pybedtools Documentation

Release 0.5rc1

Ryan Dale

July 08, 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://packages.python.org/pybedtools/.

Created by Ryan Dale 2010

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

The documentation is separated into 4 main parts, depending on the depth you'd like to cover:

- Lazy, or just want to jump in? Check out *Three brief examples* to get a feel for the package.
- Want a guided tour? Give the *Tutorial Contents* a shot.
- More advanced features are described in the *Topical Documentation* section.
- Finally, doctested module documentation can be found in *pybedtools Reference*.

CHAPTER

TWO

CONTENTS:

2.1 Installation

2.1.1 Requirements

First, make sure you have the following requirements installed before attempting to install `pybedtools`:

1. BEDTools. The version is not important, but later versions will have more features so it's a good idea to get the latest. Follow the instructions at https://github.com/arq5x/bedtools to install, and make sure the programs are on your path. That is, you should be able to call `intersectBed` from any directory
2. Python 2.5 or greater (Python 3 support is coming soon)
3. **A C/C++ compiler**
    - Windows: http://www.mingw.org/
    - OSX: Install Xcode from http://developer.apple.com/xcode/
    - Linux: usually already installed; on Ubuntu, install with `sudo apt-get install build-essentials`

2.1.2 Installing `pybedtools`

Simple installation

For most users, the latest stable version will be most appropriate. To install, use pip or easy_install to automatically download the code from the Python Package Index:

```
pip pybedtools
```

or:

```
easy_install pybedtools
```

You may need to be root in order to install. If you do not have root privleges (for example, installing in your user directory on a cluster), then use the `--prefix` argument to `easy_install` to specify a location where you have write permission:

```
easy_install pybedtools --prefix /dir_you_can_write_to
```

Done! You can now run a quick test of your installation:

Quick test

A quick functional test is to create a new script with the following contents:

```
import pybedtools
a = pybedtools.example_bedtool('a.bed')
b = pybedtools.example_bedtool('b.bed')
print a.intersect(b)
```

If this script is called `test.py`, then running it with `python test.py` should print out:

```
chr1        155     200     feature2        0       +
chr1        155     200     feature3        0       -
chr1        900     901     feature4        0       +
```

Installation for developers or for running tests

A more flexible way to install `pybedtools` is as follows:

1. Get the source. There are two ways to do this:
   (a) **Stable version:** download and unzip the `pybedtools` source from http://pypi.python.org/pypi/pybedtools
   (b) **Development version:** clone the git repository at http://github.com/daler/pybedtools
2. Move to the newly created directory
3. *[optional]* Re-compile the extensions by running Cython. Typically, only developers making changes to the `.pyx`, `.h`, or `.cpp` code will need to do this. Cython needs to be installed for this step.

   ```
   python build.py
   ```

4. *[optional]* Run the tests. nosetests and PyYAML will be installed if they are not already available:

   ```
   python setup.py nosetests
   ```

5. **Install** `pybedtools`:

   ```
   python setup.py install
   ```

6. *[optional]* Install sphinx if needed (`easy_install sphinx`), then run the Sphinx doctests:

   ```
   cd docs && make doctests
   ```

7. *[optional]* Compile the HTML documentation (also needs sphinx), then point your browser to `docs/build/html.index.html`:

   ```
   cd docs && make html
   ```

8. *[optional]* Compile the PDF documentation (needs `sphinx_`), then view `manual.pdf`:

   ```
   cd docs && make latexpdf
   ```

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*.

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:

```
>>> import pybedtools
>>> a = pybedtools.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:

```
>>> import pybedtools

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

For more, see the `pybedtools.scripts.venn_mpl` and `pybedtools.scripts.venn_gchart` scripts, which wrap this functionality in command-line scripts to create Venn diagrams using either matplotlib or Google Charts API respectively.

2.2.3 Example 3: Count reads in introns and exons, in parallel

This example shows how to count the number of reads in introns and exons in parallel. It is somewhat more involved, but illustrates several additional features of `pybedtools` such as:

- BAM file support (for more, see *Working with BAM files*)
- indexing into Interval objects (for more, see *Intervals*)
- filtering (for more, see *Filtering*)
- streaming (for more, see *Using BedTool objects as iterators/generators*)
- ability to use parallel processing

The first listing has many explanatory comments, and the second listing shows the same code with no comments to give more of a feel for `pybedtools`.

```
import sys
import multiprocessing
import pybedtools

# get example GFF and BAM filenames
gff = pybedtools.example_filename('gdc.gff')
bam = pybedtools.example_filename('gdc.bam')

# Some GFF files have invalid entries -- like chromosomes with negative coords
# or features of length = 0.  This line removes them and saves the result in a
# tempfile
```

```
g = pybedtools.BedTool(gff).remove_invalid().saveas()


# Next, we create a function to pass only features for a particular
# featuretype.  This is similar to a "grep" operation when applied to every
# feature in a BedTool
def featuretype_filter(feature, featuretype):
    if feature[2] == featuretype:
        return True
    return False


# This function will eventually be run in parallel, applying the filter above
# to several different BedTools simultaneously
def subset_featuretypes(featuretype):
    return g.filter(featuretype_filter, featuretype).saveas()


# This function performs the intersection of a BAM file with a GFF file and
# returns the total number of hits.  It will eventually be run in parallel.
def count_reads_in_features(features):
    """
    Callback function to count reads in features
    """
    # This shows how to use BAM files by using the `abam` kwarg and explicitly
    # specifying the `b` kwarg as well.

    # In addition, we use stream=True so that no intermediate tempfile is
    # created, and bed=True so that the .count() method can iterate through the
    # resulting streamed BedTool.
    return features.intersect(abam=bam,
                              b=features.fn,
                              bed=True,
                              stream=True).count()


# Set up a pool of workers for parallel processing
pool = multiprocessing.Pool()

# Create separate files for introns and exons, using the function we defined
# above
featuretypes = ('intron', 'exon')
introns, exons = pool.map(subset_featuretypes, featuretypes)

# Perform some genome algebra to get unique and shared intron/exon regions
exon_only = exons.subtract(introns).merge().remove_invalid().saveas()
intron_only = introns.subtract(exons).merge().remove_invalid().saveas()
intron_and_exon = exons.intersect(introns).merge().remove_invalid().saveas()

# Do intersections with BAM file in parallel, using the other function we
# defined above
features = (exon_only, intron_only, intron_and_exon)
results = pool.map(count_reads_in_features, features)

# Print the results
labels = ('      exon only:',
          '    intron only:',
          'intron and exon:')
```

```
for label, reads in zip(labels, results):
    sys.stdout.write('%s %s\n' % (label, reads))

# Clean up any tempfiles that were created
pybedtools.cleanup(verbose=False)
```

Here's the same code but with no comments:

```
import sys
import multiprocessing
import pybedtools

gff = pybedtools.example_filename('gdc.gff')
bam = pybedtools.example_filename('gdc.bam')

g = pybedtools.BedTool(gff).remove_invalid().saveas()


def featuretype_filter(feature, featuretype):
    if feature[2] == featuretype:
        return True
    return False


def subset_featuretypes(featuretype):
    return g.filter(featuretype_filter, featuretype).saveas()


def count_reads_in_features(features):
    """
    Callback function to count reads in features
    """
    return features.intersect(abam=bam,
                              b=features.fn,
                              bed=True,
                              stream=True).count()

pool = multiprocessing.Pool()

featuretypes = ('intron', 'exon')
introns, exons = pool.map(subset_featuretypes, featuretypes)

exon_only = exons.subtract(introns).merge().remove_invalid().saveas()
intron_only = introns.subtract(exons).merge().remove_invalid().saveas()
intron_and_exon = exons.intersect(introns).merge().remove_invalid().saveas()

features = (exon_only, intron_only, intron_and_exon)
results = pool.map(count_reads_in_features, features)
labels = ('      exon only:',
          '    intron only:',
          'intron and exon:')

for label, reads in zip(labels, results):
    sys.stdout.write('%s %s\n' % (label, reads))

pybedtools.cleanup(verbose=False)
```

For more on using `pybedtools`, continue on to the *Tutorial Contents* . . .

2.3 Tutorial Contents

2.3.1 Intro

This tutorial assumes that

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

A brief note on conventions

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

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

Onward!

2.3.2 Create a `BedTool`

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

Then import the `pybedtools` module and make a new `BedTool`:

```
>>> import pybedtools

>>> # use a BED file that ships with pybedtools...
>>> a = pybedtools.example_bedtool('a.bed')

>>> # ...or use your own by passing a filename
>>> a = pybedtools.BedTool('peaks.bed')
```

This documentation uses example files that ship with `pybedtools`. To access these files from their installation location, we use the `example_bedtool()` function. This is convenient because if you copy-paste the examples, they will work. If you would rather learn using your own files, just pass the filename to a new `BedTool`, like the above example.

You can use any file that BEDTools supports – this includes BED, VCF, GFF, and gzipped versions of any of these. See *Creating a BedTool* for more on the different ways of creating a `BedTool`, including from iterators and directly from a string.

Now, let's see how to do a common task performed on BED files: intersections.

2.3.3 Intersections

One common use of BEDTools and `pybedtools` is to perform intersections.

First, let's create some example `BedTool` instances:

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

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

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

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

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

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

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

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

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

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

This time, `a_with_b` is another `BedTool` object that points to a different temp file whose name is stored in `a_with_b.fn`. You can read more about the use of temp files in *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:

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


>>> # opening the underlying file shows the track line
>>> print open(c.fn).read()
track name="a and b"
chr1        155     200     feature2        0       +
chr1        155     200     feature3        0       -
chr1        900     901     feature4        0       +
```

```
>>> # printing the BedTool object will not show non-feature lines
>>> print c
chr1        155     200     feature2        0       +
chr1        155     200     feature3        0       -
chr1        900     901     feature4        0       +
```

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

2.3.5 Default arguments

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

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

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

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

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

But since we're calling a method on the `BedTool` object `a`, `pybedtools` assumes that the file `a` points to (stored in the attribute `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()`, and others. For these methods, in addition to assuming `-a` is taken care of by the `BedTool.fn` attribute, `pybedtools` also 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`).

An example may help to illustrate: these different ways of calling `BedTool.intersect()` all have the same results, with the first version being the most compact (and probably most convenient):

```
>>> # these all have identical results
>>> x1 = a.intersect(b)
>>> x2 = a.intersect(a=a.fn, b=b.fn)
>>> x3 = a.intersect(b=b.fn)
>>> x4 = a.intersect(b, a=a.fn)
>>> x1 == x2 == x3 == x4
True
```

Note that `a.intersect(a=a.fn, b)` is not a valid Python expression, since non-keyword arguments must come before keyword arguments, but `a.intersect(b, a=a.fn)` works fine.

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

```
.. note::

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

OK, enough about arguments for now, but you can read more about them in *Principle 2: Names and arguments are as similar as possible to BEDTools*, *Principle 4: Sensible default args* and *Principal 5: 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.

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

```
>>> # These two lines...
>>> x1 = a.intersect(b, u=True)
>>> x2 = x1.merge()

>>> # ...can be combined into one line:
>>> x3 = a.intersect(b, u=True).merge()

>>> x2 == x3
True
```

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

```
.. note::

    This method returns a new BedTool instance
```

A rule of thumb is that all methods that wrap BEDTools programs return `BedTool` objects, so you can chain these together. Many `pybedtools`-unique methods return `BedTool` objects too, just check the docs (according to *Principle 7: 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. For example:

```
>>> x4 = a.intersect(b, u=True).saveas('a-with-b.bed').merge().saveas('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`. Since `BedTool.saveas()` returns a `BedTool` object, `x4` points to the `a-with-b-merged.bed` file.

2.3.7 Operator overloading

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

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

```
>>> x5 = a.intersect(b, u=True)
>>> x6 = a + b

>>> x5 == x6
True
```

Just as the + operator assumes `intersectBed` with the `-u` arg, the - operator assumes `intersectBed` with the `-v` arg:

```
>>> x7 = a.intersect(b, v=True)
>>> x8 = a - b

>>> x7 == x8
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. This is another way to do the chaining together of the intersection and merge example from above:

```
>>> x9 = (a + b).merge()
```

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

```
>>> x2 == x3 == x4 == x9
True
```

You can learn more about chaining in *Principle 6: Chaining together commands*.

2.3.8 Intervals

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

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

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

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

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

Common `Interval` attributes

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

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

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

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

>>> feature.start
1L

>>> feature.stop
100L

>>> feature.name
'feature1'
```

```
>>> feature.score
'0'

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

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

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

>>> print feature2
chrX        500     1000


>>> feature2.chrom
'chrX'

>>> feature2.start
500L

>>> feature2.stop
1000L

>>> feature2.name
''

>>> feature2.score
''

>>> feature2.strand
''
```

This illustrates that default values are empty strings.

Indexing into `Interval` objects

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

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


>>> feature[0]
'chr1'

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

>>> feature[1]
'1'

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

Fields

Interval objects have a Interval.fields attribute that contains the original line split into a list of strings. When an integer index is used on the Interval (for example, feature[3]), it is the fields attribute that is actually being indexed into.

```
>>> f = iter(pybedtools.BedTool('chr1 1 100 asdf 0 + a b c d', from_string=True)).next()
>>> f.fields
['chr1', '1', '100', 'asdf', '0', '+', 'a', 'b', 'c', 'd']
>>> len(f.fields)
10
```

BED is 0-based, others are 1-based

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

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

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

Note: pybedtools follows the following conventions:

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

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

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

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


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

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

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

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

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

GFF features have access to attributes

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

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

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

>>> # add some new attributes
>>> gff.attrs['Awesomeness'] = 99
>>> gff.attrs['ID'] = 'transcript1'

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

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

2.3.9 Filtering

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

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

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

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

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

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

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


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


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

See *Using BedTool objects as iterators/generators* for more advanced and space-efficient usage of `filter()` using iterators.

Fast filtering functions in Cython

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

```
>>> from pybedtools.featurefuncs import greater_than

>>> 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
```

2.3.10 Each

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

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

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

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

Let's define a function that will take the number of counts in each feature as calculated above and divide by the number of bases in that feature. We can also supply an optional scalar, like 0.001, to get the results in "number of intersections per kb". We then insert that value into the score field of the feature. Here's the function:

```
>>> def normalize_count(feature, scalar=0.001):
...     """
...     assume feature's last field is the count
...     """
...     counts = float(feature[-1])
...     normalized = counts / len(feature) * scalar
...
...     # need to convert back to string to insert into feature
...     feature.score = str(normalized)
...     return feature
```

And we apply it like this:

```
>>> normalized = with_counts.each(normalize_count)
>>> print normalized
chr1        1       100     feature1        0.0     +       0
chr1        100     200     feature2        1e-05   +       1
chr1        150     500     feature3        2.85714285714e-06       -       1
chr1        900     950     feature4        2e-05   +       1
```

2.3.11 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/pybedtoo
```

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("/usr/local/lib/python2.6/dist-packages/pybedtools/test/data/a.be
|    |    |    |                        "/usr/local/lib/python2.6/dist-packages/pybedtools/test/data/b.be
|    |    |    |                        ),
|    |    |    |                        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(
|                       "/usr/local/lib/python2.6/dist-packages/pybedtools/test/data/b.bed",
|                       ),
|                       parent tag: zlwqblvk,
|                       result tag: reztxhen
]
```

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

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

2.3.12 Deleting temp files specific to a single `BedTool`

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

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

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

2.4 Topical Documentation

This section contains additional documentation not covered in the tutorial.

2.4.1 Design principles

Hopefully, understanding (or just being aware of) these design principles will help in getting the most out of `pybedtools` and working efficiently.

Principle 1: Temporary files are created automatically

Using `BedTool` instances typically has the side effect of creating temporary files on disk. Even when using the iterator protocol of `BedTool` objects, temporary files may be created in order to run BEDTools programs (see *Using BedTool objects as iterators/generators* for more on this latter topic).

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`.

When you are done using the `pybedtools` module, make sure to clean up all the temp files created with:

```
>>> # 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. Alternatively, in this session or another session you can use:

```
>>> pybedtools.cleanup(remove_all=True)
```

to remove all files that match the pattern `<tempdir>/pybedtools.*.tmp` where `<tempdir>` is the current value of `pybedtools.get_tempdir()`.

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. All temp files will then be written to that directory, until the tempdir is changed again.

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

As much as possible, BEDTools programs and `BedTool` methods share the same names and arguments.

Returning again to this example:

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

This demonstrates 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 *Principle 7: 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.

Principle 3: Indifference to BEDTools version

Since `BedTool` 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)`.

`pybedtools` will also raise an exception if you try to use a method that relies on a more recent version of BEDTools than you have installed.

Principle 4: 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 the `BedTool`, `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
>>> result1 == 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.example_bedtool('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)
>>> result3 == 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 5: 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 4: 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 6: 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 7: 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. The documentation is also run through doctests, so the code you read here is guaranteed to work and be up-to-date.

2.4.2 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
>>> files = pybedtools.list_example_files()
```

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')
```

2.4.3 Using BedTool objects as iterators/generators

Typically, `BedTool` objects are used somewhat like handles to individual files on disk that contain BED lines. To save disk space, `BedTool` objects also have the ability to "stream", much like piping in Unix. That is, the data are created only one line at a time in memory, instead of either creating a list of all data in memory or writing all data to disk.

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

To get a streaming BedTool, use the `stream=True` kwarg. This `BedTool` will act a little differently from a standard, file-based `BedTool`.

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

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

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

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

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

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

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

Creating a `BedTool` from an iterable

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

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


>>> b = pybedtools.BedTool(f for f in a if f.start > 200)
>>> # this is the same as using filter:
>>> c = a.filter(lambda x: x.start > 200)
```

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

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

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

Indexing a `BedTool`

In some cases it may be useful to index into a `BedTool` object. We can use standard list slice syntax, and get an iterable of `Interval` objects as a result. This iterable can in turn be used to create a new `BedTool` instance:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> a[2:4]
<itertools.islice object at 0x...>

>>> for i in a[2:4]:
...     print i
chr1        150     500     feature3        0       -
chr1        900     950     feature4        0       +


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

>>> print pybedtools.BedTool(a[:3]).intersect(b)
chr1        155     200     feature2        0       +
chr1        155     200     feature3        0       -
```

2.4.4 Low-level operations

We can use the `BedTool.as_intervalfile()` method to return an `IntervalFile` instance. This class provides low-level support to the BEDTools C++ API.

The method `IntervalFile.all_hits()` takes a single `Interval` as the query and returns a list of all features in the `IntervalFile` that intersect:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> ivf = a.as_intervalfile()
>>> query = a[2]
>>> ivf.all_hits(query)
[Interval(chr1:100-200), Interval(chr1:150-500)]
```

Similarly, we can just return if there were *any* hits, a much faster operation:

```
>>> ivf.any_hits(query)
1
```

Or count how many hits:

```
>>> ivf.count_hits(query)
2
```

See the docstrings for `IntervalFile.all_hits()`, `IntervalFile.any_hits()`, and `IntervalFile.count_hits()` for more, including stranded hits and restricting hits to a specific overlap.

2.4.5 Working with BAM files

Some BEDTools programs, like `intersecteBed`, support BAM files as input. From the command line, you would need to specify the `-abam` argument to do so. However, `pybedtools` auto-detects BAM files and passes the `abam` argument automatically for you. That means if you create a `BedTool` out of a BAM file, like this:

```
x = pybedtools.example_bedtool('gdc.bam')
```

you can intersect it with a BED file without doing anything special:

```
b = pybedtools.example_bedtool('gdc.gff')
y = x.intersect(b)
```

The output of this operation follows the semantics of BEDTools. That is, for programs like `intersectBed`, if `abam` is used then the output will be BAM format as well. But if the `-bed` argument is passed, then the output will be BED format. Similarly, in `pybedtools`, if a BAM file is used to create the `BedTool` then the results will also be in BAM format. If the `bed=True` kwarg is passed, then the results be in BED format.

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

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

The first call below will return BAM results, and the second will return BED results.

```
>>> bam_results = a.intersect(b)
>>> bed_results = a.intersect(b, bed=True)
```

We can iterate over BAM files to get `Interval` objects just like iterating over BED or GFF files. Indexing works, too:

```
>>> for i in bam_results[:2]:
...     print i
HWUSI-NAME:2:69:512:1017#0  16      chr2L   9330    3       36M     *       0       0       TACAAATC
HWUSI-NAME:2:91:1201:1113#0 16      chr2L   10213   255     36M     *       0       0       TGTAGAAT

>>> bam_results[0]
Interval(chr2L:9329-9365)

>>> bam_results[:10]
<itertools.islice object at ...>

>>> cigar_string = i[5]
```

There are several things to watch out for here.

First, note that `pybedtools` uses the convention that BAM features in plain text format are considered SAM features, so these SAM features are **one-based and include the stop coordinate** as illustrated below:

```
>>> bam_results[0].start
9329L
```

```
>>> bam_results[0][3]
'9330'
```

Second, the stop coordinate is defined as the *start coord plus the length of the sequence*; eventually a more sophisticated, CIGAR-aware approach may be used. Similarly, the length is defined to be `stop - start` – again, not CIGAR-aware at the moment. For more sophisticated low-level manipulation of BAM features, you might want to consider using HTSeq.

Third, while we can iterate over a BAM file and manipulate the features as shown above, *calling BEDTools programs on a BAM-based generator is not well-supported.*

Specifically:

```
>>> a = pybedtools.example_bed('gdc.bam')
>>> b = pybedtools.example_bed('b.bed')

>>> # works, gets BAM results
>>> results = a.intersect(b)

>>> # make a generator of features in 'a'
>>> a2 = pybedtools.BedTool(i for i in a)

>>> # this does NOT work
>>> a2.intersect(b)
```

When we specified the `bed=True` kwarg above, the intersected BAM results are converted to BED format. We can use those like a normal BED file. Note that since we are viewing BED output, *the start and stops are 0-based*:

```
>>> d = a.intersect(b, bed=True)
>>> d.head(3)
chr2L       9329    9365    HWUSI-NAME:2:69:512:1017#0      3       -
chr2L       9329    9365    HWUSI-NAME:2:69:512:1017#0      3       -
chr2L       9329    9365    HWUSI-NAME:2:69:512:1017#0      3       -
```

Consistent with BEDTools programs, BAM files are **not** supported as the second input argument. In other words, `intersectBed` does not have both `-abam` and `-bbam` arguments, so `pybedtools` will not not allow this either.

However, `pybedtools` does allow streaming BAM files to be the input of methods that allow BAM input as the first input. In this [trivial] example, we can stream the first intersection to save disk space, and then send that streaming BAM to the next `BedTool.intersect()` call. Since it's not streamed, the second intersection will be saved as a temp BAM file on disk:

```
>>> a.intersect(b, stream=True).intersect(b)
```

2.4.6 Notes on BAM file semantics

These are some implementation notes about how BAM files are handled by mod:`pybedtools` for those interested in the implementation.

The initial creation of a `BedTool` that points to a file will trigger a check on the first 15 bytes of a file to see if it's a BAM file. If so, then the BedTool's `_isbam` attribute is set to `True`. If the `BedTool` is a stream, then the check will not be made, and it is up to the creator (whether it's the user on the command line or a method or function) to set the BAM-streaming BedTool's `._isbam` attribute to `True`. This is handled automatically for wrapped BEDTools programs (described below).

Some BEDTools programs natively handle BAM files. The `@_wraps` decorator that is used to wrap each method has a `bam` kwarg that specifies what input argument the wrapped tool will accept as BAM (for example, the wrapper for `intersectBed` has the kwarg `bam="abam"`).

If `self._isbam == True`, then `self.fn` is passed to the `bam` input arg instead of the default implicit input arg (so `intersectBed`, `self.fn` is passed as `abam` instead of `-a`).

Trying to call a method that does not have a `bam` kwarg registered will result in a ValueError, along with a message that says to use `BedTool.bam_to_bed()` first. For example, `subtractBed` currently doesn't accept BAM files as input, so this doesn't work:

```
>>> a = pybedtools.example_bedtool('gdc.bam')
>>> b = pybedtools.example_bedtool('gdc.gff')

>>> # doesn't work:
>>> c = a.subtract(b)
```

However, converting to `a` to BED format first (and setting `stream=True` to save on disk I/O) works fine:

```
>>> # works:
>>> c = a.bam_to_bed(stream=True).subtract(b)
```

Iterating over a file-based BedTool that points to a BAM will call `samtools view` and yields lines which sent to `IntervalIterator`, which splits the lines and passes them to `create_interval_from_list` which in turn decides on the fly whether it's gff, bed, or sam.

However, we can't easily check the first 15 bytes of a streaming BedTool, because that would consume those bytes. The `@_wraps` decorator needs to know some information about which arguments to a wrapped program result in BAM output and which result in non-BAM output.

Given `a = BedTool('x.bam')`:

- `c = a.intersect(b)` creates BAM output, so it returns a new BedTool with `c._isbam = True`.
- `a.intersect(b, bed=True)` returns BED output. `@_wraps` needs to know, if the input was BAM, which kwarg[s] disable BAM output. For example, if `-bed` is passed to `intersectBed`, the output will NOT be BAM. This is implemented with the `nonbam` kwarg for `_wraps()`. In this case, the resulting BED file is treated like any other BED file.
- `c = a.intersect(b, stream=True)` returns streaming BAM output. In this case, iterating over `c` will send the BAM stream to stdin of a samtools call

2.4.7 Specifying genomes

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

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

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

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

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

Below are examples of each.

As a file

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

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

As a string

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

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

As a dictionary

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

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

Make sure that all these different methods return the same results

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

Converting to a file

Since BEDTools programs operate on files, the fastest choice will be to use an existing file. While the time to convert a dictionary to a file is extremely small, over 1000's of files (e.g., for Monte Carlo simulations), the time may add up. The function `pybedtools.chromsizes_to_file()` will create a file from a dictionary or string:

```
>>> # with no filename specified, a tempfile will be created
>>> pybedtools.chromsizes_to_file(pybedtools.chromsizes('dm3'), 'dm3.genome')
'dm3.genome'
>>> print open('dm3.genome').read()
chr2L       23011544
chr2LHet    368872
chr2R       21146708
chr2RHet    3288761
chr3L       24543557
chr3LHet    2555491
chr3R       27905053
chr3RHet    2517507
chr4        1351857
chrM        19517
chrU        10049037
chrUextra   29004656
chrX        22422827
chrXHet     204112
chrYHet     347038
```

2.4.8 Randomization

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

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

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

Example workflow

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

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

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

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

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

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

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

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

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

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

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

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

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

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

iterations the number of iterations we specified

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

file_a the filename of `a`

file_b the filename of `b`

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

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

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

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

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

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

median randomized the median of the distribution of randomized intersections

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

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

upper 97.5th the 97.5th percentile of the randomized distribution

lower 2.5th the 2.5th percentile of the randomized distribution

For example:

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

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

2.4.9 Wrapping new tools

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

Example program description

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

```
newProgramBed -a <BED/VCF/GFF> -b <BED/VCF/GFF> [options]
```

Method name

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

Define a method in `BedTool`

Define a method in `BedTool`... and *don't add any content to the function body*. This is because the decorator we're about to add will replace the method wholesale; anything that's in the function body will effectively be ignored.

```
def new_program(self):
    pass
```

Add the `_wraps()` decorator

This is where most of the work happens.

Since most of the work of wrapping BEDTools programs needs to happen every time a new program is wrapped, this work is abstracted out into the `_wraps()` function. The `_wraps()` docstring and source is the best place to learn the details on what it's doing; here we'll focus on using it.

Our hypothetical program, `newProgramBed`, takes `-a` as the first input. We'd like to have `-a` implicitly be passed as whatever our `BedTool` already points to, so we use the `implicit='a'` kwarg to `_wraps()` here. `newProgramBed` also takes a second input, `-b`. We describe that to the wrapper with the `other='b'` kwarg.

Any other keyword args that are used when calling the method will automatically be passed to the program. So if `newProgramBed` has an optional `-s` argument, we don't need to specify that here. When the user passes an `s=True` kwarg, it will be passed automatically to `newProgramBed` as the argument `-s`. If `newProgramBed` does not accept a `-z` argument but the user passes one anyway, we rely on the BEDTools program to do the error-checking of arguments and report any errors back to Python.

Here's what the new method looks like so far:

```
@_wraps(prog='newProgramBed', implicit='a', other='b')
def new_program(self):
    pass
```

For wrapped programs that expect a genome file or have more complex arguments, see the docstring and source for `_wrap()`.

Add doctests

While the function body will be replaced wholesale by the decorator, the docstring will be copied to the new function. This is important because it means we can write meaningful documentation and, even more importantly, doctests for this method. Writing a doctest within the method's docstring means it will automatically be found by the test suite.

```
@_wraps(prog='newProgramBed', implicit='a', other='b')
def new_program(self):
    """
    Converts all features to length of 1.

    Example usage:

    >>> a = pybedtools.example_bedtool('a.bed')
    >>> b = pybedtools.example_bedtool('b.bed')
    >>> c = a.new_program(b, s=True)
    >>> print c  #+NORMALIZE_WHITESPACE
    chr1    1       2
    chr1    100     101
    chr1    150     151
    chr1    900     901
    <BLANKLINE>
    """
```

Summary

That's it! We now have a method, `BedTool.new_program()`, that wraps a hypothetical `newProgramBed` BEDTools program, will accept any optional args that `newProgramBed` does, will return a new `BedTool` containing the results, *and it's tested.*

This new method can be be chained with other `BedTool` instances, used as an iterator or generator, or anything else a normal `BedTool` can do . . . for example:

```
a = pybedtools.example_bed('a.bed')
b = pybedtools.example_bed('b.bed')
c = a.new_program(b, s=True).filter(lambda x: x.start < 125).saveas('t.bed', trackline='track name="
```

2.5 Scripts

`pybedtools` comes with several scripts that illustrate common use-cases.

In Python 2.7, you can use:

```
python -m pybedtools
```

to get a list of scripts and their description.

2.5.1 Venn diagram scripts

There are two scripts for making Venn diagrams, depending on how you'd like the diagrams to look. Both simply take 3 BED files as input. `venn_gchart.py` uses the Google Chart API, while `venn_mpl.py` uses matplotlib if you have it installed.

Upon installing `pybedtools`, these scripts should be available on your path. Calling them with the `-h` option will print the help, and using the `--test` option will run a test, creating a new file `out.png` in the current working directory.

2.5.2 Intron/exon classification

The script `intron_exon_reads.py` accepts a GFF file (with introns and exons annotated) and a BAM file. When complete, it prints out the number of exonic, intronic, and both intronic and exonic (i.e., from overlapping genes or isoforms). This script is also a good example of how to do use Python's `multiprocessing` for parallel computation.

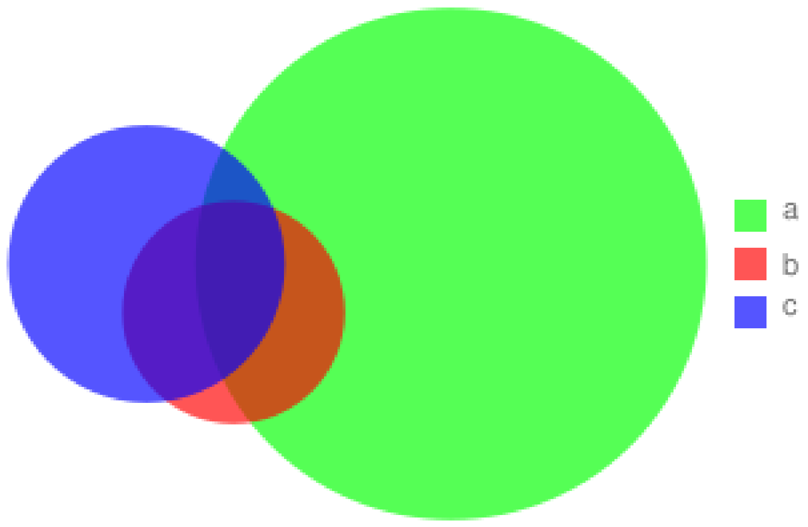

Figure 2.1: Above: using `--test` with `venn_gchart.py` results in this figure

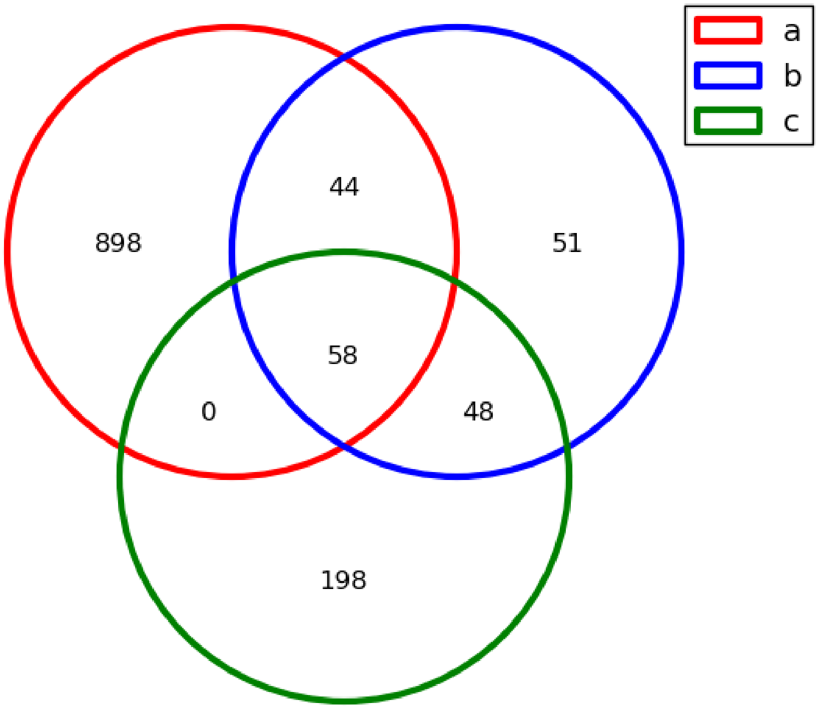

Figure 2.2: Above: Result of using `--test` with `venn_mpl.py`

2.5.3 Annotate.py

The `annotate.py` script extends `closestBed` by classifying features (intron, exon) that are a distance of 0 away from the query features.

Contents

2.6 `pybedtools` Reference

This section is the module reference documentation, and includes the full docstrings for methods and functions in `pybedtools`. It is separated into *pybedtools module-level functions*, *BedTool methods that wrap BEDTools programs*, and *BedTool methods unique to pybedtools*.

class `pybedtools.`**`BedTool`**(*fn*, *from_string=False*)

`__init__`(*fn*, *from_string=False*)

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

fn is typically the name of a BED-like file, but can also be one of the following:

- a string filename
- another BedTool object
- an iterable of Interval objects
- an open file object
- a "file contents" string (see below)

If *from_string* is True, then you can pass a string that contains the contents of the BedTool you want to create. This will treat all spaces as TABs and write to tempfile, treating whatever you pass as *fn* as the contents of the bed file. This also strips empty lines.

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_files = pybedtools.list_example_files()
>>> assert example_files[0] == 'a.bed'
>>> a = pybedtools.example_bedtool('a.bed')
```

2.6.1 `pybedtools` module-level functions

Functions for working with example files

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

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

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

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

`pybedtools.`**`list_example_files`**`()`

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

Example usage:

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

`pybedtools.`**`data_dir`**`()`

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

Functions for specifying genome assemblies

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

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

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

Example usage:

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

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

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

Returns the filename.

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

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

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

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

timeout is how long to wait for a response; mostly used for testing.

Example usage:

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

Setup

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

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

`pybedtools.`**`get_tempdir`**()

Gets the current tempdir for the module.

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

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

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

Utilities

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

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

If *verbose*, reports what it's doing

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

`pybedtools.`**`IntervalIterator`**()

`pybedtools.`**`find_tagged`**(*tag*)

Returns the bedtool object with tagged with *tag*. Useful for tracking down bedtools you made previously.

Wrapping

`pybedtools.bedtool.`**`_wraps`**(*prog=None*, *implicit=None*, *bam=None*, *other=None*, *uses_genome=False*, *make_tempfile_for=None*, *check_stderr=None*, *add_to_bedtool=None*, *nonbam=None*, *force_bam=False*)

Do-it-all wrapper, to be used as a decorator.

prog is the name of the BEDTools program that will be called. The help for this program will also be added to the decorated method's docstring.

implicit is the BEDTools program arg that should be filled in automatically.

bam will disable the implicit substitution if *bam* is in the kwargs. This is typically 'abam' or 'ibam' if the program accepts BAM input.

other is the BEDTools program arg that is passed in as the second input, if supported. Within the semantics of BEDTools, the typical case will be that if implicit='a' then other='b'; if implicit='i' then other=None.

uses_genome, if True, will check for 'g' and/or 'genome' args and retrieve the corresponding genome files as needed.

make_tempfile_for is used for the sequence methods and indicates which kwarg should have a tempfile made for it if it's not provided ('fo' for the sequence methods)

check_stderr, if not None, is a function that accepts a string (which will be anything written to stdout when calling the wrapped program). This function should return True if the string is OK, and False if it should truly be considered an error. This is needed for wrapping fastaFromBed, which will report to stderr that it's creating an index file.

add_to_bedtool is used for sequence methods. It is a dictionary mapping kwargs to attributes to be created in the resulting BedTool. Typically it is {'fo':'seqfn'} which will add the resulting sequence name to the BedTool's .seqfn attribute. If *add_to_bedtool* is not None, then the returned BedTool will be *self* with the added attribute.

nonbam is a kwarg that even if the input file was a BAM, the output will *not* be BAM format. For example, the `-bed` arg for intersectBed will cause the output to be in BED format, not BAM. If not None, this can be a string, a list of strings, or the special string "ALL", which means that the wrapped program will never return BAM output.

force_bam, if True, will force the output to be BAM. This is used for bedToBam.

2.6.2 `BedTool` methods that wrap BEDTools programs

"Genome algebra" methods

`intersect()` (wraps "intersectBed")

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

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

Example usage:

Create new BedTool object

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

Get overlaps with `b.bed`:

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

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

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

Program: intersectBed (v2.13.0) 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.

`merge()` (wraps “mergeBed”)

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

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)
```

Program: mergeBed (v2.13.0) 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.

-scores 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),

- (INTEGER)

Notes:

1. All output, regardless of input type (e.g., GFF or VCF) will in BED format with zero-based starts

`subtract()` (wraps "subtractBed")

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

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)
```

Program: subtractBed (v2.13.0) 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.

`closest()` (wraps "closestBed")

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

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)
```

Program: closestBed (v2.13.0) 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.
-no	Ignore features in B that overlap A. That is, we want close, but not touching features only.
-t	How ties for closest feature are handled. This occurs when two features in B have exactly the same "closeness" with A. By default, all such features in B are reported. Here are all the options: - "all" Report all ties (default). - "first" Report the first tie that occurred in the B file. - "last" Report the last tie that occurred in the B file.

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

`window()` (wraps "windowBed")

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

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     feature5
chr1        1       100     feature1        0       +       chr1    800     901     feature6
chr1        100     200     feature2        0       +       chr1    155     200     feature5
chr1        100     200     feature2        0       +       chr1    800     901     feature6
chr1        150     500     feature3        0       -       chr1    155     200     feature5
chr1        150     500     feature3        0       -       chr1    800     901     feature6
chr1        900     950     feature4        0       +       chr1    155     200     feature5
chr1        900     950     feature4        0       +       chr1    800     901     feature6
<BLANKLINE>
```

Program: windowBed (v2.13.0) 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."

`sort()` (wraps "sortBed")

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

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>
```

Program: sortBed (v2.13.0) 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).

`slop()` (wraps "slopBed")

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

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

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

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

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

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

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

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

First, create the file:

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

Then use it:

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

Clean up afterwards:

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

Program: slopBed (v2.13.0) 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) 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

`complementBed()` (wraps "complementBed")

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

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> a.complement(genome='hg19').head(5)
chr1    0       1
chr1    500     900
chr1    950     249250621
chr10   0       135534747
chr11   0       135006516
```

Program: complementBed (v2.13.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Returns the base pair complement of a feature file.

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

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

`flank()` (wraps "flankBed")

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

Create flanking intervals on either side of input BED.

Example usage:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> print a.flank(genome='hg19', b=100)
chr1    0       1       feature1        0       +
chr1    100     200     feature1        0       +
chr1    0       100     feature2        0       +
chr1    200     300     feature2        0       +
chr1    50      150     feature3        0       -
chr1    500     600     feature3        0       -
chr1    800     900     feature4        0       +
chr1    950     1050    feature4        0       +
<BLANKLINE>
```

Program: flankBed (v2.13.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Creates flanking interval(s) for each BED/GFF/VCF feature.

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

Options:

-b	Create flanking intervak using -b base pairs in each direction. - (Integer) or (Float, e.g. 0.1) if used with -pct.
-l	The number of base pairs that a flank should start from orig. start coordinate. - (Integer) or (Float, e.g. 0.1) if used with -pct.

-r	The number of base pairs that a flank should end from orig. 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 start the flank 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 flank would force it above the max chrom length. (3) The genome file should tab delimited and structured as follows:

<chromName><TAB><chromSize>

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

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

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

`shuffle()` (wraps "shuffleBed")

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

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>
```

Program: shuffleBed (v2.13.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Randomly permute the locations of a feature file among a genome.

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

Options:

-excl	A BED/GFF/VCF file of coordinates in which features in -i should not be placed (e.g. gaps.bed).
-incl	Instead of randomly placing features in a genome, the -incl options defines a BED/GFF/VCF file of coordinates in which features in -i should be randomly placed (e.g. genes.bed).
-chrom	Keep features in -i on the same chromosome. - By default, the chrom and position are randomly chosen.

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

Notes:

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

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

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

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

`annotate()` (wraps "annotateBed")

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

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

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

Program: annotateBed (v2.13.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Annotates the depth & breadth of coverage of features from multiple files

on the intervals in -i.

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

Options:

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

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

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

-both	Report the counts followed by the % coverage. - Default is to report the fraction of -i covered by each file.
-s	Force strandedness. That is, only include hits in A that overlap B on the same strand. - By default, hits are included without respect to strand.

`coverage()` (wraps "coverageBed")

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

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

Program: coverageBed (v2.13.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Returns the depth and breadth of coverage of features from A

on the intervals in B.

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

Options:

-abam	The A input file is in BAM format.
-s	Force strandedness. That is, only include hits in A that overlap B on the same strand. - By default, hits are included without respect to strand.
-hist	Report a histogram of coverage for each feature in B as well as a summary histogram for **all** features in B. **Output (tab delimited) after each feature in B:** 1. depth 2. # bases at depth 3. size of B 4. % of B at depth
-d	Report the depth at each position in each B feature. Positions reported are one based. Each position and depth follow the complete B feature.

-counts Only report the count of overlaps, don't compute fraction, etc.

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

Default Output:

After each entry in B, reports:

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

genome_coverage() (wraps "genomeCoverageBed")

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

Calculates coverage at each position in the genome.

Use *bg=True* to have the resulting BedTool return valid BED-like features

Example usage:

```
>>> a = pybedtools.example_bedtool('x.bam')
>>> b = a.genome_coverage(genome='dm3', bg=True)
>>> b.head(3)
chr2L   9329    9365    1
chr2L   10212   10248   1
chr2L   10255   10291   1
```

Program: genomeCoverageBed (v2.13.0) Authors: Aaron Quinlan (aaronquinlan@gmail.com)

Assaf Gordon, CSHL

Summary: Compute the coverage of a feature file among a genome.

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

Options:

-ibam	The input file is in BAM format. Note: BAM **must** be sorted by position
-d	Report the depth at each genome position (with one-based coordinates). Default behavior is to report a histogram.
-dz	Report the depth at each genome position (with zero-based coordinates). Reports only non-zero positions. Default behavior is to report a histogram.
-bg	Report depth in BedGraph format. For details, see: genome.ucsc.edu/goldenPath/help/bedgraph.html
-bga	Report depth in BedGraph format, as above (-bg). However with this option, regions with zero coverage are also reported. This allows one to quickly extract all regions of a genome with 0 coverage by applying: "grep -w 0$" to the output.
-split	Treat "split" BAM or BED12 entries as distinct BED intervals. when computing coverage. For BAM files, this uses the CIGAR "N" and "D" operations to infer the blocks for computing coverage. For BED12 files, this uses the BlockCount, BlockStarts, and BlockEnds fields (i.e., columns 10,11,12).
-strand	Calculate coverage of intervals from a specific strand. With BED files, requires at least 6 columns (strand is column 6). - (STRING): can be + or -
-5	Calculate coverage of 5" positions (instead of entire interval).
-3	Calculate coverage of 3" positions (instead of entire interval).
-max	Combine all positions with a depth >= max into a single bin in the histogram. Irrelevant for -d and -bedGraph - (INTEGER)
-scale	Scale the coverage by a constant factor. Each coverage value is multiplied by this factor before being reported. Useful for normalizing

coverage by, e.g., reads per million (RPM). - Default is 1.0; i.e., un-scaled. - (FLOAT)

-trackline Adds a UCSC/Genome-Browser track line definition in the first line of the output. - See here for more details about track line definition:

> http://genome.ucsc.edu/goldenPath/help/bedgraph.html

- **NOTE: When adding a trackline definition, the output BedGraph can be easily** uploaded to the Genome Browser as a custom track, BUT CAN NOT be converted into a BigWig file (w/o removing the first line).

-trackopts Writes additional track line definition parameters in the first line. - Example:

> -trackopts 'name="My Track" visibility=2 color=255,30,30' Note the use of single-quotes if you have spaces in your parameters.

- (TEXT)

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

2. The input BED (-i) file must be grouped by chromosome.

   A simple "sort -k 1,1 <BED> > <BED>.sorted" will suffice.

3. The input BAM (-ibam) file must be sorted by position.

   A "samtools sort <BAM>" should suffice.

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

overlap() (wraps "overlap")

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

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> b = pybedtools.example_bedtool('b.bed')
>>> c = a.window(b, w=10).overlap(cols=[2,3,8,9])
>>> print c
chr1    100     200     feature2        0       +       chr1    155     200     feature5
chr1    150     500     feature3        0       -       chr1    155     200     feature5
chr1    900     950     feature4        0       +       chr1    800     901     feature6
<BLANKLINE>
```

Program: overlap (v2.13.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Computes the amount of overlap (positive values)

or distance (negative values) between genome features and reports the result at the end of the same line.

Usage: overlap [OPTIONS] -i <input> -cols s1,e1,s2,e2

Options:

-i	Input file. Use "stdin" for pipes.
-cols	Specify the columns (1-based) for the starts and ends of the features for which you'd like to compute the overlap/distance. The columns must be listed in the following order: start1,end1,start2,end2

Example: $ windowBed -a A.bed -b B.bed -w 10 chr1 10 20 A chr1 15 25 B chr1 10 20 C chr1 25 35 D

$ windowBed -a A.bed -b B.bed -w 10 | overlap -i stdin -cols 2,3,6,7 chr1 10 20 A chr1 15 25 B 5 chr1 10 20 C chr1 25 35 D -5

`groupby()` (wraps "groupBy")

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

```
>>> a = pybedtools.example_bedtool('gdc.gff')
>>> b = pybedtools.example_bedtool('gdc.bed')
>>> c = a.intersect(b, c=True)
>>> d = c.groupby(g=[1, 4, 5], c=10, ops=['sum'])
>>> print d
chr2L   41      70      0
chr2L   71      130     2
chr2L   131     170     4
chr2L   171     200     0
chr2L   201     220     1
chr2L   41      130     2
chr2L   171     220     1
chr2L   41      220     7
chr2L   161     230     6
chr2L   41      220     7
<BLANKLINE>
```

Program: groupBy (v1.1.0) Authors: Aaron Quinlan (aaronquinlan@gmail.com)

Assaf Gordon

Summary: Summarizes a dataset column based upon common column groupings. Akin to the SQL "group by" command.

Usage: groupBy -i [FILE] -g [groupcolumn(s)] -c [op**column(s)] -o [ops]** cat [FILE] | groupBy -g [group**column(s)] -c [op**column(s)] -o [ops]

Options:

-i	Input file. Assumes "stdin" if omitted.

-g -grp Specify the columns (1-based) for the grouping. The columns must be comma separated. - Default: 1,2,3

-c -opCols Specify the column (1-based) that should be summarized.

- Required.

-o -ops Specify the operation that should be applied to opCol.

Valid operations: sum, count, min, max, mean, median, mode, antimode, stdev, sstdev (sample standard dev.), collapse (i.e., print a comma separated list), freqdesc (i.e., print desc. list of values:freq) freqasc (i.e., print asc. list of values:freq)

- Default: sum

-full	Print all columns from input file. Default: print only grouped columns.
-inheader	Input file has a header line - the first line will be ignored.
-outheader	Print header line in the output, detailing the column names. If the input file has headers (-inheader), the output file will use the input's column names. If the input file has no headers, the output file will use "col**1", "col**2", etc. as the column names.
-header	same as '-inheader -outheader'
-ignorecase	Group values regardless of upper/lower case.

Examples: $ cat ex1.out chr1 10 20 A chr1 15 25 B.1 1000 chr1 10 20 A chr1 25 35 B.2 10000

$ groupBy -i ex1.out -g 1,2,3,4 -c 9 -o sum chr1 10 20 A 11000

$ groupBy -i ex1.out -grp 1,2,3,4 -opCols 9,9 -ops sum,max chr1 10 20 A 11000 10000

$ groupBy -i ex1.out -g 1,2,3,4 -c 8,9 -o collapse,mean chr1 10 20 A B.1,B.2, 5500

$ cat ex1.out | groupBy -g 1,2,3,4 -c 8,9 -o collapse,mean chr1 10 20 A B.1,B.2, 5500

Notes:

1. The input file/stream should be sorted/grouped by the -grp. columns
2. If -i is unspecified, input is assumed to come from stdin.

`pair_to_bed()` (wraps "pairToBed")

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

Program: pairToBed (v2.13.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Report overlaps between a BEDPE file and a BED/GFF/VCF file.

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

Options:

-abam	The A input file is in BAM format. Output will be BAM as well. - Requires BAM to be grouped or sorted by query.
-ubam	Write uncompressed BAM output. Default is to write compressed BAM.
	is to write output in BAM when using -abam.
-bedpe	When using BAM input (-abam), write output as BEDPE. The default is to write output in BAM when using -abam.

-ed	Use BAM total edit distance (NM tag) for BEDPE score. - Default for BEDPE is to use the minimum of of the two mapping qualities for the pair. • When -ed is used the total edit distance from the two mates is reported as the score.
-f	Minimum overlap required as fraction of A (e.g. 0.05). Default is 1E-9 (effectively 1bp).
-s	Enforce strandedness when finding overlaps. Default is to ignore stand. Not applicable with -type inspan or -type outspan.
-type	Approach to reporting overlaps between BEDPE and BED. **either Report overlaps if either end of A overlaps B.** • Default. neither Report A if neither end of A overlaps B. both Report overlaps if both ends of A overlap B. xor Report overlaps if one and only one end of A overlaps B. notboth Report overlaps if neither end or one and only one end of A overlap B. That is, xor + neither. **ispan Report overlaps between [end1, start2] of A and B.** • Note: If chrom1 <> chrom2, entry is ignored. **ospan Report overlaps between [start1, end2] of A and B.** • Note: If chrom1 <> chrom2, entry is ignored. **notispan Report A if ispan of A doesn't overlap B.** • Note: If chrom1 <> chrom2, entry is ignored. **notospan Report A if ospan of A doesn't overlap B.** • Note: If chrom1 <> chrom2, entry is ignored.

Refer to the BEDTools manual for BEDPE format.

`pair_to_pair()` (wraps "pairToPair")

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

Program: pairToPair (v2.13.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Report overlaps between two paired-end BED files (BEDPE).

Usage: pairToPair [OPTIONS] -a <BEDPE> -b <BEDPE>

Options:

-f	Minimum overlap required as fraction of A (e.g. 0.05). Default is 1E-9 (effectively 1bp).
-type	Approach to reporting overlaps between A and B. neither Report overlaps if neither end of A overlaps B. either Report overlaps if either ends of A overlap B. both Report overlaps if both

	ends of A overlap B. notboth Report overlaps if one or neither of ends of A overlap B. - Default = both.
-slop	The amount of slop (in b.p.). to be added to each footprint. *Note*: Slop is subtracted from start1 and start2 and added to end1 and end2.
-ss	Add slop based to each BEDPE footprint based on strand. - If strand is "+", slop is only added to the end coordinates. - If strand is "-", slop is only added to the start coordinates. - By default, slop is added in both directions.
-is	Ignore strands when searching for overlaps. - By default, strands are enforced.
-rdn	Require the hits to have different names (i.e. avoid self-hits). - By default, same names are allowed.

Refer to the BEDTools manual for BEDPE format.

Methods for converting between formats

`bed6()` (wraps "Bed12To6")

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

convert a BED12 to a BED6 file

Program: bed12ToBed6 (v2.13.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Splits BED12 features into discrete BED6 features.

Usage: bed12ToBed6 [OPTIONS] -i <bed12>

Options:

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

Methods for working with sequences

`sequence()` (wraps "fastaFromBed")

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

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>
```

Program: fastaFromBed (v2.13.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Extract DNA sequences into a fasta file based on feature coordinates.

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

Options:

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

`mask_fasta()` (wraps "maskFastaFromBed")

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

Masks a fasta file at the positions in a BED file and saves result as 'out' and stores the filename in seqfn.

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

Program: maskFastaFromBed (v2.13.0) Author: Aaron Quinlan (aaronquinlan@gmail.com) Summary: Mask a fasta file based on feature coordinates.

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

Options:

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

2.6.3 `BedTool` methods unique to `pybedtools`

Introspection

`count()`

`BedTool.`**`count`**`()`

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

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

Example usage:

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> a.count()
4
```

`print_sequence()`

`BedTool.`**`print_sequence`**`()`

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

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

`field_count()`

`BedTool.`**`field_count`**`(`*n=10*`)`

Return the number of fields in the features this file contains. Checks the first *n* features.

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> a.field_count()
6
```

`head()`

`BedTool.`**`head`**`(`*n=10*`)`

Prints the first *n* lines

```
>>> a = pybedtools.example_bedtool('a.bed')
>>> a.head(2)
chr1    1       100     feature1        0       +
chr1    100     200     feature2        0       +
<BLANKLINE>
```

Saving

`saveas()`

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

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

Returns a new BedTool for the newly saved file.

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

Example usage:

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

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

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 = 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>
>>> b = a.save_seqs('example.fa')
>>> assert open(b.fn).read() == open(a.fn).read()
```

Utilities

with_attrs()

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

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

Example usage:

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

cat()

BedTool.**cat**(**args*, ***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>
```

total_coverage()

BedTool.**total_coverage**()

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

Example usage:

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

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

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

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

delete_temporary_history()

BedTool.**delete_temporary_history**(*ask=True*, *raw_input_func=None*)

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

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

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

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

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

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

(*raw_input_func* is used for testing)

as_intervalfile()

BedTool.**as_intervalfile**()

Returns an IntervalFile of this BedTool, which provides a low-level interface

Feature-by-feature operations

each()

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

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

func must return an Interval object.

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

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

filter()

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

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

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

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

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

so it has extracted 2 records from the original 4.

cut()

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

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

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

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

features()

`BedTool.features()`
Returns an iterator of `feature` objects.

Randomization helpers

randomintersection()

`BedTool.randomintersection(`*other*, *iterations*, *intersect_kwargs=None*, *shuffle_kwargs=None*, *debug=False*`)`
Performs *iterations* shufflings of self, each time intersecting with *other*.

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

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

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

Example usage:

```
>>> chromsizes = {'chr1':(0, 1000)}
>>> a = pybedtools.example_bedtool('a.bed')
>>> a = a.set_chromsizes(chromsizes)
>>> b = pybedtools.example_bedtool('b.bed')
>>> results = a.randomintersection(b, 10, debug=True)
>>> print list(results)
[2, 2, 2, 0, 2, 3, 2, 1, 2, 3]
```

randomstats()

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

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

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

Example usage:

Make chromsizes a very small genome for this example:

```
>>> chromsizes = {'chr1':(1,1000)}
>>> a = pybedtools.example_bedtool('a.bed').set_chromsizes(chromsizes)
>>> b = pybedtools.example_bedtool('b.bed')
>>> try:
...     results = a.randomstats(b, 100, debug=True)
... except ImportError:
...     # allow doctests to pass if SciPy not installed
...     pass
```

results is a dictionary that you can inspect.

(Note that the following examples are not run as part of the doctests to avoid forcing users to install SciPy just to pass tests)

The actual overlap:

```
print results['actual']
3
```

The median of all randomized overlaps:

```
print results['median randomized']
2.0
```

The percentile of the actual overlap in the distribution of randomized overlaps, which can be used to get an empirical p-value:

```
print results['percentile']
90.0
```

`Interval` class

class `pybedtools.`**`Interval`**

Constructor:

> Interval(chrom, start, end, name=".", score=".", strand=".", otherfields=None)

Class to represent a genomic interval of any format. Requires at least 3 args: chrom (string), start (int), end (int).

`start` is *always* the 0-based start coordinate. If this Interval is to represent a GFF object (which uses a 1-based coordinate system), then subtract 1 from the 4th item in the line to get the start position in 0-based coords for this Interval. The 1-based GFF coord will still be available, albeit as a string, in fields[3].

`otherfields` is a list of fields that don't fit into the other kwargs, and will be stored in the `fields` attribute of the Interval.

All the items in `otherfields` must be strings for proper conversion to C++.

By convention, for BED files, `otherfields` is everything past the first 6 items in the line. This allows an Interval to represent composite features (e.g., a GFF line concatenated to the end of a BED line)

But for other formats (VCF, GFF, SAM), the entire line should be passed in as a list for `otherfields` so that we can always check the Interval.file_type and extract the fields we want, knowing that they'll be in the right order as passed in with `otherfields`.

Example usage:

```
>>> from pybedtools import Interval
>>> i = Interval("chr1", 22, 44, strand='-')
>>> i
Interval(chr1:22-44)
```

```
>>> i.start, i.end, i.strand, i.length
(22L, 44L, '-', 22L)
```

chrom
the chromosome of the feature

end
The end of the feature

file_type
bed/vcf/gff

length
the length of the feature

name

```
>>> import pybedtools
>>> vcf = pybedtools.example_bedtool('v.vcf')
>>> [v.name for v in vcf]
['rs6054257', 'chr1:16', 'rs6040355', 'chr1:222', 'microsat1']
```

start
The 0-based start of the feature.

stop
the end of the feature

strand
the strand of the feature

IntervalFile class

class pybedtools.**IntervalFile**

all_hits

Signature IntervalFile.all_hits(interval, same_strand=False, overlap=0.0)

Search for the Interval `interval` this file and return **all** overlaps as a list.

`same_strand`, if True, will only consider hits on the same strand as `interval`.

`overlap` can be used to specify the fraction of overlap between `interval` and each feature in the IntervalFile.

Example usage:

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

>>> # create an Interval to query with
>>> i = pybedtools.Interval('chr1', 1, 10000, strand='+')

>>> # Create an IntervalFile out of a.bed
>>> intervalfile = pybedtools.IntervalFile(fn)
>>> # get stranded hits
>>> intervalfile.all_hits(i, same_strand=True)
[Interval(chr1:1-100), Interval(chr1:100-200), Interval(chr1:900-950)]
```

any_hits

Signature `IntervalFile.any_hits(interval, same_strand=False, overlap=0.0)`

Return 1 if the Interval `interval` had >=1 hit in this IntervalFile, 0 otherwise.

`same_strand`, if True, will only consider hits on the same strand as `interval`.

`overlap` can be used to specify the fraction of overlap between `interval` and each feature in the IntervalFile.

Example usage:

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

>>> # create an Interval to query with
>>> i = pybedtools.Interval('chr1', 1, 10000, strand='+')

>>> # Create an IntervalFile out of a.bed
>>> intervalfile = pybedtools.IntervalFile(fn)

>>> # any stranded hits?
>>> intervalfile.any_hits(i, same_strand=True)
1
```

count_hits

Signature `IntervalFile.count_hits(interval, same_strand=False, overlap=0.0)`

Return the number of overlaps of the Interval `interval` had with this IntervalFile.

`same_strand`, if True, will only consider hits on the same strand as `interval`.

`overlap` can be used to specify the fraction of overlap between `interval` and each feature in the IntervalFile.

Example usage:

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

>>> # create an Interval to query with
>>> i = pybedtools.Interval('chr1', 1, 10000, strand='+')

>>> # Create an IntervalFile out of a.bed
>>> intervalfile = pybedtools.IntervalFile(fn)

>>> # get number of stranded hits
>>> intervalfile.count_hits(i, same_strand=True)
3
```

loadIntoMap

Prepares file for checking intersections. Used by other methods like all_hits()

next

x.next() -> the next value, or raise StopIteration

search

Signature `IntervalFile.all_hits(interval, same_strand=False, overlap=0.0)`

Search for the Interval `interval` this file and return **all** overlaps as a list.

`same_strand`, if True, will only consider hits on the same strand as `interval`.

`overlap` can be used to specify the fraction of overlap between `interval` and each feature in the IntervalFile.

Example usage:

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

>>> # create an Interval to query with
>>> i = pybedtools.Interval('chr1', 1, 10000, strand='+')

>>> # Create an IntervalFile out of a.bed
>>> intervalfile = pybedtools.IntervalFile(fn)

>>> # get stranded hits
>>> intervalfile.all_hits(i, same_strand=True)
[Interval(chr1:1-100), Interval(chr1:100-200), Interval(chr1:900-950)]
```

CHAPTER

THREE

INDICES AND TABLES

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

INDEX