pybedtools Documentation

Release 0.2.0dev

Ryan Dale

January 27, 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 at http://daler.github.com/pybedtools.

CHAPTER TWO

CONTENTS:

2.1 Installation

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

1. To install the latest version of `pybedtools`:
   - 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)
2. 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*. 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(f
```

For more, continue on to the *Tutorial*.

2.3 Tutorial

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)

Follow the *Installation* instructions if you haven't already done so to install both BEDTools and `pybedtools`.

2.3.1 A brief note on conventions

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

- Python variables and arguments are shown in italics: *s=True*
- Files look like this: `filename.bed`
- Methods, which are often linked to documentation look like this: `bedtool.merge()`.
- BEDTools programs look like this: `intersectBed`.
- 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 (don't type the >>>)

Onward!

2.3.2 Create a `bedtool`

First, import the `pybedtools` module:

```
>>> import pybedtools
```

Then set up a `bedtool` instance using a BED format file. This can be a file that you already have, or one of the example files as shown below. Currently, only BED format files are supported – see *Limitations* for more info on this.

For this tutorial, we'll use some example files that come with `pybedtools`. We can get the filename for the example files using the `pybedtools.example_files()` function:

```
>>> # get an example filename to use
>>> bed_filename_a = pybedtools.example_bed_fn('a.bed')
```

The filename will depend on where you have installed `pybedtools`. Once you have a filename, creating a `bedtool` object is easy:

```
>>> # create a new bedtool using that filename
>>> a = pybedtools.bedtool(bed_filename_a)
```

Set up a second one so we can do intersections and subtractions – this time, let's make a new `bedtool` all in one line:

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

See *Creating a bedtool* for more information, including convenience functions for working with example bed files and making `bedtool` objects directly from strings.

2.3.3 Intersections

Here's how to intersect *a* with *b*:

```
>>> 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 get 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   +
```

a_with_b is another, different temp file whose name is stored in *a_with_b.fn*. You can read more about the use of temp files in *Principle 1: Temporary files are created automatically*. 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:

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

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*. 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.

OK, enough about arguments for now, but you can read more about them in *Principle 2: Names and arguments are as similar as possible to BEDTools*, *Principle 3: Sensible default args* and *Principal 4: Other arguments have no defaults*.

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.

```
>>> # Intersect and then merge all on one line, displaying the first
>>> # 10 lines of the results
>>> a.intersect(b, u=True).merge().head()
chr1    100 500
chr1    900 950
```

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 *Principle 6: Check the help*). 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:

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

Now we have new files in the current directory called `a-with-b.bed` and `a-with-b-merged.bed`.

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:

```
>>> result_1 = a.intersect(b, u=True)
>>> result_2 = a+b

>>> # To test equality, convert to strings
>>> str(result_1) == str(result_2)
True
```

And the − operator assumes `intersectBed`'s `-v` option:

```
>>> result_1 = a.intersect(b, v=True)
>>> result_2 = a-b

>>> # To test equality, convert to strings
>>> str(result_1) == str(result_2)
True
```

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:

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

You can learn more about chaining in *Principle 5: Chaining together commands*.

2.3.7 Methods specific to `pybedtools`

In no particular order, here are some other useful things that `bedtool` objects can do.

Counting

We can easily count features in `bedtool` objects:

```
>>> a.count()
4

>>> a.intersect(b, u=True).merge().count()
2
```

Lengths of features

We can get the lengths of features, which is useful for things like getting a histogram of binding site sizes after running a peak-calling algorithm on ChIP-seq data:

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

>>> a.lengths()
[99, 100, 350, 50]
```

Creating genome files ('chromSizes')

Some BEDTools programs require a "chromSizes" file to prevent out-of-range coordinates. `bedtool` objects have a convenience method to get this file for you and return the resulting filename:

```
>>> # Download Drosophila melanogaster chromSizes table from UCSC
>>> chromsizes = a.get_genome('hg19', fn='hg19.genome')

>>> # print first few lines of that file
>>> f = open(chromsizes)
>>> for i in range(5):
...     print f.readline(),
chr1    249250621
chr2    243199373
chr3    198022430
chr4    191154276
chr5    180915260
```

Feature centers

We can get the midpoints of features, with `bedtool.feature_centers()`, which is useful for things like extracting out the center N bp of features for de novo motif detection.

```
>>> a.feature_centers(n=50).saveas('middle-100-bp.bed')
<bedtool (middle-100-bp.bed)>
```

Filtering

Only get features of a certain size:

```
>>> # only features that are smaller than 60 bp
>>> a.size_filter(min=0, max=60).head()
chr1    900 950 feature4    0.0 +
```

Working with counted intersections

There are several methods that help you deal with counted intersections – the files that are returned when you use the *c=True* kwarg with `bedtool.intersect()`. First, do an intersection:

```
>>> print a.intersect(b, c=True)
chr1    1   100 feature1    0   +   0
chr1    100 200 feature2    0   +   1
chr1    150 500 feature3    0   -   1
chr1    900 950 feature4    0   +   1
```

You can retrieve these counts later using the `bedtool.counts()` method:

```
>>> result = a.intersect(b, c=True)
>>> print result.counts()
[0, 1, 1, 1]
```

2.4 Topical documentation

2.4.1 Future work

The following is a list of items I plan to work on for future releases of `pybedtools`. Help and feedback are welcome!:

- Better mechanism for handling temp files.

    - `bedtool` objects could keep track of all their 'parent' tempfiles, and as such would retain the history of their creation.
    - indexing into the history of a `bedtool` would give you access to these previous files
    - could have a `bedtool.delete_history()`, which would delete all the intermediate tempfiles on disk, cleaning up only for this particular `bedtool`
- Universal "interval" file support
    - parent `UniversalInterval` class, GFF, GTF, VCF, etc. etc. all subclassed from that
    - `bedtool` should auto-detect the interval file format
- Support for "derived" file types
    - currently the handling of output created by `a.intersect(b, c=True)` is special-cased, and `a.intersect(b, wao=True)` is unsupported by custom `bedtool` methods.
    - **these should probably just be implemented as subclasses of the as-yet** hypothetical `UniversalInterval` class.
- Support for `groupBy`
    - it would be nice to have support for filo
- Randomization methods are still under development

2.4.2 Why use `pybedtools`?

`pybedtools` makes working with BEDTools from Python code easy.

Calling BEDTools from Python "by hand" gets awkward for things like geting the number of intersecting features between two bed files, for example:

```
>>> # The annoying way of calling BEDTools from Python...
>>> p1 = subprocess.Popen(['intersectBed','-a','a.bed','-b','b.bed','-u'], stdout=subprocess.PIPE)

>>> # then pipe it to wc -l to get a line count
>>> p2 = subprocess.Popen(['wc','-l'], stdin=subprocess.PIPE)

>>> # finally, parse the results
>>> results = p2.communicate()[0]
>>> count = int(results.split()[-1])
```

If we wanted to get the number of features unique to `a.bed` , it would mean another 4 lines of this, with the only difference being the `-v` argument instead of `-u` for the `intersectBed` call.. For me, this got old quickly, hence the creation of `pybedtools`.

Here's how to do the same thing with `pybedtools`:

```
>>> from pybedtools import bedtool
>>> a = bedtool('a.bed')
>>> count = a.intersect('b.bed', u=True).count()
```

Behind the scenes, the `pybedtools.bedtool` class does something very similar to the subprocess example above, but in a more Python-friendly way.

Furthermore, for the specific case of intersections, the + and − operators have been overloaded, making many intersections extremely easy:

```
>>> a = bedtool('a.bed')
>>> b = bedtool('b.bed')
>>> c = bedtool('c.bed')

>>> (a+b).count()   # number of features in a and b
>>> (a-b).count()   # number of features in a not b
>>> (a+b+c).count() # number of features in a, b and c
```

The other BEDTools programs are wrapped as well, like `bedtool.merge()`, `bedtool.slop()`, and others.

In addition to wrapping the BEDtools programs, there are many additional `bedtool` methods provided in this module that you can use in your Python code.

2.4.3 Limitations

There are some limitations you need to be aware of.

- `pybedtools` makes heavy use of temporary files. This makes it very convenient to work with, but if you are limited by disk space, you'll have to pay attention to this feature (see temp principle below for more info).
- Second, `bedtool` methods that wrap BEDTools programs will work on BAM, GFF, VCF, and everything that BEDTools supports. However, many `pybedtools`-specific methods (for example `bedtool.lengths()` or `bedtool.size_filter()`) **currently only work on BED files**. I hope to add support for all interval files soon.

2.4.4 Creating a `bedtool`

To create a `bedtool`, first you need to import the `pybedtools` module. For these examples, I'm assuming you have already done the following:

```
>>> import pybedtools
>>> from pybedtools import bedtool
```

Next, you need a BED file to work with. If you already have one, then great – move on to the next section. If not, `pybedtools` comes with some example bed files used for testing. You can take a look at the list of example files that ship with `pybedtools` with the `list_example_beds()` function:

```
>>> # list the example bed files
>>> pybedtools.list_example_beds()
['a.bed', 'b.bed']
```

Once you decide on a file to use, feed the your choice to the `example_bed_fn()` function to get the full path:

```
>>> # get the full path to an example bed file
>>> bedfn = pybedtools.example_bed_fn('a.bed')
```

The full path of *bedfn* will depend on your installation (this is similar to the `data()` function in R, if you're familiar with that).

Now that you have a filename – either one of the example files or your own, you create a new `bedtool` simply by pointing it to that filename:

```
>>> # create a new bedtool from the example bed file
>>> mybedtool = bedtool(bedfn)
```

Alternatively, you can construct BED files from scratch by using the `from_string` keyword argument. However, all spaces will be converted to tabs using this method, so you'll have to be careful if you add "name" columns. This can be useful if you want to create *de novo* BED files on the fly:

```
>>> # an "inline" example:
>>> fromscratch1 = pybedtools.bedtool('chrX 1 100', from_string=True)
>>> print fromscratch1
chrX    1   100
```

```
>>> # using a longer string to make a bed file.  Note that
>>> # newlines don't matter, and one or more consecutive
>>> # spaces will be converted to a tab character.
>>> larger_string = """
... chrX 1     100   feature1  0 +
... chrX 50    350   feature2  0 -
... chr2 5000 10000 another_feature 0 +
... """

>>> fromscratch2 = bedtool(larger_string, from_string=True)
>>> print fromscratch2
chrX    1   100 feature1    0   +
chrX    50  350 feature2    0   -
chr2    5000    10000   another_feature 0   +
```

Of course, you'll usually be using your own bed files that have some biological importance for your work that are saved in places convenient for you, for example:

```
>>> a = bedtool('/data/sample1/peaks.bed')
```

2.4.5 Design principles: an example

Principle 1: Temporary files are created automatically

Let's illustrate some of the design principles behind `pybedtools` by merging features in `a.bed` that are 100 bp or less apart (*d=100*) in a strand-specific way (*s=True*):

```
>>> from pybedtools import bedtool
>>> import pybedtools
>>> a = bedtool(pybedtools.example_bed_fn('a.bed'))
>>> merged_a = a.merge(d=100, s=True)
```

Now *merged_a* is a `bedtool` instance that contains the results of the merge.

`bedtool` objects must always point to a file on disk. So in the example above, *merged_a* is a `bedtool`, but what file does it point to? You can always check the `bedtool.fn` attribute to find out:

```
>>> # what file does *merged_a* point to?
>>> merged_a.fn
'/tmp/pybedtools.MPPp5f.tmp'
```

Note that the specific filename will be different for you since it is a randomly chosen name (handled by Python's `tempfile` module). This shows one important aspect of `pybedtools`: every operation results in a new temporary file. Temporary files are stored in `/tmp` by default, and have the form `/tmp/pybedtools.*.tmp`.

Future work on `pybedtools` will focus on streamlining the temp files, keeping only those that are needed. For now, when you are done using the `pybedtools` module, make sure to clean up all the temp files created with:

```
>>> # Deletes all tempfiles created this session.
>>> # Don't do this yet if you're following the tutorial!
>>> pybedtools.cleanup()
```

If you forget to do this, from the command line you can always do a:

```
rm /tmp/pybedtools.*.tmp
```

to clean everything up.

If you need to specify a different directory than that used by default by Python's tempdir module, then you can set it with:

```
>>> pybedtools.set_tempdir('/scratch')
```

You'll need write permissions to this directory, and it needs to already exist.

Principle 2: Names and arguments are as similar as possible to BEDTools

Returning again to this example:

```
>>> merged_a = a.merge(d=100, s=True)
```

Another `pybedtools` principle is that the `bedtool` methods that wrap BEDTools programs do the same thing and take the exact same arguments as the BEDTools program. Here we can pass *d=100* and *s=True* only because the underlying BEDTools program, `mergeBed`, can accept these arguments. Need to know what arguments `mergeBed` can take? See the docs for `bedtool.merge()`; for more on this see good docs principle.

In general, remove the "Bed" from the end of the BEDTools program to get the corresponding `bedtool` method. So there's a `bedtool.subtract()` method for `subtractBed`, a `bedtool.intersect()` method for `intersectBed`, and so on.

Since these methods just wrap BEDTools programs, they are as up-to-date as the version of BEDTools you have installed on disk. If you are using a cutting-edge version of BEDTools that has some hypothetical argument `-z` for `intersectBed`, then you can use `a.intersectBed(z=True)`.

Principle 3: Sensible default args

If we were running the `mergeBed` program from the command line, we would would have to specify the input file with the *`mergeBed -i`* option.

`pybedtools` assumes that if we're calling the `merge()` method on *a*, we want to operate on the bed file that *a* points to.

In general, BEDTools programs that accept a single BED file as input (by convention typically specified with the *`-i`* option) the default behavior for `pybedtools` is to use the `bedtool`'s file (indicated in the `bedtool.fn` attribute) as input.

We can still pass a file using the *i* keyword argument if we wanted to be absolutely explicit. In fact, the following two versions produce the same output:

```
>>> # The default is to use existing file for input -- no need
>>> # to specify "i" . . .
>>> result1 = a.merge(d=100, s=True)

>>> # . . . but you can always be explicit if you'd like
>>> result2 = a.merge(i=a.fn, d=100, s=True)

>>> # Confirm that the output is identical
>>> str(result1) == str(result2)
True
```

There are some BEDTools programs that accept two BED files as input, like `intersectBed` where the the first file is specified with *-a* and the second file with *-b*. The default behavior for `pybedtools` is to consider the `bedtool`'s file as `-a` and the first non-keyword argument to the method as `-b`, like this:

```
>>> result3 = a.intersect(b)
```

This is exactly the same as passing the *a* and *b* arguments explicitly:

```
>>> result4 = a.intersect(a=a.fn, b=b.fn)
>>> str(result3) == str(result4)
True
```

Furthermore, the first non-keyword argument used as `-b` can either be a filename *or* another `bedtool` object.

Principal 4: Other arguments have no defaults

`-d` is an option to BEDTools `mergeBed` that accepts a value, while `-s` is an option that acts as a switch. In `pybedtools`, simply pass a value (integer, float, whatever) for value-type options like `-d`, and boolean values (*True* or *False*) for the switch-type options like `-s`.

Here's another example using both types of keyword arguments; the `bedtool` object *b* (or it could be a string filename too) is implicitly passed to `intersectBed` as `-b` (see default args principle above):

```
>>> a.intersect(b, v=True, f=0.5)
```

Other than the *-i*, *-a*, and *-b* options for input files mentioned above, these other options like *-d* and *s* have no defaults.

Again, any option that can be passed to a BEDTools program can be passed to the corresonding `bedtool` method.

Principle 5: Chaining together commands

Most methods return new `bedtool` objects, allowing you to chain things together just like piping commands together on the command line. To give you a flavor of this, here is how you would get the merged regions of features shared between `a.bed` (as referred to by the `bedtool` *a* we made previously) and `b.bed`: (as referred to by the `bedtool` *b*):

```
>>> a.intersect(b).merge().saveas('shared_merged.bed')
<bedtool (shared_merged.bed)>
```

This is equivalent to the following BEDTools commands:

```
intersectBed -a a.bed -b b.bed | merge -i stdin > shared_merged.bed
```

Principle 6: Check the help

If you're unsure of whether a method uses a default, or if you want to read about what options an underlying BEDTools program accepts, check the help. Each `pybedtool` method that wraps a BEDTools program also wraps the BEDTools program help string. There are often examples of how to use a method in the docstring as well.

2.4.6 Example: Flanking seqs

The `bedtool.slop()` method (which calls `slopBed`) needs a chromosome size file. If you specify a genome name to the `bedtool.slop()` method, it will retrieve this file for you automatically from the UCSC Genome Browser MySQL database.

```
import pybedtools
a = pybedtools.bedtool('in.bed')
extended = a.slop(genome='dm3',l=100,r=100)
flanking = extended.subtract(a).saveas('flanking.bed')
flanking.sequence(fi='dm3.fa')
flanking.save_seqs('flanking.fa')
```

Or, as a one-liner:

```
pybedtools.bedtool('in.bed').slop(genome='dm3',l=100,r=100).subtract(a).sequence(fi='dm3.fa').save_se
```

Don't forget to clean up!:

```
pybedtools.cleanup()
```

2.4.7 Example: Region centers that are fully intergenic

Useful for, e.g., motif searching:

```
a = pybedtools.bedtool('in.bed')

# Sort by score
a = a.sorted(col=5,reverse=True)

# Exclude some regions
a = a.subtract('regions-to-exclude.bed')

# Get 100 bp on either side of center
a = a.peak_centers(100).saveas('200-bp-peak-centers.bed')
```

2.4.8 Example: Histogram of feature lengths

Note that you need matplotlib installed to plot the histogram.

```
import pylab as p
a = pybedtools.bedtool('in.bed')
p.hist(a.lengths(),bins=50)
p.show()
```

2.5 Module documentation

2.5.1 `pybedtools` module-level functions

pybedtools.**data_dir**()
Returns the data directory that contains example files for tests and documentation.

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

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

`pybedtools.`**`list_example_beds`**`()`

Returns a list of bed files in the examples dir. Choose one and pass it to `example_bed()` to get the full path to an example BED file.

Example usage:

```
>>> choices = list_example_beds()
>>> bedfn = example_bed(choices[0])
>>> mybedtool = bedtool(bedfn)
```

2.5.2 `bedtool` methods

class `pybedtools.`**`bedtool`**`(`*fn*, *genome=None*, *from_string=False*`)`

Wrapper around Aaron Quinlans `BEDtools` suite of programs (https://github.com/arq5x/bedtools); also contains many useful methods for more detailed work with BED files.

Typical usage is to point to an existing file:

a = bedtool('a.bed')

But you can also create one from scratch from a string:

s = ''‘ chrX 1 100 chrX 25 800 ''‘

a = bedtool(s,from_string=True).saveas('a.bed')

Or use examples that come with pybedtools:

```
example_beds = pybedtools.list_example_beds()
a = pybedtools.example_bedtool('a.bed')
```

`cat``(`*other*, *postmerge=True*, ***kwargs*`)`

Concatenates two bedtools 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 = bedtool('in.bed')

# concatenate and merge features together if they overlap and are
# on the same strand
b = a.cat('other.bed', s=True)
```

Note: This method returns a new bedtool instance

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

`closest``(`*other*, ***kwargs*`)`

pybedtools help:

Return a new bedtool object containing closest features in *other*. 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 (v2.10.0) 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

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()
```

counts()

After running `bedtool.intersect()` with the kwarg *c=True*, use this method to return a list of the count of features in "b" that intersected each feature in "a".

Example usage:

```
a = bedtool('in.bed')
b = a.intersect('other.bed', c=True)
counts = b.counts()

# assuming you have matplotlib installed, plot a histogram

import pylab
pylab.hist(counts)
pylab.show()
```

feature_centers(*n*, *report_smaller=True*)

Returns a new bedtools object with just the centers of size n extracted from this object's features.

If *report_smaller* is True, then report features that are smaller than *n*. Otherwise, ignore them.

Example usage:

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

# 5bp on either side of the center of each feature
b = a.feature_centers(100)
```

Note: This method returns a new bedtool instance

features()

Returns an iterator of `bedfeature` objects.

get_genome(*genome*, *fn=None*)

Download chrom size info for *genome* from UCSC, removes the header line, and saves in a temp file. Could be useful for end users, but mostly called internally by `bedtool.slop()` and other methods that need the genome file.

If *fn* is None, then saves as a temp file.

Returns the filename.

Example usage:

```
a = bedtool('in.bed')
fn = a.get_genome('dm3', 'dm3.genome')
```

head(*n=10*)

Prints the first *n* lines

intersect(*b=None*, ***kwargs*)

pybedtools help:

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

Example usage:

```
# create new bedtool object
a = bedtool('in.bed')

# get overlaps with "other.bed"
overlaps = a.intersect('other.bed')

# use v=True to get the inverse, or those unique to in.bed
unique_to_a = a.intersect('other.bed', v=True)

# features unique to "other.bed"
unique_to_other = bedtool('other.bed').intersect(a, 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 (v2.10.0) 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.

intersection_report(*other*, *basename=True*, ***kwargs*)

Prints a report of the reciprocal intersections with another bed file or `bedtool` object.

If *basename* is True (default), only prints the basename of the file and not the whole path.

a = bedtool('in.bed') a.intersection_report('other.bed')

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

lengths()

Returns a list of feature lengths.

Example usage:

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

lengths = a.lengths()

# if you have pylab installed, plot a histogram
import pylab
pylab.hist(lengths)
pylab.show()
```

merge(***kwargs*)

pybedtools help:

Merge overlapping features together. Returns a new bedtool object.

Example usage:

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

# allow merging of features 100 bp apart
b = a.merge(d=100)
```

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 (v2.10.0) 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.

newshuffle()

Quite fast implementation of shuffleBed; assumes 'dm3' as the genome and assumes shuffling within chroms.

Example usage:

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

# randomly shuffled version of "a"
b = a.newshuffle()
```

This is equivalent to the following command-line usage of `shuffleBed`:

```
shuffleBed -i in.bed -g dm3.genome -chrom -seed $RANDOM > /tmp/tmpfile
```

normalized_counts()

After running `bedtool.intersect()` with the kwarg *c=True*, use this method to return a list of the density of features in "b" that intersected each feature in "a".

This takes the counts in each feature and divides by the bp in that feature.

Example usage:

```
a = bedtool('in.bed')
b = a.intersect('other.bed', c=True)
counts = b.normalized_counts()

# assuming you have matplotlib installed, plot a histogram

import pylab
pylab.hist(counts)
pylab.show()
```

parse_kwargs(***kwargs*)

Given a set of keyword arguments, turns them into a command line-ready list of strings. E.g., the kwarg dict:

```
kwargs = dict(c=True,f=0.5)
```

will be returned as:

```
['-c','-f','0.5']
```

If there are symbols (e.g., "|"), then the parameter is quoted."

print_randomstats(*other*, *iterations*, *intersectkwargs={}*)

Nicely prints the reciprocal randomization of two files.

print_sequence()

Print the sequence that was retrieved by the `bedtool.sequence()` method.

See usage example in `bedtool.sequence()`.

random_subset(*n*)

Returns a new bedtools object containing a random subset of the features in this subset. Currently does so by reading in all features; future updates should fix this to something more robust (e.g., newlines in a memory map)

Example usage:

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

# Choose 5 random features from 'in.bed'
b = a.random_subset(5)
```

Note: This method returns a new bedtool instance

randomintersection(*other*, *iterations*, *intersectkwargs={}*)

Performs *iterations* shufflings of self, each time intersecting with *other*. *intersectkwargs* are passed to self.intersect(). Returns a list of integers where each integer is the number of intersections of one shuffled file with *other*.

Example usage:

```
r = bedtool('in.bed').randomintersection('other.bed', 100, {'u': True})
```

randomstats(*other*, *iterations*, *intersectkwargs={}*)

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

Example usage:

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

# Randomization results from 100 iterations, using the u=True kwarg (report
# features in "a" only once for each intersection).
results = a.randomstats('other.bed', iterations=100, intersectkwargs={'u':True})
```

rename_features(*new_name*)

Forces a rename of all features. Useful for if you have a BED file of exons and you want all of them to have the name "exon".

Note: This method returns a new bedtool instance

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')
```

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 = bedtool('in.bed')
b = a.random_subset(5)
b.saveas('random-5.bed',trackline='track name="random subset" color=128,128,255')
```

Note: This method returns a new bedtool instance

sequence (***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.

Example usage:

```
a = bedtool('in.bed')
a.sequence(fi='genome.fa')
a.print_sequence()
```

Note: This method returns a new bedtool instance

Original BEDtools program help:

Program: fastaFromBed (v2.10.0) 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.

`sequence_coverage``()`

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

Example usage:

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

# total bp in genome covered by 'in.bed'
total_bp = a.sequence_coverage()
```

`shuffle``(`*genome=None*, ***kwargs*`)`

pybedtools help:

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

Original BEDtools program help:

Program: shuffleBed (v2.10.0) 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).
-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)

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

size_filter (*min=0*, *max=1000000000000000.0*)

Returns a new bedtool object containing only those features that are > *min* and < *max*.

Example usage:

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

# Only return features that are over 10 bp.
b = a.size_filter(min=10)
```

slop (*genome=None*, ***kwargs*)

pybedtools help:

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

If *genome* is specified with a genome name, the genome file will be automatically retrieved from UCSC Genome Browser.

Example usage:

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

# increase the size of features by 100 bp in either direction
b = a.slop(genome='dm3', b=100)

# grow features by 10 bp upstream and 500 bp downstream,
# using a genome file you already have constructed called
# dm3.genome.
c = a.slop(g='dm3.genome', l=10, r=500, s=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: slopBed (v2.10.0) 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)
-l	The number of base pairs to subtract from the start coordinate. - (Integer)
-r	The number of base pairs to add to the end coordinate. - (Integer)
-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.

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

sort(***kwargs*)

pybedtools help:

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

Original BEDtools program help:

Program: sortBed (v2.10.0) 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).

sorted(*col*, *reverse=None*)

Returns a new bedtool object, sorted by the column specified. col can be a list of columns. BED columns that are ints (start, stop and value) will be sorted numerically; other columns will be alphabetical.

reverse is a list of booleans, same length as col, specifying which fields to reverse-sort.

TODO: currently multiple columns aren't working!

a = bedtool('in.fn') b = a.sorted(col=2) # sort by start position c = a.sorted(col=5,reverse=True) # reverse sort on the values

subtract(*other*, ***kwargs*)

pybedtools help:

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

Example usage:

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

# do a "stranded" subtraction
b = a.subtract('other.bed',s=True)
```

```
# Require 50% of features in a to overlap
c = a.subtract('other.bed', s=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 (v2.10.0) 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.

tostring()

Returns the BED file as a string. You can also `print` the bedtool object to view its contents.

Example usage:

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

# this is one looong string which contains the entire file
long_string = a.tostring()
```

window(*other*, ***kwargs*)

pybedtools help:

Intersect with a window.

Example usage:

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

# Consider features up to 500 bp away as overlaps
b = a.window(w=500)
```

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 (v2.10.0) 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."

with_attrs(***kwargs*)

Given arbitrary keyword arguments, turns the keys and values into attributes.

Example usage:

```
# add a "label" attribute to each bedtool
a = bedtool('a.bed').with_attrs(label='transcription factor 1')
b = bedtool('b.bed').with_attrs(label='transcription factor 2')
for i in [a,b]:
    print i.count(), 'features for', i.label
```

Note: This method returns a new bedtool instance

CHAPTER

THREE

INDICES AND TABLES

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

PYTHON MODULE INDEX

INDEX