pybedtools Documentation

Release 0.2.3dev

Ryan Dale

April 29, 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:

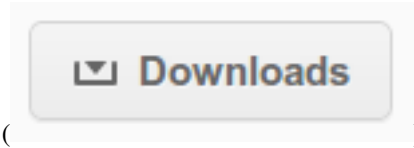
```
pip install cython argparse
```

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

Then 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. Some methods currently only work for BED files – 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_filename('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_filename('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 = pybedtools.example_bedtool('a.bed')
>>> b = pybedtools.example_bedtool('b.bed')
>>> 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 = pybedtools.example_bedtool('a.bed')
>>> b = pybedtools.example_bedtool('b.bed')
>>> 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 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, to only get features of a certain size:

```
>>> 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 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):
...     return len(feature) > L

>>> 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       -
```

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:

```
>>> 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       -
```

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
```

2.3.8 Each

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.

```
>>> from pybedtools import featurefuncs
>>> a = pybedtools.example_bedtool('a.bed')
>>> b = pybedtools.example_bedtool('b.bed')
>>> with_counts = a.intersect(b, c=True)
>>> 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
```

2.3.9 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.

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.

2.3.10 More documentation

For more info, see the *Topical documentation*.

2.4 Topical documentation

2.4.1 Release notes

0.2.3dev

- Added history mechanism – see *Using the history and tags*.
- Added tagging mechanism

2.4.2 Future work

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

- Finish wrapping the rest of the BEDTools programs
- **DONE:** Better mechanism for handling temp files.
  - **DONE:** `BedTool` objects could keep track of all their 'parent' tempfiles, and as such would retain the history of their creation.
  - **N/A:** indexing into the history of a `BedTool` would give you access to these previous files. *not implemented – complex histories would require a tree, which is not easily indexed. Instead I'm using a "tag" system*
  - **DONE:** 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
  - Should have a parent `UniversalInterval` class and then have 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 still under development
- Have wrappers use subprocess.Popen instead of os.system calls for better trapping of errors
- More universal wrapper – perhaps as decorator?
- Full unit tests to bolster the tutorial doctests

2.4.3 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.

2.4.4 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).
- `BedTool` methods that wrap BEDTools programs (`BedTool.intersect()`, `BedTool.merge()`, `BedTool.subtract()`, etc) 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.
- **Not all BEDTools programs are wrapped** – this package is still a work in progress. I wrapped the ones I use most often and still need to wrap the others. The following table shows what's currently wrapped:

BEDTools program	`BedTool` method name
`intersectBed`	`BedTool.intersect()`
`subtractBed`	`BedTool.subtract()`
`fastaFromBed`	`BedTool.sequence()`
`slopBed`	`BedTool.slop()`
`windowBed`	`BedTool.window()`
`closestBed`	`BedTool.closest()`
`shuffleBed`	`BedTool.shuffle()`

2.4.5 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_files()` function:

```
>>> # list the example bed files
>>> pybedtools.list_example_files()
['a.bed', 'b.bed', 'c.gff', 'd.gff', 'rmsk.hg18.chr21.small.bed', 'rmsk.hg18.chr21.small.bed.gz']
```

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

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

2.4.6 Design principles

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_filename('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)
```

This demonstrates another pybedtools principle: 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 *Principle 6: Check the help*.

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
```

Methods that have this type of default behavior are indicated by the following text in their docstring:

```
.. note::

    For convenience, the file this BedTool object points to is passed as "-i"
```

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:

```
>>> b = pybedtools.BedTool(pybedtools.example_filename('b.bed'))
>>> 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; that is, these commands also do the same thing:

```
>>> result5 = a.intersect(b=b.fn)
>>> result6 = a.intersect(b=b)
>>> str(result5) == str(result6)
True
```

Methods that accept either a filename or another `BedTool` instance as their first non-keyword argument are indicated by the following text in their docstring:

```
.. note::

    This method accepts either a BedTool or a file name as the first
    unnamed argument
```

Principal 4: Other arguments have no defaults

Only the BEDTools arguments that refer to BED (or other interval) files have defaults. In the current version of BEDTools, this means only the `-i`, `-a`, and `-b` arguments have defaults. All others have no defaults specified by `pybedtools`; they pass the buck to BEDTools programs. This means if you do not specify the *d* kwarg when calling `BedTool.merge()`, then it will use whatever the installed version of BEDTools uses for `-d` (currently, `mergeBed`'s default for `-d` is 0).

`-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 *Principle 3: Sensible default args* above):

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

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
```

Methods that return a new `BedTool` instance are indicated with the following text in their docstring:

```
.. note::

    This method returns a new BedTool instance
```

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.

2.4.7 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()
```

2.4.8 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')
```

2.4.9 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.**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', (1, 23011544))
('chr2LHet', (1, 368872))
('chr2R', (1, 21146708))
('chr2RHet', (1, 3288761))
('chr3L', (1, 24543557))
('chr3LHet', (1, 2555491))
('chr3R', (1, 27905053))
('chr3RHet', (1, 2517507))
('chr4', (1, 1351857))
('chrM', (1, 19517))
('chrU', (1, 10049037))
('chrUextra', (1, 29004656))
('chrX', (1, 22422827))
('chrXHet', (1, 204112))
('chrYHet', (1, 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.**data_dir**()

Returns the data directory that contains example files for tests and documentation.

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.**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', (1, 23011544))
('chr2LHet', (1, 368872))
('chr2R', (1, 21146708))
('chr2RHet', (1, 3288761))
('chr3L', (1, 24543557))
('chr3LHet', (1, 2555491))
('chr3R', (1, 27905053))
('chr3RHet', (1, 2517507))
('chr4', (1, 1351857))
('chrM', (1, 19517))
('chrU', (1, 10049037))
('chrUextra', (1, 29004656))
('chrX', (1, 22422827))
('chrXHet', (1, 204112))
('chrYHet', (1, 347038))
```

pybedtools.**list_example_files**()
Returns a list of files in the examples dir. Choose one and pass it to `example_file_fnl()` 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 `BedTool` methods that wrap BEDTools programs

The following methods wrap BEDTools programs. This package is still in development; the goal is to eventually support all BEDTools programs.

`BedTool.intersect`

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 – that is, 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.

BedTool.merge

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),

BedTool.subtract

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.

BedTool.sequence

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 (v2.11.2) 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.

BedTool.closest

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

BedTool.window

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."

BedTool.sort

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).

BedTool.slop

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.

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

BedTool.shuffle

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 (v2.11.2) 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

2.5.3 BedTool methods unique to pybedtools

The following methods are currently only supported for use with BED format files; support for other file types is under development.

BedTool.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()
```

BedTool.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=128,255,0")
>>> open(b.fn).readline()
"name='test run' color=128,255,0\n"
```

Note: This method returns a new bedtool instance

BedTool.features

BedTool.**features**()

Returns an iterator of `feature` objects.

BedTool.print_sequence

BedTool.**print_sequence**()

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

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

BedTool.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')
```

BedTool.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

---

BedTool.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
```

BedTool.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)

CHAPTER

THREE

INDICES AND TABLES

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

PYTHON MODULE INDEX

p

INDEX