentile']
>>> for key in keys:
...     print '%s: %s' % (key, results_dict[key])
self: 4
other: 2
actual: 3
median randomized: 2.0
normalized: 1.5
percentile: 92.0
```

Contributions toward improving this code or implementing other methods of statistical testing are very welcome!

Contents

2.5 `pybedtools` module-level functions

2.5.1 Functions for working with example files

pybedtools.**example_bedtool**(*fn*)

Return a bedtool using a bed file from the pybedtools examples directory. Use `list_example_files()` to see a list of files that are included.

pybedtools.**example_filename**(*fn*)

Return a bed file from the pybedtools examples directory. Use `list_example_files()` to see a list of files that are included.

pybedtools.**list_example_files**()

Returns a list of files in the examples dir. Choose one and pass it to `example_file_fn1()` to get the full path to an examplefile.

Example usage:

```
>>> choices = list_example_files()
>>> assert 'a.bed' in choices
>>> bedfn = example_filename('a.bed')
>>> mybedtool = BedTool(bedfn)
```

2.5.2 Functions for specifying genome assemblies

pybedtools.**chromsizes**(*genome*)

Looks for a *genome* already included in the genome registry; if not found then it looks it up on UCSC. Returns the dictionary of chromsize tuples where each tuple has (start,stop).

Chromsizes are described as (start, stop) tuples to allow randomization within specified regions; e. g., you can make a chromsizes dictionary that represents the extent of a tiling array.

Example usage:

```
>>> dm3_chromsizes = chromsizes('dm3')
>>> for i in sorted(dm3_chromsizes.items()):
...     print i
('chr2L', (0, 23011544))
('chr2LHet', (0, 368872))
('chr2R', (0, 21146708))
('chr2RHet', (0, 3288761))
('chr3L', (0, 24543557))
('chr3LHet', (0, 2555491))
('chr3R', (0, 27905053))
('chr3RHet', (0, 2517507))
('chr4', (0, 1351857))
('chrM', (0, 19517))
('chrU', (0, 10049037))
('chrUextra', (0, 29004656))
('chrX', (0, 22422827))
('chrXHet', (0, 204112))
('chrYHet', (0, 347038))
```

pybedtools.**chromsizes_to_file**(*chromsizes*, *fn=None*)

Converts a *chromsizes* dictionary to a file. If *fn* is None, then a tempfile is created (which can be deleted with pybedtools.cleanup()).

Returns the filename.

pybedtools.**get_chromsizes_from_ucsc**(*genome*, *saveas=None*, *mysql='mysql'*, *timeout=None*)

Download chrom size info for *genome* from UCSC and returns the dictionary.

If you need the file, then specify a filename with *saveas* (the dictionary will still be returned as well).

If `mysql` is not on your path, specify where to find it with *mysql=<path to mysql executable>*.

timeout is how long to wait for a response; mostly used for testing. Will only be used if

Example usage:

```
>>> dm3_chromsizes = get_chromsizes_from_ucsc('dm3')
>>> for i in sorted(dm3_chromsizes.items()):
...     print i
('chr2L', (0, 23011544))
('chr2LHet', (0, 368872))
('chr2R', (0, 21146708))
('chr2RHet', (0, 3288761))
('chr3L', (0, 24543557))
('chr3LHet', (0, 2555491))
('chr3R', (0, 27905053))
('chr3RHet', (0, 2517507))
('chr4', (0, 1351857))
('chrM', (0, 19517))
('chrU', (0, 10049037))
('chrUextra', (0, 29004656))
('chrX', (0, 22422827))
('chrXHet', (0, 204112))
('chrYHet', (0, 347038))
```

2.5.3 Setup

pybedtools.**set_tempdir**(*tempdir*)

Sets the directory for temp files. Useful for clusters that use a /scratch partition rather than a /tmp dir. Convenience function to simply set tempfile.tempdir.

pybedtools.**get_tempdir**()

Gets the current tempdir for the module.

pybedtools.**set_bedtools_path**(*path=''*)

If BEDTools is not available on your system path, specify the path to the dir containing the BEDTools executables (intersectBed, subtractBed, etc) with this function.

To reset and use the default system path, call this function with no arguments or use path="".

2.5.4 Utilities

pybedtools.**cleanup**(*verbose=True*, *remove_all=False*)

Deletes all temporary files in the *BedTool.TEMPFILES* class variable.

If *verbose*, reports what it's doing

If *remove_all*, then ALL files matching "pybedtools.*.tmp" in the temp dir will be deleted.

2.6 `BedTool` methods that wrap BEDTools programs

2.6.1 "Genome algebra" methods

`intersect()` (wraps "intersectBed")

`BedTool.`**`intersect`**`(`**args, **kwargs*`)`

pybedtools help:

Intersect with another BED file. If you want to use BAM as input, you need to specify *abam='filename.bam'*. Returns a new BedTool object.

Example usage:

Create new BedTool object

```
>>> a = pybedtools.example_bedtool('a.bed')
```

Get overlaps with `b.bed`:

```
>>> b = pybedtools.example_bedtool('b.bed')
>>> overlaps = a.intersect(b)
```

Use `v=True` to get the inverse – those unique to "a.bed":

```
>>> unique_to_a = a.intersect(b, v=True)
```

Note: This method returns a new bedtool instance

Note: For convenience, the file this bedtool object points to is passed as "-a"

Note: This method accepts either a bedtool or a file name as the first unnamed argument

Original BEDtools program help:

Program: intersectBed (v2.11.2) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Report overlaps between two feature files.

Usage: intersectBed [OPTIONS] -a <bed/gff/vcf> -b <bed/gff/vcf>

Options:

-abam	The A input file is in BAM format. Output will be BAM as well.
-ubam	Write uncompressed BAM output. Default is to write compressed BAM.
-bed	When using BAM input (-abam), write output as BED. The default is to write output in BAM when using -abam.
-wa	Write the original entry in A for each overlap.
-wb	Write the original entry in B for each overlap. - Useful for knowing **what** A overlaps. Restricted by -f and -r.

-wo Write the original A and B entries plus the number of base pairs of overlap between the two features. - Overlaps restricted by -f and -r.

> Only A features with overlap are reported.

-wao Write the original A and B entries plus the number of base pairs of overlap between the two features. - Overlapping features restricted by -f and -r.

> However, A features w/o overlap are also reported with a NULL B feature and overlap = 0.

-u Write the original A entry **once** if **any** overlaps found in B. - In other words, just report the fact >=1 hit was found. - Overlaps restricted by -f and -r.

-c For each entry in A, report the number of overlaps with B. - Reports 0 for A entries that have no overlap with B. - Overlaps restricted by -f and -r.

-v Only report those entries in A that have **no overlaps** with B. - Similar to "grep -v" (an homage).

-f Minimum overlap required as a fraction of A. - Default is 1E-9 (i.e., 1bp). - FLOAT (e.g. 0.50)

-r Require that the fraction overlap be reciprocal for A and B. - In other words, if -f is 0.90 and -r is used, this requires

> that B overlap 90% of A and A **also** overlaps 90% of B.

-s Force strandedness. That is, only report hits in B that overlap A on the same strand. - By default, overlaps are reported without respect to strand.

-split Treat "split" BAM or BED12 entries as distinct BED intervals.

`merge()` (wraps "mergeBed")

`BedTool.`**`merge`**`(`**args*, ***kwargs*`)`

pybedtools help:

Merge overlapping features together. Returns a new BedTool object.

Example usage:

```
>>> a = pybedtools.example_bedtool('a.bed')
```

Merge:

```
>>> c = a.merge()
```

Allow merging of features 500 bp apart:

```
>>> c = a.merge(d=500)
```

Report number of merged features:

```
>>> c = a.merge(n=True)
```

Report names of merged features:

```
>>> c = a.merge(nms=True)
```

Note: This method returns a new bedtool instance

Note: For convenience, the file this bedtool object points to is passed as "-i"

Original BEDtools program help:

Program: mergeBed (v2.11.2) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Merges overlapping BED/GFF/VCF entries into a single interval.

Usage: mergeBed [OPTIONS] -i <bed/gff/vcf>

Options:

-s	Force strandedness. That is, only merge features that are the same strand. - By default, merging is done without respect to strand.
-n	Report the number of BED entries that were merged. - Note: "1" is reported if no merging occurred.
-d	Maximum distance between features allowed for features to be merged. - Def. 0. That is, overlapping & book-ended features are merged. - (INTEGER)
-nms	Report the names of the merged features separated by semicolons.

-scores [STRING] Report the scores of the merged features. Specify one of

the following options for reporting scores: sum, min, max, mean, median, mode, antimode, collapse (i.e., print a semicolon-separated list),

subtract() (wraps "subtractBed")

BedTool.**subtract**(**args*, ***kwargs*)

pybedtools help:

Subtracts from another BED file and returns a new BedTool object.

Example usage:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> b = pybedtools.example_bedtool('b.bed')
```

Do a "stranded" subtraction:

```
>>> c = a.subtract(b, s=True)
```

Require 50% of features in a to overlap:

```
>>> c = a.subtract(b, f=0.5)
```

Note: This method returns a new bedtool instance

Note: This method accepts either a bedtool or a file name as the first unnamed argument

Original BEDtools program help:

Program: subtractBed (v2.11.2) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Removes the portion(s) of an interval that is overlapped

by another feature(s).

Usage: subtractBed [OPTIONS] -a <bed/gff/vcf> -b <bed/gff/vcf>

Options:

-f	Minimum overlap required as a fraction of A. - Default is 1E-9 (i.e., 1bp). - (FLOAT) (e.g. 0.50)
-s	Force strandedness. That is, only report hits in B that overlap A on the same strand. - By default, overlaps are reported without respect to strand.

closest() (wraps "closestBed")

BedTool.**closest**(**args*, ***kwargs*)

pybedtools help:

Return a new BedTool object containing closest features in *b*. Note that the resulting file is no longer a valid BED format; use the special "_closest" methods to work with the resulting file.

Example usage:

```
a = BedTool('in.bed')

# get the closest feature in 'other.bed' on the same strand
b = a.closest('other.bed', s=True)
```

Note: This method returns a new bedtool instance

Note: For convenience, the file this bedtool object points to is passed as "-a"

Note: This method accepts either a bedtool or a file name as the first unnamed argument

Original BEDtools program help:

Program: closestBed (v2.11.2) Authors: Aaron Quinlan (aaronquinlan@gmail.com)

Erik Arner, Riken

Summary: For each feature in A, finds the closest feature (upstream or downstream) in B.

Usage: closestBed [OPTIONS] -a <bed/gff/vcf> -b <bed/gff/vcf>

Options:

-s Force strandedness. That is, find the closest feature in B that overlaps A on the same strand. - By default, overlaps are reported without respect to strand.

-d In addition to the closest feature in B, report its distance to A as an extra column. - The reported distance for overlapping features will be 0.

-t How ties for closest feature are handled. This occurs when two features in B have exactly the same overlap with A. By default, all such features in B are reported. Here are all the options: - “all” Report all ties (default). - “first” Report the first tie that occurred in the B file. - “last” Report the last tie that occurred in the B file.

Notes: Reports “none” for chrom and “-1” for all other fields when a feature is not found in B on the same chromosome as the feature in A. E.g. none -1 -1

window() (wraps “windowBed”)

BedTool.**window**(**args*, ***kwargs*)

pybedtools help:

Intersect with a window.

Example usage:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> b = pybedtools.example_bedtool('b.bed')
>>> print a.window(b, w=1000)
chr1        1       100     feature1        0       +       chr1    155     200     feat
chr1        1       100     feature1        0       +       chr1    800     901     feat
chr1        100     200     feature2        0       +       chr1    155     200     feat
chr1        100     200     feature2        0       +       chr1    800     901     feat
chr1        150     500     feature3        0       -       chr1    155     200     feat
chr1        150     500     feature3        0       -       chr1    800     901     feat
chr1        900     950     feature4        0       +       chr1    155     200     feat
chr1        900     950     feature4        0       +       chr1    800     901     feat
<BLANKLINE>
```

Note: This method returns a new bedtool instance

Note: For convenience, the file this bedtool object points to is passed as “-a”

Note: This method accepts either a bedtool or a file name as the first unnamed argument

Original BEDtools program help:

Program: windowBed (v2.11.2) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Examines a "window" around each feature in A and

> reports all features in B that overlap the window. For each overlap the entire entry in A and B are reported.

Usage: windowBed [OPTIONS] -a <bed/gff/vcf> -b <bed/gff/vcf>

Options:

-abam The A input file is in BAM format. Output will be BAM as well.

-ubam Write uncompressed BAM output. Default is to write compressed BAM.

-bed When using BAM input (-abam), write output as BED. The default is to write output in BAM when using -abam.

-w Base pairs added upstream and downstream of each entry in A when searching for overlaps in B. - Creates symterical "windows" around A. - Default is 1000 bp. - (INTEGER)

-l Base pairs added upstream (left of) of each entry in A when searching for overlaps in B. - Allows one to define assymterical "windows". - Default is 1000 bp. - (INTEGER)

-r Base pairs added downstream (right of) of each entry in A when searching for overlaps in B. - Allows one to define assymterical "windows". - Default is 1000 bp. - (INTEGER)

-sw Define -l and -r based on strand. For example if used, -l 500 for a negative-stranded feature will add 500 bp downstream. - Default = disabled.

-sm Only report hits in B that overlap A on the same strand. - By default, overlaps are reported without respect to strand.

-u Write the original A entry **once** if **any** overlaps found in B. - In other words, just report the fact >=1 hit was found.

-c For each entry in A, report the number of overlaps with B. - Reports 0 for A entries that have no overlap with B. - Overlaps restricted by -f.

-v Only report those entries in A that have **no overlaps** with B. - Similar to "grep -v."

`sort()` (wraps "sortBed")

`BedTool.`**`sort`**`(`**args*, ***kwargs*`)`

pybedtools help:

Note that chromosomes are sorted lexograpically, so chr12 will come before chr9.

Example usage:

```
>>> a = pybedtools.BedTool('''
... chr9 300 400
... chr1 100 200
... chr1 1 50
... chr12 1 100
... chr9 500 600
... ’’’, from_string=True)
>>> print a.sort()
chr1    1       50
chr1    100     200
chr12   1       100
chr9    300     400
chr9    500     600
<BLANKLINE>
```

Note: This method returns a new bedtool instance

Note: For convenience, the file this bedtool object points to is passed as “-i”

Original BEDtools program help:

Program: sortBed (v2.11.2) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Sorts a feature file in various and useful ways.

Usage: sortBed [OPTIONS] -i <bed/gff/vcf>

Options:

-sizeA	Sort by feature size in ascending order.
-sizeD	Sort by feature size in descending order.
-chrThenSizeA	Sort by chrom (asc), then feature size (asc).
-chrThenSizeD	Sort by chrom (asc), then feature size (desc).
-chrThenScoreA	Sort by chrom (asc), then score (asc).
-chrThenScoreD	Sort by chrom (asc), then score (desc).

slop() (wraps “slopBed”)

BedTool.**slop**(**args*, ***kwargs*)

pybedtools help:

Wraps slopBed, which adds bp to each feature. Returns a new BedTool object.

If *g* is a dictionary (for example, return values from pybedtools.chromsizes()) it will be converted to a temp file for use with slopBed. If it is a string, then it is assumed to be a filename.

Alternatively, use *genome* to indicate a pybedtools-created genome. Example usage:

```
>>> a = pybedtools.example_bedtool(’a.bed’)
```

Increase the size of features by 100 bp in either direction. Note that you need to specify either a dictionary of chromsizes or a filename containing chromsizes for the genome that your bed file corresponds to:

```
>>> c = a.slop(g=pybedtools.chromsizes(’hg19’), b=100)
```

Grow features by 10 bp upstream and 500 bp downstream, using a genome file you already have constructed called 'hg19.genome'

First, create the file:

```
>>> fout = open('hg19.genome','w')
>>> chromdict = pybedtools.get_chromsizes_from_ucsc('hg19')
>>> for chrom, size in chromdict.items():
...     fout.write("%s\t%s\n" % (chrom, size[1]))
>>> fout.close()
```

Then use it:

```
>>> c = a.slop(g='hg19.genome', l=10, r=500, s=True)
```

Clean up afterwards:

```
>>> os.unlink('hg19.genome')
```

Note: This method returns a new bedtool instance

Note: For convenience, the file this bedtool object points to is passed as "-i"

Original BEDtools program help:

Program: slopBed (v2.11.2) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Add requested base pairs of "slop" to each feature.

Usage: slopBed [OPTIONS] -i <bed/gff/vcf> -g <genome> [-b <int> or (-l and -r)]

Options:

-b Increase the BED/GFF/VCF entry by -b base pairs in each direction. - (Integer) or (Float, e.g. 0.1) if used with -pct.

-l The number of base pairs to subtract from the start coordinate. - (Integer) or (Float, e.g. 0.1) if used with -pct.

-r The number of base pairs to add to the end coordinate. - (Integer) or (Float, e.g. 0.1) if used with -pct.

-s Define -l and -r based on strand. E.g. if used, -l 500 for a negative-stranded feature, it will add 500 bp downstream. Default = false.

-pct Define -l and -r as a fraction of the feature's length. E.g. if used on a 1000bp feature, -l 0.50, will add 500 bp "upstream". Default = false.

Notes:

1. Starts will be set to 0 if options would force it below 0.

(2) Ends will be set to the chromosome length if requested slop would force it above the max chrom length. (3) The genome file should tab delimited and structured as follows:

<chromName><TAB><chromSize>

For example, Human (hg19): chr1 249250621 chr2 243199373 ... chr18**gl000207**random 4262

Tips: One can use the UCSC Genome Browser's MySQL database to extract chromosome sizes. For example, H. sapiens:

mysql –user=genome –host=genome-mysql.cse.ucsc.edu -A -e / "select chrom, size from hg19.chromInfo" > hg19.genome

`shuffle()` (wraps "shuffleBed")

`BedTool.`**`shuffle`**`(`*`*args`*`, `*`**kwargs`*`)`

pybedtools help:

Shuffle coordinates.

Example usage:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> seed = 1 # so this test always returns the same results
>>> b = a.shuffle(genome='hg19', chrom=True, seed=seed)
>>> print b
chr1    59535036        59535135        feature1        0       +
chr1    99179023        99179123        feature2        0       +
chr1    186189051       186189401       feature3        0       -
chr1    219133189       219133239       feature4        0       +
<BLANKLINE>
```

Note: For convenience, the file this bedtool object points to is passed as "-i"

Original BEDtools program help:

Program: shuffleBed (v2.12.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Randomly permute the locations of a feature file among a genome.

Usage: shuffleBed [OPTIONS] -i <bed/gff/vcf> -g <genome>

Options:

-excl A BED/GFF/VCF file of coordinates in which features in -i should not be placed (e.g. gaps.bed).

-incl Instead of randomly placing features in a genome, the -incl options defines a BED/GFF/VCF file of coordinates in which features in -i should be randomly placed (e.g. genes.bed).

-chrom Keep features in -i on the same chromosome. - By default, the chrom and position are randomly chosen.

-seed Supply an integer seed for the shuffling. - By default, the seed is chosen automatically. - (INTEGER)

-f Maximum overlap (as a fraction of the -i feature) with an -excl feature that is tolerated before searching for a new, randomized locus. For example, -f 0.10 allows up to 10% of a randomized feature to overlap with a given feature in the -excl file. **Cannot be used with -incl file.** - Default is 1E-9 (i.e., 1bp). - FLOAT (e.g. 0.50)

Notes:

1. The genome file should tab delimited and structured as follows: <chromName><TAB><chromSize>

For example, Human (hg19): chr1 249250621 chr2 243199373 ... chr18**gl000207**random 4262

Tips: One can use the UCSC Genome Browser's MySQL database to extract chromosome sizes. For example, H. sapiens:

mysql –user=genome –host=genome-mysql.cse.ucsc.edu -A -e / "select chrom, size from hg19.chromInfo" > hg19.genome

`annotate()` (wraps "annotateBed")

`BedTool.`**`annotate`**`(`**args*, ***kwargs*`)`

pybedtools help:

Annotate this BedTool with a list of other files. Example usage:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> b_fn = pybedtools.example_filename('b.bed')
>>> print a.annotate(files=b_fn)
chr1    1       100     feature1        0       +       0.000000
chr1    100     200     feature2        0       +       0.450000
chr1    150     500     feature3        0       -       0.128571
chr1    900     950     feature4        0       +       0.020000
<BLANKLINE>
```

Note: This method returns a new bedtool instance

Note: For convenience, the file this bedtool object points to is passed as "-i"

Original BEDtools program help:

Program: annotateBed (v2.11.2) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Annotates the depth & breadth of coverage of features from multiple files

on the intervals in -i.

Usage: annotateBed [OPTIONS] -i <bed/gff/vcf> -files FILE1 FILE2 .. FILEn

Options:

-names A list of names (one / file) to describe each file in -i. These names will be printed as a header line.

-counts Report the count of features in each file that overlap -i.

- Default is to report the fraction of -i covered by each file.

-both Report the counts followed by the % coverage. - Default is to report the fraction of -i covered by each file.

-s Force strandedness. That is, only include hits in A that overlap B on the same strand. - By default, hits are included without respect to strand.

`coverage()` (wraps "coverageBed")

`BedTool.coverage(*args, **kwargs)`

pybedtools help:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> b = pybedtools.example_bedtool('b.bed')
>>> c = a.coverage(b)
>>> c.head(3)
chr1    155     200     feature5        0       -       2       45      45      1.0000000
chr1    800     901     feature6        0       +       1       1       101     0.0099010
```

Note: This method returns a new bedtool instance

Note: For convenience, the file this bedtool object points to is passed as "-a"

Original BEDtools program help:

Program: coverageBed (v2.12.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Returns the depth and breadth of coverage of features from A

on the intervals in B.

Usage: coverageBed [OPTIONS] -a <bed/gff/vcf> -b <bed/gff/vcf>

Options:

-abam The A input file is in BAM format.

-s Force strandedness. That is, only include hits in A that overlap B on the same strand. - By default, hits are included without respect to strand.

-hist Report a histogram of coverage for each feature in B as well as a summary histogram for **all** features in B.

Output (tab delimited) after each feature in B:

1. depth
2. # bases at depth
3. size of B
4. % of B at depth

-d Report the depth at each position in each B feature. Positions reported are one based. Each position and depth follow the complete B feature.

-split Treat "split" BAM or BED12 entries as distinct BED intervals. when computing coverage. For BAM files, this uses the CIGAR "N" and "D" operations to infer the blocks for computing coverage. For BED12 files, this uses the BlockCount, BlockStarts, and BlockEnds fields (i.e., columns 10,11,12).

Default Output:

After each entry in B, reports:

1. The number of features in A that overlapped the B interval.
2. The number of bases in B that had non-zero coverage.
3. The length of the entry in B.
4. The fraction of bases in B that had non-zero coverage.

2.6.2 Methods for converting between formats

`bed6()` (wraps "Bed12To6")

`BedTool.`**`bed6`**`(`**args*, ***kwargs*`)`

pybedtools help:

convert a BED12 to a BED6 file

Note: This method returns a new bedtool instance

Note: For convenience, the file this bedtool object points to is passed as "-i"

Note: This method accepts either a bedtool or a file name as the first unnamed argument

Original BEDtools program help:

Program: bed12ToBed6 (v2.11.2) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Splits BED12 features into discrete BED6 features.

Usage: bed12ToBed6 [OPTIONS] -i <bed12>

Options:

-n	Force the score to be the (1-based) block number from the BED12.

2.6.3 Methods for working with sequences

`sequence()` (wraps "fastaFromBed")

`BedTool.`**`sequence`**`(`*fi*, ***kwargs*`)`

pybedtools help:

Wraps `fastaFromBed`. *fi* is passed in by the user; *bed* is automatically passed in as the bedfile of this object; *fo* by default is a temp file. Use save_seqs() to save as a file.

The end result is that this BedTool will have an attribute, self.seqfn, that points to the new fasta file.

Example usage:

```
>>> a = pybedtools.BedTool("""
```

... chr1 1 10 ... chr1 50 55”“”, from_string=True) >>> fasta = pybedtools.example_filename(‘test.fa’) >>> a = a.sequence(fi=fasta) >>> print open(a.seqfn).read() >chr1:1-10 GATGAGTCT >chr1:50-55 CCATC <BLANKLINE>

Note: This method returns a new bedtool instance

Note: For convenience, the file this bedtool object points to is passed as “-bed”

Original BEDtools program help:

Program: fastaFromBed (v2.12.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Extract DNA sequences into a fasta file based on feature coordinates.

Usage: fastaFromBed [OPTIONS] -fi <fasta> -bed <bed/gff/vcf> -fo <fasta>

Options:

-fi	Input FASTA file
-bed	BED/GFF/VCF file of ranges to extract from -fi
-fo	Output file (can be FASTA or TAB-delimited)
-name	Use the name field for the FASTA header
-tab	Write output in TAB delimited format. - Default is FASTA format.
-s	Force strandedness. If the feature occupies the antisense strand, the sequence will be reverse complemented. - By default, strand information is ignored.

mask_fasta() (wraps “maskFastaFromBed”)

BedTool.**mask_fasta**(**args*, ***kwargs*)

pybedtools help:

Masks a fasta file at the positions in a BED file and saves result as *out*. This method returns None, and sets self.seqfn to *out*.

```
>>> a = pybedtools.BedTool('chr1 100 110', from_string=True)
>>> fasta_fn = pybedtools.example_filename('test.fa')
>>> a = a.mask_fasta(fi=fasta_fn, fo='masked.fa.example')
>>> b = a.slop(b=2, genome='hg19')
>>> b = b.sequence(a.seqfn)
>>> print b.print_sequence()
>chr1:98-112
TTNNNNNNNNNNAT
<BLANKLINE>
>>> os.unlink('masked.fa.example')
>>> if os.path.exists('masked.fa.example.fai'):
...     os.unlink('masked.fa.example.fai')
```

Note: This method returns a new bedtool instance

Note: For convenience, the file this bedtool object points to is passed as “-bed”

Original BEDtools program help:

Program: maskFastaFromBed (v2.11.2) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Mask a fasta file based on feature coordinates.

Usage: maskFastaFromBed [OPTIONS] -fi <fasta> -out <fasta> -bed <bed/gff/vcf>

Options:

-fi	Input FASTA file
-bed	BED/GFF/VCF file of ranges to mask in -fi
-fo	Output FASTA file
-soft	Enforce "soft" masking. That is, instead of masking with Ns, mask with lower-case bases.
-mc	Replace masking character. That is, instead of masking with Ns, use another character.

2.7 BedTool methods unique to pybedtools

2.7.1 Introspection

count()

BedTool.**count**()

Number of features in BED file. Does the same thing as len(self), which actually just calls this method.

Only counts the actual features. Ignores any track lines, browser lines, lines starting with a "#", or blank lines.

Example usage:

```
a = BedTool('in.bed')
a.count()
```

print_sequence()

BedTool.**print_sequence**()

Print the sequence that was retrieved by the BedTool.sequence() method.

See usage example in BedTool.sequence().

file_type()

BedTool.**file_type**()

Return the type of the current file. One of ('bed','vcf','gff').

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> a.file_type
'bed'
```

field_count()

BedTool.**field_count**(*n=10*)
Return the number of fields in the features this file contains. Checks the first *n* features.

2.7.2 Saving

saveas()

BedTool.**saveas**(*fn*, *trackline=None*)

Save BED file as a new file, adding the optional *trackline* to the beginning.

Returns a new BedTool for the newly saved file.

A newline is automatically added to the trackline if it does not already have one.

Example usage:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> b = a.saveas('other.bed')
>>> b.fn
```

'other.bed' >>> print b == a True

```
>>> b = a.saveas('other.bed', trackline="name='test run' color=0,55,0")
>>> open(b.fn).readline()
"name='test run' color=0,55,0\n"
```

Note: This method returns a new bedtool instance

save_seqs()

BedTool.**save_seqs**(*fn*)
Save sequences of features in this BedTool object as a fasta file *fn*.

In order to use this function, you need to have called the `BedTool.sequence()` method.

A new BedTool object is returned which references the newly saved file.

Example usage:

```
a = BedTool('in.bed')

# specify the filename of the genome in fasta format
a.sequence('data/genomes/genome.fa')

# use this method to save the seqs that correspond to the features
# in "a"
a.save_seqs('seqs.fa')
```

2.7.3 Utilities

with_attrs()

BedTool.**with_attrs**(***kwargs*)

Given arbitrary keyword arguments, turns the keys and values into attributes. Useful for labeling BedTools at creation time.

Example usage:

```
>>> # add a "label" attribute to each BedTool
>>> a = pybedtools.example_bedtool(’a.bed’)                                  .with_attrs(la
>>> b = pybedtools.example_bedtool(’b.bed’)                                  .with_attrs(la
>>> for i in [a, b]:
```

... print i.count(), ‘features for’, i.label 4 features for transcription factor 1 2 features for transcription factor 2

Note: This method returns a new bedtool instance

cat()

BedTool.**cat**(*other*, *postmerge=True*, ***kwargs*)

Concatenates two BedTool objects (or an object and a file) and does an optional post-merge of the features.

Use *postmerge=False* if you want to keep features separate.

TODO:

currently truncates at BED3 format!

kwargs are sent to `BedTool.merge()`.

Example usage:

```
>>> a = pybedtools.example_bedtool(’a.bed’)
>>> b = pybedtools.example_bedtool(’b.bed’)
>>> print a.cat(b)
```

chr1 1 500 chr1 800 950 <BLANKLINE>

Note: This method returns a new bedtool instance

Note: This method accepts either a bedtool or a file name as the first unnamed argument

total_coverage()

BedTool.**total_coverage**()

Returns the total number of bases covered by this BED file. Does a self.merge() first to remove potentially multiple-counting bases.

Example usage:

```
>>> a = pybedtools.example_bedtool(’a.bed’)
```

This does a merge() first, so this is what the total coverage is counting:

```
>>> print a.merge()
chr1    1       500
chr1    900     950
<BLANKLINE>

>>> print a.total_coverage()
549
```

delete_temporary_history()

`BedTool.`**delete_temporary_history**(_ask=True_, _raw_input_func=None_)

Use at your own risk! This method will delete temp files. You will be prompted for deletion of files unless you specify _ask=False_.

Deletes all temporary files created during the history of this BedTool up to but not including the file this current BedTool points to.

Any filenames that are in the history and have the following pattern will be deleted:

```
<TEMP_DIR>/pybedtools.*.tmp
```

(where <TEMP_DIR> is the result from get_tempdir() and is by default "/tmp")

Any files that don't have this format will be left alone.

(_raw_input_func_ is used for testing)

2.7.4 Feature-by-feature operations

each()

`BedTool.`**each**(_func_, _*args_, _**kwargs_)

Applies user-defined function _func_ to each feature. _func_ must accept an Interval as its first argument; _args and **kwargs will be passed to *func_.

func must return an Interval object.

```
>>> def truncate_feature(feature, limit=0):
...     feature.score = str(len(feature))
...     if len(feature) > limit:
...         feature.stop = feature.start + limit
...         feature.name = feature.name + '.short'
...     return feature

>>> a = pybedtools.example_bedtool('a.bed')
>>> b = a.each(truncate_feature, limit=100)
>>> print b
chr1    1       100     feature1        99      +
chr1    100     200     feature2        100     +
chr1    150     250     feature3.short  350     -
chr1    900     950     feature4        50      +
<BLANKLINE>
```

filter()

BedTool.**filter**(*func*, **args*, ***kwargs*)

Takes a function *func* that is called for each feature in the BedTool object and returns only those for which the function returns True.

*args and **kwargs are passed directly to *func*.

Returns a streaming BedTool; if you want the filename then use the .saveas() method.

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> subset = a.filter(lambda b: b.chrom == 'chr1' and b.start < 150)
>>> len(a), len(subset)
(4, 2)
```

so it has extracted 2 records from the original 4.

cut()

BedTool.**cut**(*indexes*)

Similar to unix cut except indexes are 0-based, must be a list and the columns are returned in the order requested.

In addition, indexes can contain keys of the GFF/GTF attributes, in which case the values are returned. e.g. ‘gene_name’ will return the corresponding name from a GTF, or ‘start’ will return the start attribute of a BED Interval.

See .with_column() if you need to do more complex operations.

features()

BedTool.**features**()

Returns an iterator of feature objects.

2.7.5 Randomization helpers

randomintersection()

BedTool.**randomintersection**(*other*, *iterations*, *intersect_kwargs=None*, *shuffle_kwargs=None*, *debug=False*)

Performs *iterations* shufflings of self, each time intersecting with *other*.

Returns a generator of integers where each integer is the number of intersections of a shuffled file with *other*. This distribution can be used in downstream analysis for things like empirical p-values.

intersect_kwargs and *shuffle_kwargs* are passed to self.intersect() and self.shuffle() respectively. By default for intersect, u=True is specified – but s=True might be a useful option for strand-specific work.

Useful kwargs for *shuffle_kwargs* are chrom, excl, or incl. If you use the “seed” kwarg, that seed will be used *each* time shuffleBed is called – so all your randomization results will be identical for each iteration. To get around this and to allow for tests, debug=True will set the seed to the iteration number.

Example usage:

```
>>> chromsizes = {'chr1':(0, 1000)}
>>> a = pybedtools.example_bedtool('a.bed').set_chromsizes(chromsizes)
>>> b = pybedtools.example_bedtool('b.bed')
>>> results = a.randomintersection(b, 10, debug=True)
>>> print list(results)
```

[2, 2, 2, 0, 2, 3, 2, 1, 2, 3]

randomstats()

`BedTool.randomstats`(*other*, *iterations*, ***kwargs*)

Sends args and **kwargs to `BedTool.randomintersection()` and compiles results into a dictionary with useful stats. Requires scipy and numpy.

This is one possible way of assigning significance to overlaps between two files. See, for example:

> Negre N, Brown CD, Shah PK, Kheradpour P, Morrison CA, et al. 2010 A Comprehensive Map of Insulator Elements for the Drosophila Genome. PLoS Genet 6(1): e1000814. doi:10.1371/journal.pgen.1000814

Example usage:

Make chromsizes a very small genome for this example: >>> chromsizes = {'chr1':(1,1000)} >>> a = pybedtools.example_bedtool('a.bed').set_chromsizes(chromsizes) >>> b = pybedtools.example_bedtool('b.bed') >>> results = a.randomstats(b, 100, debug=True)

results is a dictionary that you can inspect. The actual overlap: >>> print results['actual'] 3

The median of all randomized overlaps: >>> print results['median randomized'] 2.0

The percentile of the actual overlap in the distribution of randomized overlaps, which can be used to get an empirical p-value: >>> print results['percentile'] 90.0

CHAPTER THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX