Python

Release 0.6.4

Open2C

Apr 29, 2024

GUIDE

Bioframe is a library to enable flexible and scalable operations on genomic interval dataframes in python. Building bioframe directly on top of pandas enables immediate access to a rich set of dataframe operations. Working in python enables rapid visualization and iteration of genomic analyses.

CHAPTER

ONE

QUICKSTART

1.1 Installation

```
$ pip install bioframe
```

To install the latest development version of *bioframe* from github, first make a local clone of the github repository:

```
$ git clone https://github.com/open2c/bioframe
```

Then, compile and install *bioframe* in development mode. This installs the package without moving it to a system folder, and thus allows for testing changes to the python code on the fly.

```
$ cd bioframe
$ pip install -e ./
```

CHAPTER

TWO

GENOMIC INTERVAL OPERATIONS

This guide provides an introdution into how to use bioframe to perform genomic interval operations. For the full list of genomic interval operations, see the *API reference*.

The following modules are used in this guide:

```
import itertools

import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd

import bioframe as bf
import bioframe.vis
```

2.1 DataFrames & BedFrames

The core objects in bioframe are pandas DatFrames of genomic intervals, or BedFrames. These can either be defined directly with `pandas.DataFrame`:

```
df1 = pd.DataFrame([
    ['chr1', 1, 5],
    ['chr1', 3, 8],
    ['chr1', 8, 10],
    ['chr1', 12, 14]],
    columns=['chrom', 'start', 'end']
)
```

Or via functions in *bioframe.core.construction*, e.g.:

```
df2 = bioframe.from_any(
    [['chr1', 4, 8],
     ['chr1', 10, 11]],
    name_col='chrom')
```

Or ingested from datasets and databases with functions in *bioframe.io.fileops* and *bioframe.io.resources*.

BedFrames satisfy the following properties:

- chrom, start, end columns

- columns have valid dtypes (object/string/categorical, int/pd.Int64Dtype(), int/pd.Int64Dtype())
- for each interval, if any of chrom, start, end are null, then all are null
- all starts <= ends.

Whether a dataframe satisfies these properties can be checked with *bioframe.core.checks.is_bedframe()*:

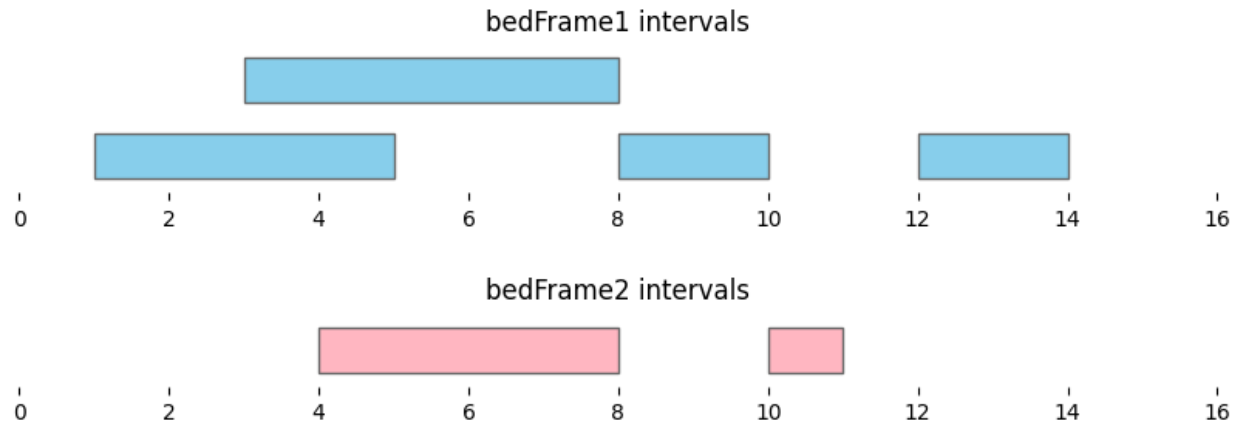
```
bioframe.is_bedframe(df2)
```

```
True
```

See *bioframe.core.checks* for other functions that test properties of BedFrames and Technical Notes for detailed definitions. *bioframe.core.construction.sanitize_bedframe()* attempts to modfiy a DataFrame such that it satisfies bedFrame requirements.

bioframe.vis provides plotting utilities for intervals:

```
bf.vis.plot_intervals(df1, show_coords=True, xlim=(0,16))
plt.title('bedFrame1 intervals');

bf.vis.plot_intervals(df2, show_coords=True, xlim=(0,16), colors='lightpink')
plt.title('bedFrame2 intervals');
```



2.2 Overlap

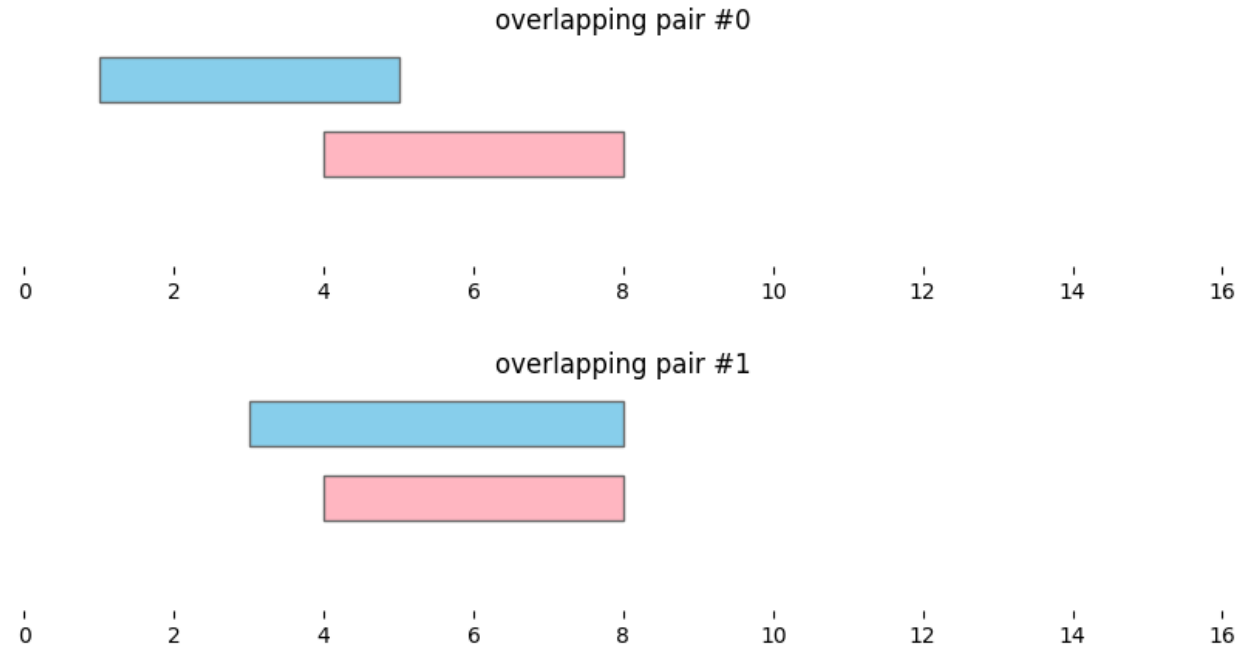
Calculating the overlap between two sets of genomic intervals is a crucial genomic interval operation.

Using `bioframe.overlap()`, we can see the two dataframes defined above, `df1` and `df2`, contain two pairs of overlapping intervals:

```
overlapping_intervals = bf.overlap(df1, df2, how='inner', suffixes=('_1','_2'))
display(overlapping_intervals)
```

	chrom_1	start_1	end_1	chrom_2	start_2	end_2
0	chr1	1	5	chr1	4	8
1	chr1	3	8	chr1	4	8

```
for i, reg_pair in overlapping_intervals.iterrows():
    bf.vis.plot_intervals_arr(
        starts = [reg_pair.start_1,reg_pair.start_2],
        ends = [reg_pair.end_1,reg_pair.end_2],
        colors = ['skyblue', 'lightpink'],
        levels = [2,1],
        xlim = (0,16),
        show_coords = True)
    plt.title(f'overlapping pair #{i}')
```



Note that we passed custom suffixes for the outputs (defaults are `suffixes=("","_")`), as well as a custom overlap mode (`how='inner'`). The default overlap mode, `how='left'` returns each interval in `df1` whether or not it overlaps an interval in `df2`.

```
overlapping_intervals = bf.overlap(df1, df2)
display(overlapping_intervals)
```

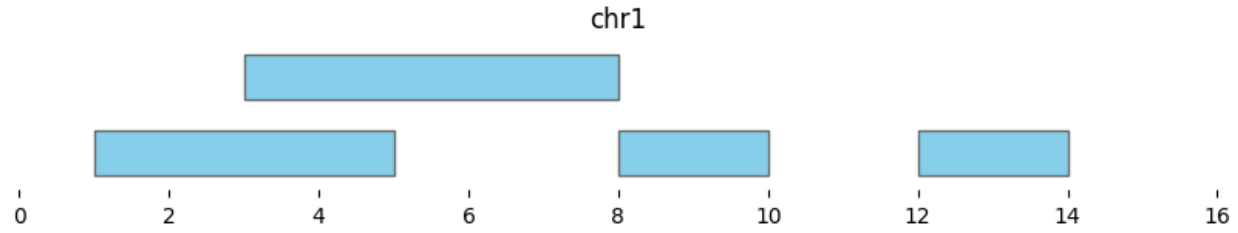
```
  chrom  start  end chrom_  start_  end_
0  chr1      1    5   chr1       4     8
1  chr1      3    8   chr1       4     8
2  chr1      8   10   None    <NA>  <NA>
3  chr1     12   14   None    <NA>  <NA>
```

2.3 Cluster

It is often useful to find overlapping intervals within a single set of genomic intervals. In *bioframe*, this is achieved with `bioframe.cluster()`. This function returns a DataFrame where subsets of overlapping intervals are assigned to the same group, reported in a new column.

To demonstrate the usage of `bioframe.cluster()`, we use the same `df1` as above:

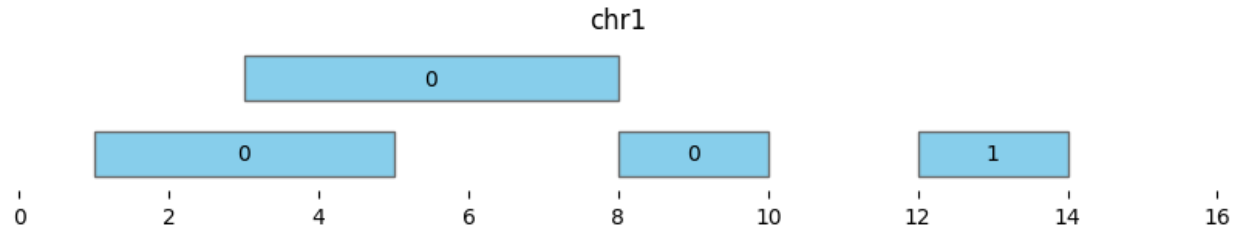
```
df1 = pd.DataFrame([
    ['chr1', 1, 5],
    ['chr1', 3, 8],
    ['chr1', 8, 10],
    ['chr1', 12, 14],
    ],
    columns=['chrom', 'start', 'end']
)

bf.vis.plot_intervals(df1, show_coords=True, xlim=(0,16))
```



Cluster returns a DataFrame where each interval is assigned to a group:

```
df_annotated = bf.cluster(df1, min_dist=0)
display(df_annotated)
bf.vis.plot_intervals(df_annotated, labels=df_annotated['cluster'], show_coords=True,␣
↪xlim=(0,16))
```

	chrom	start	end	cluster	cluster_start	cluster_end
0	chr1	1	5	0	1	10
1	chr1	3	8	0	1	10
2	chr1	8	10	0	1	10
3	chr1	12	14	1	12	14



Note that using `min_dist=0` and `min_dist=None` give different results, as the latter only clusters overlapping intervals and not adjacent intervals:

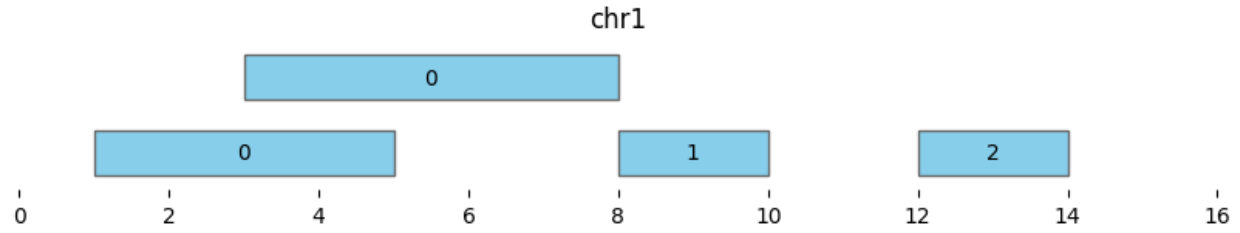
```
df_annotated = bf.cluster(df1, min_dist=None)
display(df_annotated)
```

(continues on next page)

(continued from previous page)

```
bf.vis.plot_intervals(df_annotated, labels=df_annotated['cluster'], show_coords=True,␣
↪xlim=(0,16))
```

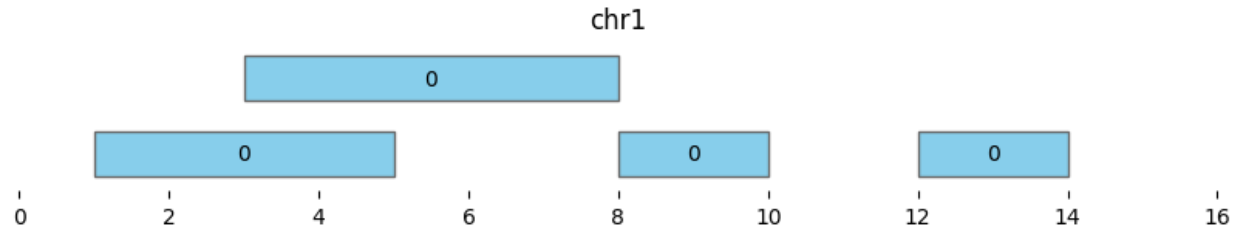
	chrom	start	end	cluster	cluster_start	cluster_end
0	chr1	1	5	0	1	8
1	chr1	3	8	0	1	8
2	chr1	8	10	1	8	10
3	chr1	12	14	2	12	14



Extending the minimum distance to two (`min_dist=2`) makes all intervals part of the same cluster "0":

```
df_annotated = bf.cluster(df1, min_dist=2)
display(df_annotated)
bf.vis.plot_intervals(df_annotated, labels=df_annotated['cluster'], show_coords=True,␣
↪xlim=(0,16))
```

	chrom	start	end	cluster	cluster_start	cluster_end
0	chr1	1	5	0	1	14
1	chr1	3	8	0	1	14
2	chr1	8	10	0	1	14
3	chr1	12	14	0	1	14



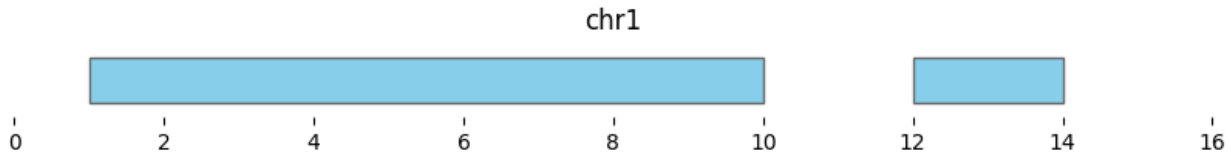
2.4 Merge

Instead of returning cluster assignments, `bioframe.merge()` returns a new dataframe of merged genomic intervals. As with `bioframe.cluster()`, using `min_dist=0` and `min_dist=None` gives different results.

If `min_dist=0`, this returns a dataframe of two intervals:

```
df_merged = bf.merge(df1, min_dist=0)

display(df_merged)
bf.vis.plot_intervals(df_merged, show_coords=True, xlim=(0,16))
```

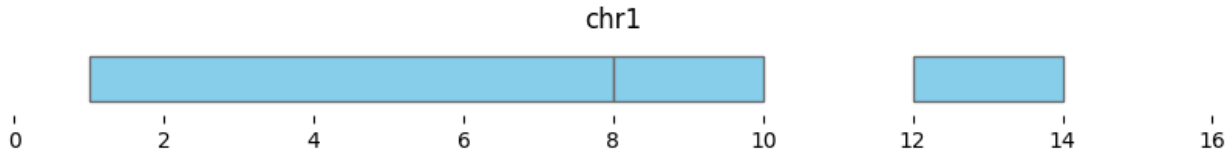
```
  chrom  start  end  n_intervals
0  chr1      1   10            3
1  chr1     12   14            1
```



If `min_dist=None`, this returns a dataframe of three intervals:

```
df_merged = bf.merge(df1, min_dist=None)
display(df_merged)
bf.vis.plot_intervals(df_merged, show_coords=True, xlim=(0,16))
```

```
  chrom  start  end  n_intervals
0  chr1      1    8            2
1  chr1      8   10            1
2  chr1     12   14            1
```



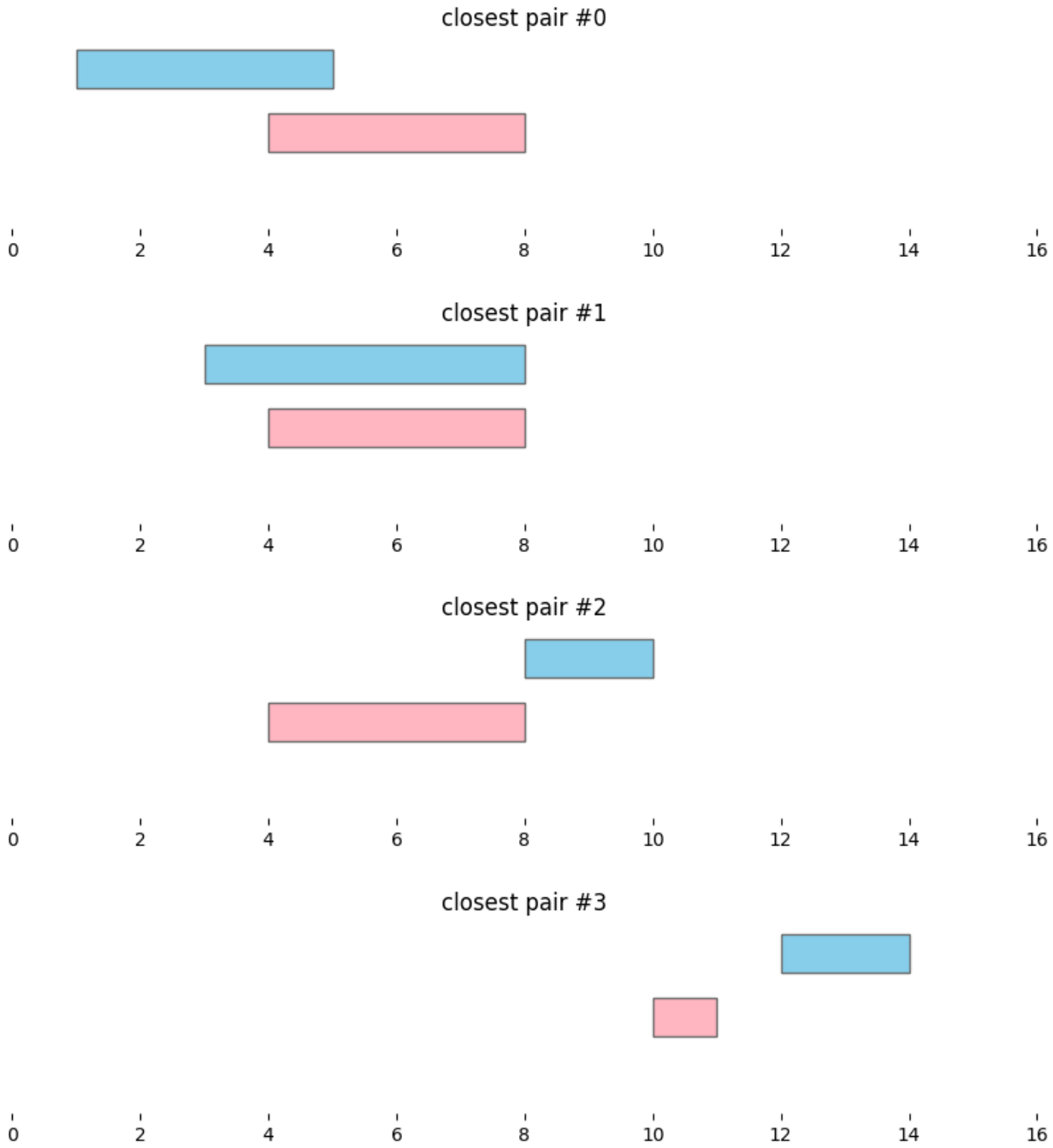
2.5 Closest

In genomics, it is often useful not only to find features that overlap, but also features that are nearby along the genome. In bioframe, this is achieved using `bioframe.closest()`.

```
closest_intervals = bf.closest(df1, df2, suffixes=('_1','_2'))
display(closest_intervals)
```

```
  chrom_1  start_1  end_1 chrom_2  start_2  end_2  distance
0    chr1        1      5    chr1        4      8         0
1    chr1        3      8    chr1        4      8         0
2    chr1        8     10    chr1        4      8         0
3    chr1       12     14    chr1       10     11         1
```

```
for i, reg_pair in closest_intervals.iterrows():
    bf.vis.plot_intervals_arr(
        starts = [reg_pair.start_1,reg_pair.start_2],
        ends = [reg_pair.end_1,reg_pair.end_2],
        colors = ['skyblue', 'lightpink'],
        levels = [2,1],
        xlim = (0,16),
        show_coords = True)
    plt.title(f'closest pair #{i}')
```



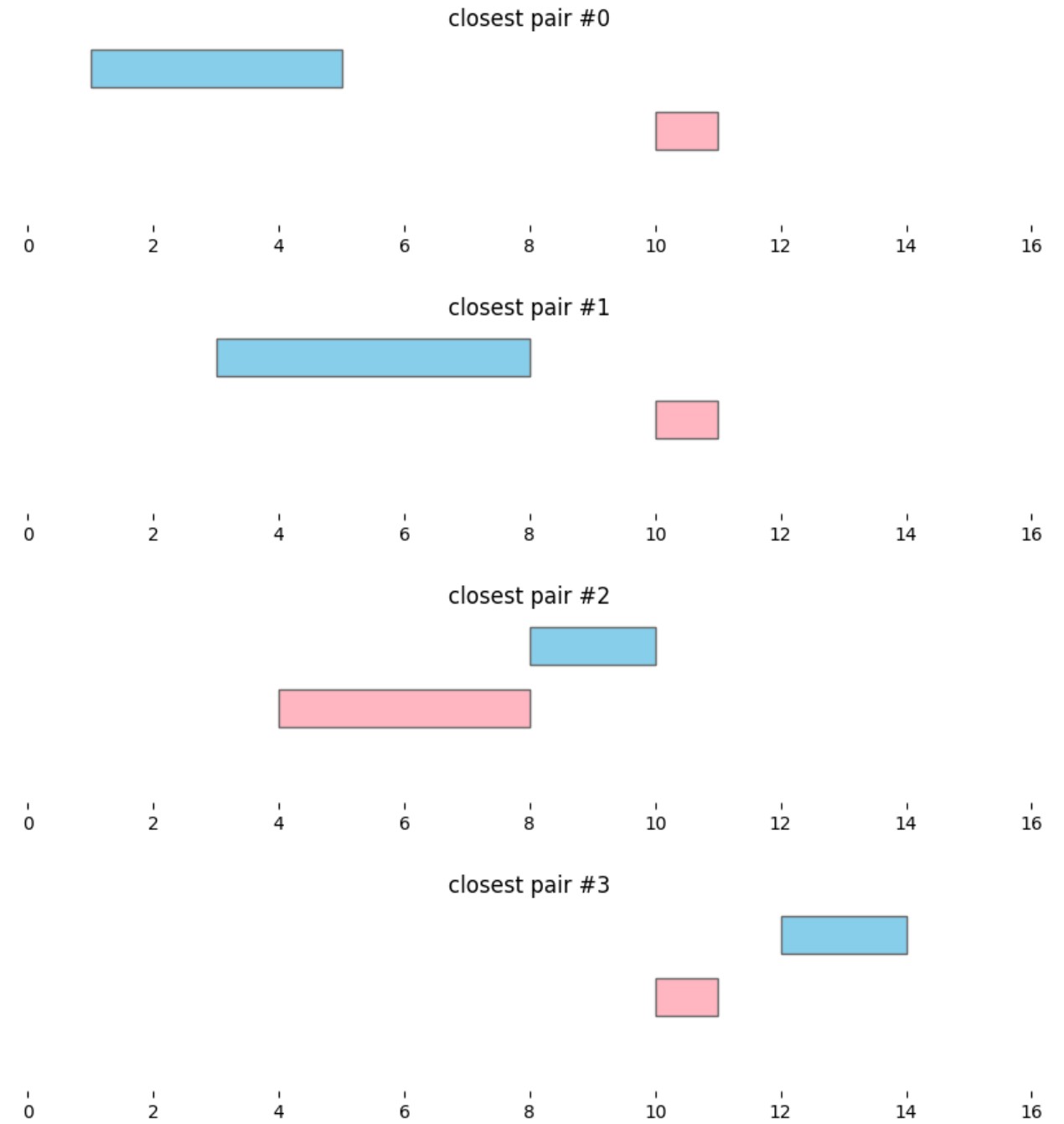
By default, `bioframe.closest()` reports overlapping intervals. This can be modified by passing `ignore_overlap=True`. Note the closest pair #2 and #3, which did not overlap, remain the same:

```
closest_intervals = bf.closest(df1, df2, ignore_overlaps=True, suffixes=('_1','_2'))
for i, reg_pair in closest_intervals.iterrows():
    bf.vis.plot_intervals_arr(
        starts = [reg_pair.start_1,reg_pair.start_2],
        ends = [reg_pair.end_1,reg_pair.end_2],
        colors = ['skyblue', 'lightpink'],
        levels = [2,1],
```

(continues on next page)

(continued from previous page)

```
        xlim = (0,16),
        show_coords = True)
    plt.title(f'closest pair #{i}')
```



Closest intervals within a single DataFrame can be found simply by passing a single dataframe to `bioframe.closest()`. The number of closest intervals to report per query interval can be adjusted with `k`.

```
bf.closest(df1, k=2)
```

```
  chrom  start  end chrom_  start_  end_  distance
0  chr1      1    5   chr1       3     8         0
1  chr1      1    5   chr1       8    10         3
2  chr1      3    8   chr1       1     5         0
3  chr1      3    8   chr1       8    10         0
4  chr1      8   10   chr1       3     8         0
5  chr1      8   10   chr1      12    14         2
6  chr1     12   14   chr1       8    10         2
7  chr1     12   14   chr1       3     8         4
```

Closest intervals upstream of the features in df1 can be found by ignoring downstream and overlaps. Upstream/downstream direction is defined by genomic coordinates by default (smaller coordinate is upstream).

```
bf.closest(df1, df2,
    ignore_overlaps=True,
    ignore_downstream=True)
```

```
  chrom  start  end chrom_  start_  end_  distance
0  chr1      8   10   chr1       4     8         0
1  chr1     12   14   chr1      10    11         1
2  chr1      1    5   <NA>    <NA>  <NA>      <NA>
3  chr1      3    8   <NA>    <NA>  <NA>      <NA>
```

If the features in df1 have direction (e.g., genes have transcription direction), then the definition of upstream/downstream direction can be changed to the direction of the features by *direction_col*:

```
df1["strand"] = np.where(np.random.rand(len(df1)) > 0.5, "+", "-")
bf.closest(df1, df2,
    ignore_overlaps=True,
    ignore_downstream=True,
    direction_col='strand')
```

```
  chrom  start  end strand chrom_  start_  end_  distance
0  chr1      1    5      -   chr1      10    11         5
1  chr1      8   10      +   chr1       4     8         0
2  chr1      3    8      +   <NA>    <NA>  <NA>      <NA>
3  chr1     12   14      -   <NA>    <NA>  <NA>      <NA>
```

2.6 Coverage & Count Overlaps

For two sets of genomic features, it is often useful to calculate the number of basepairs covered and the number of overlapping intervals. While these are fairly straightforward to compute from the output of `bioframe.overlap()` with `pandas.groupby()` and column renaming, since these are very frequently used, they are provided as core bioframe functions.

```
df1_coverage = bf.coverage(df1, df2)
display(df1_coverage)
```

```
  chrom  start  end strand  coverage
0  chr1      1    5      -         1
```

(continues on next page)

(continued from previous page)

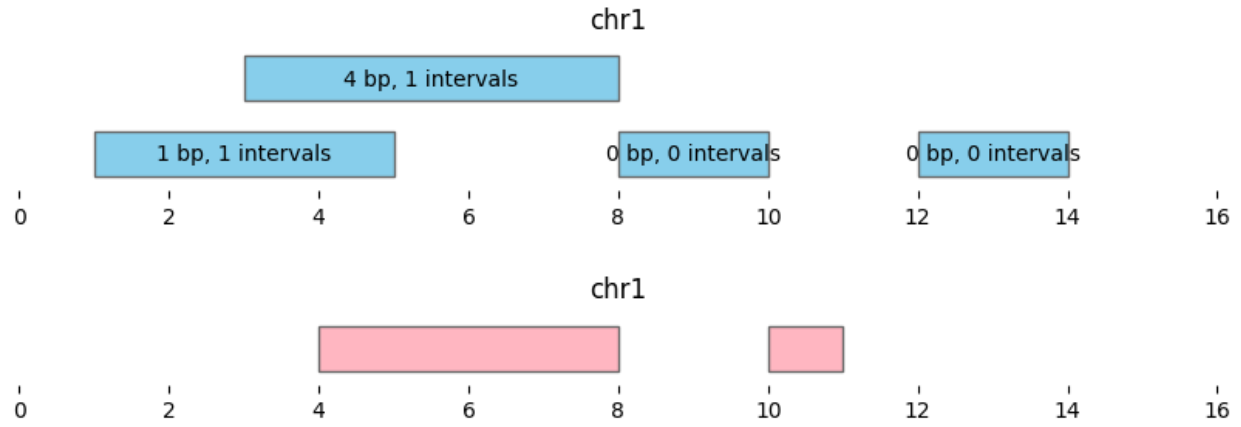
```
1  chr1      3    8      +         4
2  chr1      8   10      +         0
3  chr1     12   14      -         0
```

```
df1_coverage_and_count = bf.count_overlaps(df1_coverage, df2)
display(df1_coverage_and_count)
```

```
  chrom  start  end strand  coverage  count
0  chr1      1    5      -         1      1
1  chr1      3    8      +         4      1
2  chr1      8   10      +         0      0
3  chr1     12   14      -         0      0
```

This plot shows the coverage and number of overlaps for intervals in `df1` by `df2`:

```
bf.vis.plot_intervals(
    df1_coverage_and_count,
    show_coords=True, xlim=(0,16),
    labels = [f'{cov} bp, {n} intervals'
              for cov, n in zip(df1_coverage_and_count.coverage, df1_coverage_and_count[
↪'count'])])

bf.vis.plot_intervals(df2, show_coords=True, xlim=(0,16), colors='lightpink')
```



2.7 Subtract & Set Difference

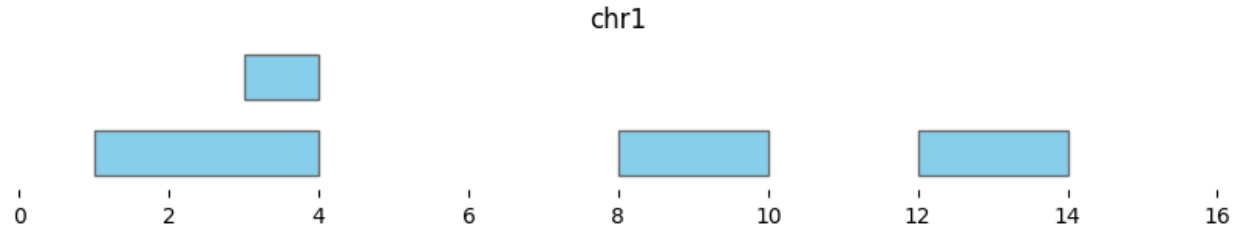
Bioframe has two functions for computing differences between sets of intervals: at the level of basepairs and at the level of whole intervals.

Basepair-level subtraction is performed with `bioframe.subtract()`:

```
subtracted_intervals = bf.subtract(df1, df2)
display(subtracted_intervals)
```

```
  chrom  start  end strand
0  chr1      1    4      -
1  chr1      3    4      +
2  chr1      8   10      +
3  chr1     12   14      -
```

```
bf.vis.plot_intervals(subtracted_intervals, show_coords=True, xlim=(0,16))
```

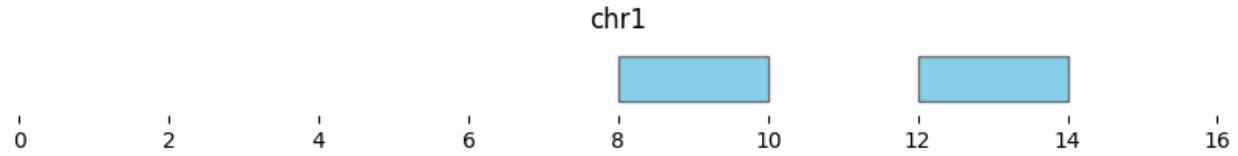


Interval-level differences are calculated with `bioframe.setdiff()`:

```
setdiff_intervals = bf.setdiff(df1, df2)
display(setdiff_intervals)
```

```
  chrom  start  end strand
2  chr1      8   10      +
3  chr1     12   14      -
```

```
bf.vis.plot_intervals(setdiff_intervals, show_coords=True, xlim=(0,16))
```



2.8 Expand

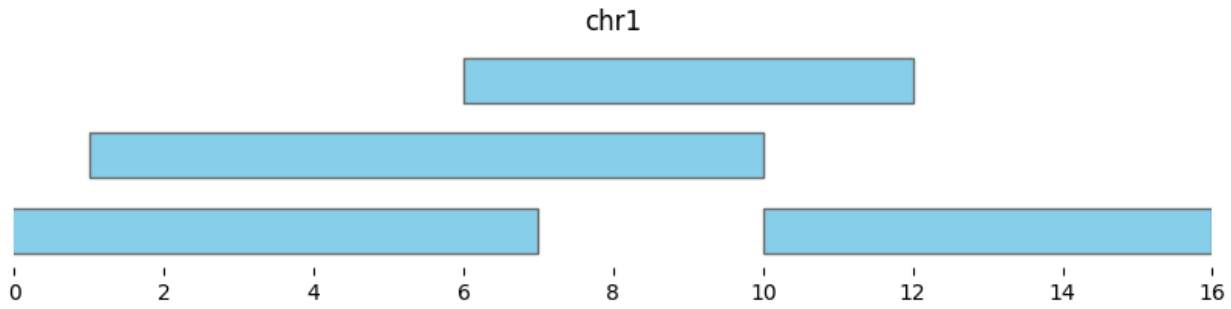
`bioframe.expand()` enables quick resizing of intervals.

Expand supports additive resizing, with `pad`. Note that unless subsequently trimmed (with `bioframe.trim()`), expanded intervals can have negative values:

```
expanded_intervals = bf.expand(df1, pad=2)
display(expanded_intervals)
```

```
  chrom  start  end strand
0  chr1     -1    7      -
1  chr1      1   10      +
2  chr1      6   12      +
3  chr1     10   16      -
```

```
bf.vis.plot_intervals(expanded_intervals, show_coords=True, xlim=(0,16))
```



Expand also supports multiplicative resizing, with `scale`. Note that `scale=0` resizes all intervals to their midpoints:

```
expanded_intervals = bf.expand(df1, scale=0)
display(expanded_intervals)
```

	chrom	start	end	strand
0	chr1	3	3	-
1	chr1	6	6	+
2	chr1	9	9	+
3	chr1	13	13	-

```
bf.vis.plot_intervals(expanded_intervals, show_coords=True, xlim=(0,16))
```

2.9 Genomic Views

Certain interval operations are often used relative to a set of reference intervals, whether those are chromosomes, scaffolds, or sub-intervals of either. Bioframe formalizes this with the concept of a *genomic view*, implemented as pandas dataframes, termed viewFrames, that satisfy the following:

- all requirements for bedFrames, including columns for ('chrom', 'start', 'end')
- it has an additional column, `view_name_col`, with default 'name'
- entries in the view_name_col are unique
- intervals are non-overlapping
- it does not contain null values

Importantly a *genomic view* specifies a global coordinate axis, i.e. a conversion from a genomic coordinate system to a single axis. See Technical Notes for more details.

Bioframe provides a function to assign intervals to corresponding intervals in a view, `bioframe.assign_view()`, a function to construct views from various input formats, *bioframe.core.construction.make_viewframe()*, and a function check that viewframe requirements are met, *bioframe.core.checks.is_viewframe()*.

The following genomic interval operations make use of views, though also have useful default behavior if no view is provided: `bioframe.complement()`, `bioframe.trim()`, `bioframe.sort_bedframe()`.

2.10 Complement

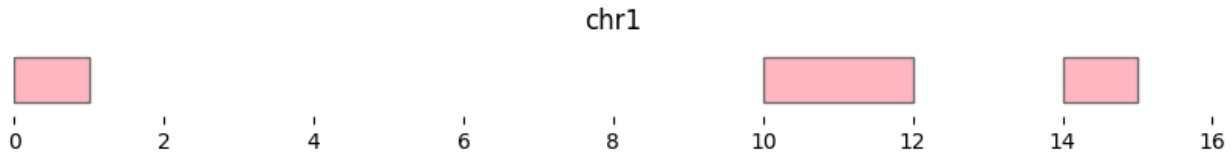
Equally important to finding which genomic features overlap is finding those that do not. `bioframe.complement()` returns a BedFrame of intervals not covered by any intervals in an input BedFrame.

Complments are defined relative to a *genomic view*. Here this is provided as a dictionary with a single chromosome of length 15:

```
df_complemented = bf.complement(df1, view_df={'chr1':15})
display(df_complemented)
```

	chrom	start	end	view_region
0	chr1	0	1	chr1
1	chr1	10	12	chr1
2	chr1	14	15	chr1

```
bf.vis.plot_intervals(df_complemented, show_coords=True, xlim=(0,16), colors='lightpink')
```



If no view is provided, complement is calculated per unique chromosome in the input with right limits of `np.iinfo(np.int64).max`.

```
df_complemented = bf.complement(df1)
display(df_complemented)
```

	chrom	start	end	view_region
0	chr1	0	1	chr1
1	chr1	10	12	chr1
2	chr1	14	9223372036854775807	chr1

2.11 Trim

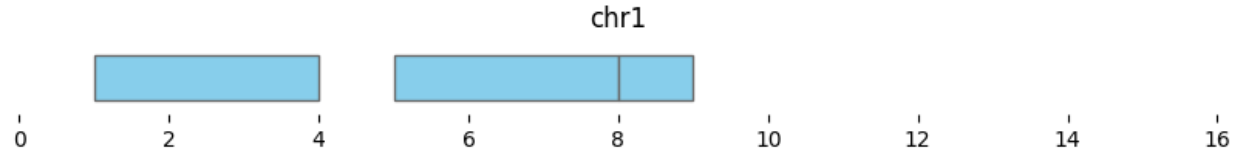
Certain regions are often best avoided for genomic analyses. `bioframe.trim()` trims intervals to a specified view. Intervals falling outside of view regions have their filled with null values.

```
view_df = pd.DataFrame(
    [
        ["chr1", 0, 4, "chr1p"],
        ["chr1", 5, 9, "chr1q"],
        ["chrX", 0, 50, "chrX"],
        ["chrM", 0, 10, "chrM"]],
    columns=["chrom", "start", "end", "name"],
)

trimmed_intervals = bf.trim(df1, view_df)
display(trimmed_intervals)
```

```
  chrom  start  end strand
0  chr1    1.0  4.0      -
1  chr1    5.0  8.0      +
2  chr1    8.0  9.0      +
3  <NA>    NaN  NaN      -
```

Note that the last interval of `df1` fell beyond 'chr1q' and is now null, and the last interval now ends at 9 instead of 10.

```
bf.vis.plot_intervals(trimmed_intervals, show_coords=True, xlim=(0,16))
```



If no view is provided, this function trims intervals at zero to avoid negative values.

2.12 Sorting

If no view is provided, `bioframe.sort_bedframe()` sorts by ("chrom", "start", "end") columns:

```
df_unsorted = pd.DataFrame([
    ['chrM', 3, 8],
    ['chrM', 1, 5],
    ['chrX', 12, 14],
    ['chrX', 8, 10]],
    columns=['chrom', 'start', 'end']
)

display( bf.sort_bedframe(df_unsorted) )
```

```
  chrom  start  end
0  chrM      1    5
```

(continues on next page)

(continued from previous page)

```
1  chrM      3    8
2  chrX      8   10
3  chrX     12   14
```

Views enable a specifying a sort order on a set of intervals. This flexibility is useful when the desired sorting order is non-lexicographical, e.g. with chrM after autosomes and chrX:

```
display( bf.sort_bedframe(df_unsorted, view_df) )
```

```
  chrom  start  end
0  chrX      8   10
1  chrX     12   14
2  chrM      1    5
3  chrM      3    8
```

2.13 Selecting & Slicing

Since bioFrame operates directly with pandas *DataFrames*, all typical selection and slicing operations are directly relevant.

Bioframe also provides a function `bioframe.select()` that enables selecting interval subsets using UCSC string format:

```
display( bioframe.select(df_unsorted,'chrX:8-14') )
```

```
  chrom  start  end
2  chrX     12   14
3  chrX      8   10
```

2.14 Flexible column naming

Genomic analyses often deal with dataframes with inhomogeneously named columns. Bioframe offers a way to set the default column names that are most convenient for your analyses.

Default bedframe column names are stored in `bioframe.core.specs_rc`.

```
bf.core.specs._rc
```

```
{'colnames': {'chrom': 'chrom', 'start': 'start', 'end': 'end'}}
```

If the dataframes we wish to work with have `['CHROMOSOME', 'LEFT', 'RIGHT']`, we can either pass cols to operations in `bioframe.ops`:

```
df1_diff_colnames = pd.DataFrame([
    ['chr1', 1, 5],
    ['chr1', 3, 8]],
    columns=['CHROMOSOME', 'LEFT', 'RIGHT']
)
```

(continues on next page)

(continued from previous page)

```
df2_diff_colnames = pd.DataFrame([
    ['chr1', 4, 8],
    ['chr1', 10, 11]],
    columns=['CHROMOSOME', 'LEFT', 'RIGHT']
)
```

```
bf.overlap(
    df1_diff_colnames, df2_diff_colnames,
    cols1=['CHROMOSOME', 'LEFT', 'RIGHT'],
    cols2=['CHROMOSOME', 'LEFT', 'RIGHT'],
)
```

```
  CHROMOSOME  LEFT  RIGHT CHROMOSOME_  LEFT_  RIGHT_
0       chr1     1      5        chr1      4       8
1       chr1     3      8        chr1      4       8
```

Or, we can update the default column names:

```
with bf.core.specs.update_default_colnames(['CHROMOSOME', 'LEFT', 'RIGHT']):
    display(bf.overlap(
        df1_diff_colnames, df2_diff_colnames,
    ))
```

```
  CHROMOSOME  LEFT  RIGHT CHROMOSOME_  LEFT_  RIGHT_
0       chr1     1      5        chr1      4       8
1       chr1     3      8        chr1      4       8
```

```
# setting colnames back to default.
bf.core.specs.update_default_colnames(['chrom', 'start', 'end'])
bf.core.specs._rc
```

```
{'colnames': {'chrom': 'chrom', 'start': 'start', 'end': 'end'}}
```

CHAPTER

THREE

READING GENOMIC DATAFRAMES

```
import bioframe
```

Bioframe provides multiple methods to convert data stored in common genomic file formats to pandas dataFrames in `bioframe.io`.

3.1 Reading tabular text data

The most common need is to read tablular data, which can be accomplished with `bioframe.read_table`. This function wraps pandas `pandas.read_csv/pandas.read_table` (tab-delimited by default), but allows the user to easily pass a **schema** (i.e. list of pre-defined column names) for common genomic interval-based file formats.

For example,

```
df = bioframe.read_table(
    "https://www.encodeproject.org/files/ENCFF001XKR/@@download/ENCFF001XKR.bed.gz",
    schema="bed9",
)
display(df[0:3])
```

```
  chrom   start     end name  score strand thickStart thickEnd   itemRgb
0  chr1  193500  194500    .    400      +          .        .  179,45,0
1  chr1  618500  619500    .    700      +          .        .  179,45,0
2  chr1  974500  975500    .   1000      +          .        .  179,45,0
```

```
df = bioframe.read_table(
    "https://www.encodeproject.org/files/ENCFF401MQL/@@download/ENCFF401MQL.bed.gz",
    schema="narrowPeak",
)
display(df[0:3])
```

```
   chrom      start        end name  score strand       fc  -log10p  -log10q  \
0  chr19   48309541   48309911    .   1000      .  5.04924     -1.0  0.00438
1   chr4  130563716  130564086    .    993      .  5.05052     -1.0  0.00432
2   chr1  200622507  200622877    .    591      .  5.05489     -1.0  0.00400

   relSummit
0        185
1        185
2        185
```

```
df = bioframe.read_table(
    "https://www.encodeproject.org/files/ENCFF001VRS/@@download/ENCFF001VRS.bed.gz",
    schema="bed12",
)
display(df[0:3])
```

```
   chrom      start        end                                          name  \
0  chr19   54331773   54620705    5C_304_ENm007_FOR_1.5C_304_ENm007_REV_40
1  chr19   54461360   54620705   5C_304_ENm007_FOR_26.5C_304_ENm007_REV_40
2   chr5  131346229  132145236  5C_299_ENm002_FOR_241.5C_299_ENm002_REV_33

   score strand  thickStart   thickEnd  itemRgb  blockCount    blockSizes  \
0   1000      .    54331773   54620705        0           2  14528,19855,
1   1000      .    54461360   54620705        0           2    800,19855,
2   1000      .   131346229  132145236        0           2    2609,2105,

  blockStarts
0   0,269077,
1   0,139490,
2   0,796902,
```

The `schema` argument looks up file type from a registry of schemas stored in the `bioframe.SCHEMAS` dictionary:

```
bioframe.SCHEMAS["bed6"]
```

```
['chrom', 'start', 'end', 'name', 'score', 'strand']
```

3.2 UCSC Big Binary Indexed files (BigWig, BigBed)

Bioframe also has convenience functions for reading and writing bigWig and bigBed binary files to and from pandas DataFrames.

```
bw_url = "http://genome.ucsc.edu/goldenPath/help/examples/bigWigExample.bw"
df = bioframe.read_bigwig(bw_url, "chr21", start=10_000_000, end=10_010_000)
df.head(5)
```

```
   chrom     start       end  value
0  chr21  10000000  10000005   40.0
1  chr21  10000005  10000010   40.0
2  chr21  10000010  10000015   60.0
3  chr21  10000015  10000020   80.0
4  chr21  10000020  10000025   40.0
```

```
df["value"] *= 100
df.head(5)
```

```
   chrom     start       end   value
0  chr21  10000000  10000005  4000.0
1  chr21  10000005  10000010  4000.0
```

(continues on next page)

(continued from previous page)

```
2  chr21  10000010  10000015  6000.0
3  chr21  10000015  10000020  8000.0
4  chr21  10000020  10000025  4000.0
```

```
chromsizes = bioframe.fetch_chromsizes("hg19")
# bioframe.to_bigwig(df, chromsizes, 'times100.bw')

# note: requires UCSC bedGraphToBigWig binary, which can be installed as
# !conda install -y -c bioconda ucsc-bedgraphtobigwig
```

```
bb_url = "http://genome.ucsc.edu/goldenPath/help/examples/bigBedExample.bb"
bioframe.read_bigbed(bb_url, "chr21", start=48000000).head()
```

```
   chrom     start       end
0  chr21  48003453  48003785
1  chr21  48003545  48003672
2  chr21  48018114  48019432
3  chr21  48018244  48018550
4  chr21  48018843  48019099
```

3.3 Reading genome assembly information

The most fundamental information about a genome assembly is its set of chromosome sizes.

Bioframe provides functions to read chromosome sizes file as `pandas.Series`, with some useful filtering and sorting options:

```
bioframe.read_chromsizes(
    "https://hgdownload.soe.ucsc.edu/goldenPath/hg38/bigZips/hg38.chrom.sizes"
)
```

```
chr1     248956422
chr2     242193529
chr3     198295559
chr4     190214555
chr5     181538259
chr6     170805979
chr7     159345973
chr8     145138636
chr9     138394717
chr10    133797422
chr11    135086622
chr12    133275309
chr13    114364328
chr14    107043718
chr15    101991189
chr16     90338345
chr17     83257441
chr18     80373285
```

(continues on next page)

(continued from previous page)

```
chr19     58617616
chr20     64444167
chr21     46709983
chr22     50818468
chrX     156040895
chrY      57227415
chrM         16569
Name: length, dtype: int64
```

```
bioframe.read_chromsizes(
    "https://hgdownload.soe.ucsc.edu/goldenPath/hg38/bigZips/hg38.chrom.sizes",
    filter_chroms=False,
)
```

```
chr1               248956422
chr2               242193529
chr3               198295559
chr4               190214555
chr5               181538259
                       ...
chrUn_KI270539v1         993
chrUn_KI270385v1         990
chrUn_KI270423v1         981
chrUn_KI270392v1         971
chrUn_KI270394v1         970
Name: length, Length: 455, dtype: int64
```

```
dm6_url = "https://hgdownload.soe.ucsc.edu/goldenPath/dm6/database/chromInfo.txt.gz"
```

```
bioframe.read_chromsizes(
    dm6_url,
    filter_chroms=True,
    chrom_patterns=("^chr2L$", "^chr2R$", "^chr3L$", "^chr3R$", "^chr4$", "^chrX$"),
)
```

```
chr2L    23513712
chr2R    25286936
chr3L    28110227
chr3R    32079331
chr4      1348131
chrX     23542271
Name: length, dtype: int64
```

```
bioframe.read_chromsizes(
    dm6_url, chrom_patterns=[r"^chr\d+L$", r"^chr\d+R$", "^chr4$", "^chrX$", "^chrM$"]
)
```

```
chr2L    23513712
chr3L    28110227
chr2R    25286936
```

(continues on next page)

(continued from previous page)

```
chr3R    32079331
chr4      1348131
chrX     23542271
chrM        19524
Name: length, dtype: int64
```

Bioframe provides a convenience function to fetch chromosome sizes from UCSC given an assembly name:

```
chromsizes = bioframe.fetch_chromsizes("hg38")
chromsizes[-5:]
```

```
name
chr21     46709983
chr22     50818468
chrX     156040895
chrY      57227415
chrM         16569
Name: length, dtype: int64
```

Bioframe can also generate a list of centromere positions using information from some UCSC assemblies:

```
display(bioframe.fetch_centromeres("hg38")[:3])
```

```
  chrom      start        end        mid
0  chr1  121700000  125100000  123400000
1  chr2   91800000   96000000   93900000
2  chr3   87800000   94000000   90900000
```

These functions are just wrappers for a UCSC client. Users can also use `UCSCClient` directly:

```
client = bioframe.UCSCClient("hg38")
client.fetch_cytoband()
```

```
                    chrom    start       end    name gieStain
0                    chr1        0   2300000  p36.33     gneg
1                    chr1  2300000   5300000  p36.32   gpos25
2                    chr1  5300000   7100000  p36.31     gneg
3                    chr1  7100000   9100000  p36.23   gpos25
4                    chr1  9100000  12500000  p36.22     gneg
...                   ...      ...       ...     ...      ...
1544  chr19_MU273387v1_alt        0     89211     NaN     gneg
1545  chr16_MU273376v1_fix        0     87715     NaN     gneg
1546   chrX_MU273393v1_fix        0     68810     NaN     gneg
1547   chr8_MU273360v1_fix        0     39290     NaN     gneg
1548   chr5_MU273352v1_fix        0     34400     NaN     gneg

[1549 rows x 5 columns]
```

3.4 Curated genome assembly build information

New in v0.5.0

Bioframe also has locally stored information for common genome assembly builds.

For a given provider and assembly build, this API provides additional sequence metadata:

- A canonical **name** for every sequence, usually opting for UCSC-style naming.
- A canonical **ordering** of the sequences.
- Each sequence's **length**.
- An **alias dictionary** mapping alternative names or aliases to the canonical sequence name.
- Each sequence is assigned to an assembly **unit**: e.g., primary, non-nuclear, decoy.
- Each sequence is assigned a **role**: e.g., assembled molecule, unlocalized, unplaced.

```
bioframe.assemblies_available()
```

```
                   organism provider provider_build  release_year  \
0              homo sapiens     ncbi          GRCh37          2009
1              homo sapiens     ucsc            hg19          2009
2              homo sapiens     ncbi          GRCh38          2013
3              homo sapiens     ucsc            hg38          2013
4              homo sapiens     ncbi  T2T-CHM13v2.0          2022
5              homo sapiens     ucsc             hs1          2022
6              mus musculus     ncbi         MGSCv37          2010
7              mus musculus     ucsc             mm9          2007
8              mus musculus     ncbi          GRCm38          2011
9              mus musculus     ucsc            mm10          2011
10             mus musculus     ncbi          GRCm39          2020
11             mus musculus     ucsc            mm39          2020
12   drosophila melanogaster     ucsc             dm3          2006
13   drosophila melanogaster     ucsc             dm6          2014
14    caenorhabditis elegans     ucsc            ce10          2010
15    caenorhabditis elegans     ucsc            ce11          2013
16              danio rerio     ucsc        danRer10          2014
17              danio rerio     ucsc        danRer11          2017
18  saccharomyces cerevisiae     ucsc         sacCer3          2011

                seqinfo          cytobands default_roles  \
0       hg19.seqinfo.tsv  hg19.cytoband.tsv   [assembled]
1       hg19.seqinfo.tsv  hg19.cytoband.tsv   [assembled]
2       hg38.seqinfo.tsv  hg38.cytoband.tsv   [assembled]
3       hg38.seqinfo.tsv  hg38.cytoband.tsv   [assembled]
4        hs1.seqinfo.tsv   hs1.cytoband.tsv   [assembled]
5        hs1.seqinfo.tsv   hs1.cytoband.tsv   [assembled]
6        mm9.seqinfo.tsv                NaN   [assembled]
7        mm9.seqinfo.tsv                NaN   [assembled]
8       mm10.seqinfo.tsv                NaN   [assembled]
9       mm10.seqinfo.tsv                NaN   [assembled]
10      mm39.seqinfo.tsv                NaN   [assembled]
11      mm39.seqinfo.tsv                NaN   [assembled]
```

(continues on next page)

(continued from previous page)

```
12       dm3.seqinfo.tsv                NaN   [assembled]
13       dm6.seqinfo.tsv                NaN   [assembled]
14      ce10.seqinfo.tsv                NaN   [assembled]
15      ce11.seqinfo.tsv                NaN   [assembled]
16  danRer10.seqinfo.tsv                NaN   [assembled]
17  danRer10.seqinfo.tsv                NaN   [assembled]
18   sacCer3.seqinfo.tsv                NaN   [assembled]

                     default_units  \
0   [primary, non-nuclear-revised]
1           [primary, non-nuclear]
2           [primary, non-nuclear]
3           [primary, non-nuclear]
4           [primary, non-nuclear]
5           [primary, non-nuclear]
6           [primary, non-nuclear]
7           [primary, non-nuclear]
8           [primary, non-nuclear]
9           [primary, non-nuclear]
10          [primary, non-nuclear]
11          [primary, non-nuclear]
12          [primary, non-nuclear]
13          [primary, non-nuclear]
14          [primary, non-nuclear]
15          [primary, non-nuclear]
16          [primary, non-nuclear]
17          [primary, non-nuclear]
18          [primary, non-nuclear]

                                                  url
0   https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/0...
1   https://hgdownload.soe.ucsc.edu/goldenPath/hg1...
2   https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/0...
3   https://hgdownload.soe.ucsc.edu/goldenPath/hg3...
4   https://ftp.ncbi.nlm.nih.gov/genomes/all/GCA/0...
5   https://hgdownload.soe.ucsc.edu/goldenPath/hs1...
6   https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/0...
7   https://hgdownload.soe.ucsc.edu/goldenPath/mm9...
8   https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/0...
9   https://hgdownload.soe.ucsc.edu/goldenPath/mm1...
10  https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/0...
11  https://hgdownload.soe.ucsc.edu/goldenPath/mm3...
12  https://hgdownload.soe.ucsc.edu/goldenPath/dm3...
13  https://hgdownload.soe.ucsc.edu/goldenPath/dm6...
14  https://hgdownload.soe.ucsc.edu/goldenPath/ce1...
15  https://hgdownload.soe.ucsc.edu/goldenPath/ce1...
16  https://hgdownload.soe.ucsc.edu/goldenPath/dan...
17  https://hgdownload.soe.ucsc.edu/goldenPath/dan...
18  https://hgdownload.soe.ucsc.edu/goldenPath/sac...
```

```
hg38 = bioframe.assembly_info("hg38")
print(hg38.provider, hg38.provider_build)
```

(continues on next page)

(continued from previous page)

```
hg38.seqinfo
```

```
ucsc hg38
```

```
     name     length       role molecule         unit  \
0    chr1  248956422  assembled     chr1      primary
1    chr2  242193529  assembled     chr2      primary
2    chr3  198295559  assembled     chr3      primary
3    chr4  190214555  assembled     chr4      primary
4    chr5  181538259  assembled     chr5      primary
5    chr6  170805979  assembled     chr6      primary
6    chr7  159345973  assembled     chr7      primary
7    chr8  145138636  assembled     chr8      primary
8    chr9  138394717  assembled     chr9      primary
9   chr10  133797422  assembled    chr10      primary
10  chr11  135086622  assembled    chr11      primary
11  chr12  133275309  assembled    chr12      primary
12  chr13  114364328  assembled    chr13      primary
13  chr14  107043718  assembled    chr14      primary
14  chr15  101991189  assembled    chr15      primary
15  chr16   90338345  assembled    chr16      primary
16  chr17   83257441  assembled    chr17      primary
17  chr18   80373285  assembled    chr18      primary
18  chr19   58617616  assembled    chr19      primary
19  chr20   64444167  assembled    chr20      primary
20  chr21   46709983  assembled    chr21      primary
21  chr22   50818468  assembled    chr22      primary
22   chrX  156040895  assembled     chrX      primary
23   chrY   57227415  assembled     chrY      primary
24   chrM      16569  assembled     chrM  non-nuclear

                      aliases
0    1,CM000663.2,NC_000001.11
1    2,CM000664.2,NC_000002.12
2    3,CM000665.2,NC_000003.12
3    4,CM000666.2,NC_000004.12
4    5,CM000667.2,NC_000005.10
5    6,CM000668.2,NC_000006.12
6    7,CM000669.2,NC_000007.14
7    8,CM000670.2,NC_000008.11
8    9,CM000671.2,NC_000009.12
9   10,CM000672.2,NC_000010.11
10  11,CM000673.2,NC_000011.10
11  12,CM000674.2,NC_000012.12
12  13,CM000675.2,NC_000013.11
13   14,CM000676.2,NC_000014.9
14  15,CM000677.2,NC_000015.10
15  16,CM000678.2,NC_000016.10
16  17,CM000679.2,NC_000017.11
17  18,CM000680.2,NC_000018.10
18  19,CM000681.2,NC_000019.10
```

(continues on next page)

(continued from previous page)

```
19  20,CM000682.2,NC_000020.11
20   21,CM000683.2,NC_000021.9
21  22,CM000684.2,NC_000022.11
22   X,CM000685.2,NC_000023.11
23   Y,CM000686.2,NC_000024.10
24     MT,J01415.2,NC_012920.1
```

```
hg38.chromsizes
```

```
name
chr1     248956422
chr2     242193529
chr3     198295559
chr4     190214555
chr5     181538259
chr6     170805979
chr7     159345973
chr8     145138636
chr9     138394717
chr10    133797422
chr11    135086622
chr12    133275309
chr13    114364328
chr14    107043718
chr15    101991189
chr16     90338345
chr17     83257441
chr18     80373285
chr19     58617616
chr20     64444167
chr21     46709983
chr22     50818468
chrX     156040895
chrY      57227415
chrM         16569
Name: length, dtype: int64
```

```
hg38.alias_dict["MT"]
```

```
'chrM'
```

```
bioframe.assembly_info("hg38", roles="all").seqinfo
```

```
                name     length       role molecule     unit  \
0               chr1  248956422  assembled     chr1  primary
1               chr2  242193529  assembled     chr2  primary
2               chr3  198295559  assembled     chr3  primary
3               chr4  190214555  assembled     chr4  primary
4               chr5  181538259  assembled     chr5  primary
..               ...        ...        ...      ...      ...
```

(continues on next page)

(continued from previous page)

```
189  chrUn_KI270753v1      62944   unplaced      NaN  primary
190  chrUn_KI270754v1      40191   unplaced      NaN  primary
191  chrUn_KI270755v1      36723   unplaced      NaN  primary
192  chrUn_KI270756v1      79590   unplaced      NaN  primary
193  chrUn_KI270757v1      71251   unplaced      NaN  primary

                                        aliases
0                      1,CM000663.2,NC_000001.11
1                      2,CM000664.2,NC_000002.12
2                      3,CM000665.2,NC_000003.12
3                      4,CM000666.2,NC_000004.12
4                      5,CM000667.2,NC_000005.10
..                                           ...
189  HSCHRUN_RANDOM_CTG30,KI270753.1,NT_187508.1
190  HSCHRUN_RANDOM_CTG33,KI270754.1,NT_187509.1
191  HSCHRUN_RANDOM_CTG34,KI270755.1,NT_187510.1
192  HSCHRUN_RANDOM_CTG35,KI270756.1,NT_187511.1
193  HSCHRUN_RANDOM_CTG36,KI270757.1,NT_187512.1

[194 rows x 6 columns]
```

3.5 Contributing metadata for a new assembly build

To contribute a new assembly build to bioframe's internal metadata registry, make a pull request with the following items:

1. Add a record to the assembly manifest file located at `bioframe/io/data/_assemblies.yml`. Required fields are as shown in the example below.
2. Create a `seqinfo.tsv` file for the new assembly build and place it in `bioframe/io/data`. Reference the exact file name in the manifest record's `seqinfo` field. The seqinfo is a tab-delimited file with a required header line as shown in the example below.
3. Optionally, a `cytoband.tsv` file adapted from a `cytoBand.txt` file from UCSC.

Note that we currently do not include sequences with alt or patch roles in seqinfo files.

3.5.1 Example

Metadata for the mouse mm9 assembly build as provided by UCSC.

`_assemblies.yml`

```
...
- organism: mus musculus
  provider: ucsc
  provider_build: mm9
  release_year: 2007
  seqinfo: mm9.seqinfo.tsv
  default_roles: [assembled]
  default_units: [primary, non-nuclear]
```

(continues on next page)

(continued from previous page)

```
  url: https://hgdownload.soe.ucsc.edu/goldenPath/mm9/bigZips/
...
```

mm9.seqinfo.tsv

```
name        length        role        molecule        unit        aliases
chr1        197195432        assembled        chr1        primary        1,
↪CM000994.1,NC_000067.5
chr2        181748087        assembled        chr2        primary        2,
↪CM000995.1,NC_000068.6
chr3        159599783        assembled        chr3        primary        3,
↪CM000996.1,NC_000069.5
chr4        155630120        assembled        chr4        primary        4,
↪CM000997.1,NC_000070.5
chr5        152537259        assembled        chr5        primary        5,
↪CM000998.1,NC_000071.5
chr6        149517037        assembled        chr6        primary        6,
↪CM000999.1,NC_000072.5
chr7        152524553        assembled        chr7        primary        7,
↪CM001000.1,NC_000073.5
chr8        131738871        assembled        chr8        primary        8,
↪CM001001.1,NC_000074.5
chr9        124076172        assembled        chr9        primary        9,
↪CM001002.1,NC_000075.5
chr10        129993255        assembled        chr10        primary        10,
↪CM001003.1,NC_000076.5
chr11        121843856        assembled        chr11        primary        11,
↪CM001004.1,NC_000077.5
chr12        121257530        assembled        chr12        primary        12,
↪CM001005.1,NC_000078.5
chr13        120284312        assembled        chr13        primary        13,
↪CM001006.1,NC_000079.5
chr14        125194864        assembled        chr14        primary        14,
↪CM001007.1,NC_000080.5
chr15        103494974        assembled        chr15        primary        15,
↪CM001008.1,NC_000081.5
chr16        98319150        assembled        chr16        primary        16,
↪CM001009.1,NC_000082.5
chr17        95272651        assembled        chr17        primary        17,
↪CM001010.1,NC_000083.5
chr18        90772031        assembled        chr18        primary        18,
↪CM001011.1,NC_000084.5
chr19        61342430        assembled        chr19        primary        19,
↪CM001012.1,NC_000085.5
chrX        166650296        assembled        chrX        primary        X,
↪CM001013.1,NC_000086.6
chrY        15902555        assembled        chrY        primary        Y,
↪CM001014.1,NC_000087.6
chrM        16299        assembled        chrM        non-nuclear        MT,
↪AY172335.1,NC_005089.1
chr1_
↪random        1231697        unlocalized        chr1        primary
```

(continues on next page)

(continued from previous page)

```
chr3_random         41899         unlocalized         chr3         primary
chr4_random         160594         unlocalized         chr4         primary
chr5_random         357350         unlocalized         chr5         primary
chr7_random         362490         unlocalized         chr7         primary
chr8_random         849593         unlocalized         chr8         primary
chr9_random         449403         unlocalized         chr9         primary
chr13_
↪random         400311         unlocalized         chr13         primary
chr16_random         3994         unlocalized         chr16         primary
chr17_
↪random         628739         unlocalized         chr17         primary
chrX_
↪random         1785075         unlocalized         chrX         primary
chrY_
↪random         58682461         unlocalized         chrY         primary
chrUn_random         5900358         unplaced                 primary
```

3.6 Reading other genomic formats

See the docs for File I/O for other supported file formats.

CHAPTER

FOUR

PERFORMANCE

This notebook illustrates performance of typical use cases for bioframe on sets of randomly generated intervals.

```
# ! pip install pyranges
## Optional:
# ! conda install -c bioconda bedtools
# ! pip install pybedtools
```

```
import platform

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import psutil
import pyranges

import bioframe

plt.rcParams["figure.facecolor"] = "white"
plt.rcParams["font.size"] = 16

# Note that by default we switch off the demo of pybedtools.
# It runs for minutes for 10^5 intervals
include_pybedtools = False
if include_pybedtools:
    import pybedtools

    pybedtools.helpers.set_bedtools_path(
        path="/usr/bin/"
    )  # Set the path to bedtools CLI
```

```
print(f"Bioframe v.{bioframe.__version__}")
print(f"PyRanges v.{pyranges.__version__}")
if include_pybedtools:
    print(f"Pybedtools v.{pybedtools.__version__}")

print(f"System Platform: {platform.platform()}")
print(f"{psutil.cpu_count()} CPUs at {psutil.cpu_freq().current:.0f} GHz")
```

```
Bioframe v.0.5.1
PyRanges v.0.0.129
```

(continues on next page)

(continued from previous page)

```
System Platform: Linux-5.19.0-46-generic-x86_64-with-glibc2.35
24 CPUs at 3040 GHz
```

Below we define a function to generate random intervals with various properties, returning a dataframe of intervals.

```
def make_random_intervals(
    n=1e5,
    n_chroms=1,
    max_coord=None,
    max_length=10,
    sort=False,
    categorical_chroms=False,
):
    n = int(n)
    n_chroms = int(n_chroms)
    max_coord = (n // n_chroms) if max_coord is None else int(max_coord)
    max_length = int(max_length)

    chroms = np.array(["chr" + str(i + 1) for i in range(n_chroms)])[
        np.random.randint(0, n_chroms, n)
    ]
    starts = np.random.randint(0, max_coord, n)
    ends = starts + np.random.randint(1, max_length, n)

    df = pd.DataFrame({"chrom": chroms, "start": starts, "end": ends})

    if categorical_chroms:
        df["chrom"] = df["chrom"].astype("category")

    if sort:
        df = df.sort_values(["chrom", "start", "end"]).reset_index(drop=True)

    return df
```

4.1 Overlap

In this chapter we characterize the performance of the key function, `bioframe.overlap`. We show that the speed depends on:

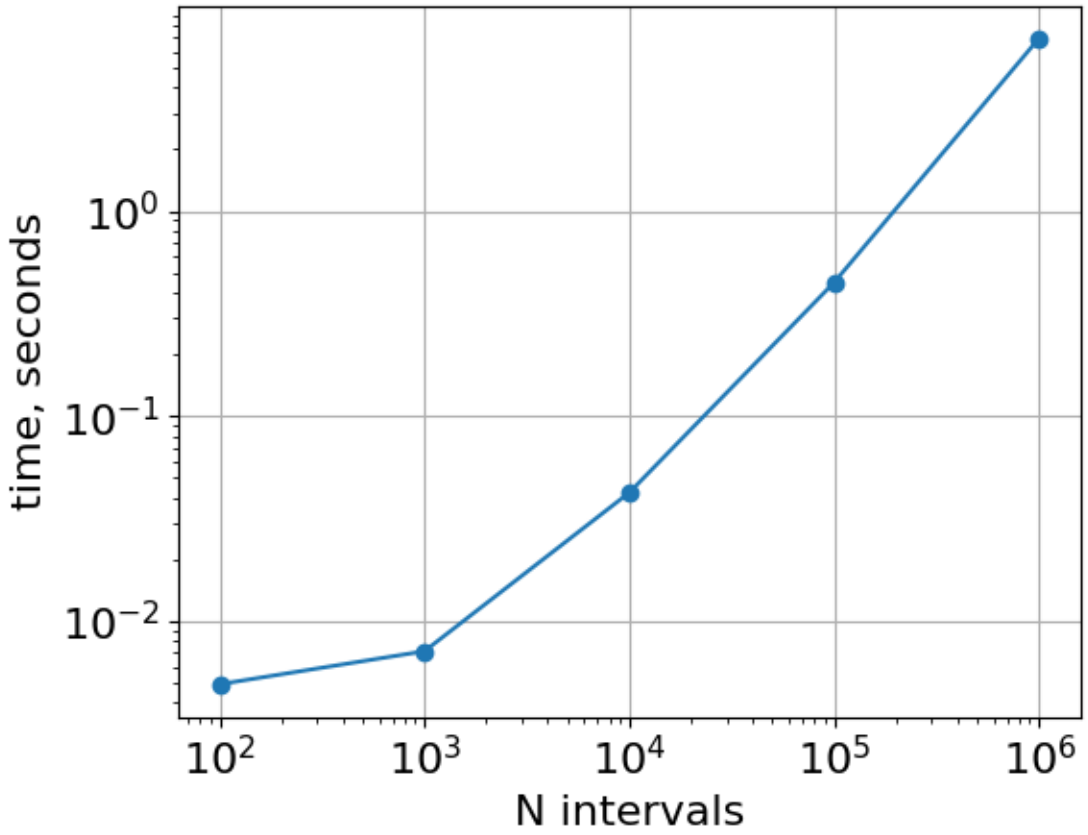
- the number of intervals
- number of intersections (or density of intervals)
- type of overlap (inner, outer, left)
- dtype of chromosomes

4.1.1 vs number of intervals

```
timings = {}
for n in [1e2, 1e3, 1e4, 1e5, 1e6]:
    df = make_random_intervals(n=n, n_chroms=1)
    df2 = make_random_intervals(n=n, n_chroms=1)
    timings[n] = %timeit -o -r 1 bioframe.overlap(df, df2)
```

```
4.92 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
7.13 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
42.3 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
448 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
6.95 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
plt.loglog(
    list(timings.keys()),
    list([r.average for r in timings.values()]),
    "o-",
)
plt.xlabel("N intervals")
plt.ylabel("time, seconds")
plt.gca().set_aspect(1.0)
plt.grid()
```



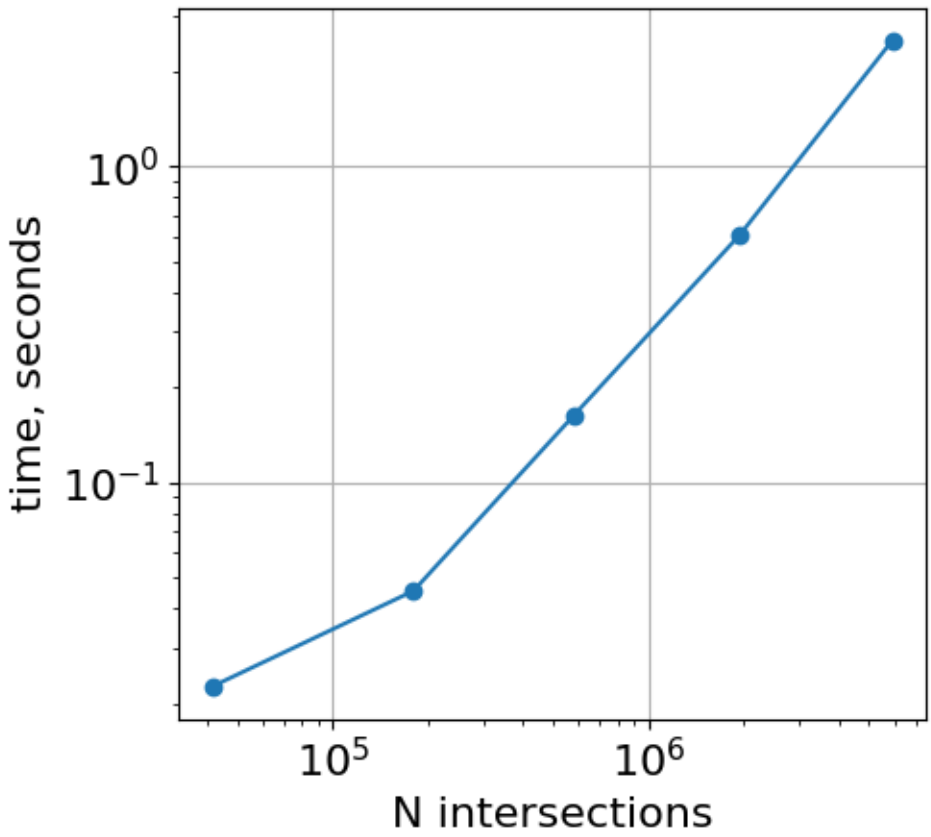
4.1.2 vs total number of intersections

Note that not only the number of intervals, but also the density of intervals determines the performance of overlap.

```
timings = {}
n_intersections = {}
n = 1e4
for avg_interval_len in [3, 1e1, 3e1, 1e2, 3e2]:
    df = make_random_intervals(n=n, n_chroms=1, max_length=avg_interval_len * 2)
    df2 = make_random_intervals(n=n, n_chroms=1, max_length=avg_interval_len * 2)
    timings[avg_interval_len] = %timeit -o -r 1 bioframe.overlap(df, df2)
    n_intersections[avg_interval_len] = bioframe.overlap(df, df2).shape[0]
```

```
22.7 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
45.5 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
163 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
611 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
2.51 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
plt.loglog(
    list(n_intersections.values()),
    list([r.average for r in timings.values()]),
    "o-",
)
plt.xlabel("N intersections")
plt.ylabel("time, seconds")
plt.gca().set_aspect(1.0)
plt.grid()
```



4.1.3 vs number of chromosomes

If we consider a genome of the same length, divided into more chromosomes, the timing is relatively unaffected.

```
timings = {}
n_intersections = {}
n = 1e5
for n_chroms in [1, 3, 10, 30, 100, 300, 1000]:
    df = make_random_intervals(n, n_chroms)
    df2 = make_random_intervals(n, n_chroms)
    timings[n_chroms] = %timeit -o -r 1 bioframe.overlap(df, df2)
    n_intersections[n_chroms] = bioframe.overlap(df, df2).shape[0]
```

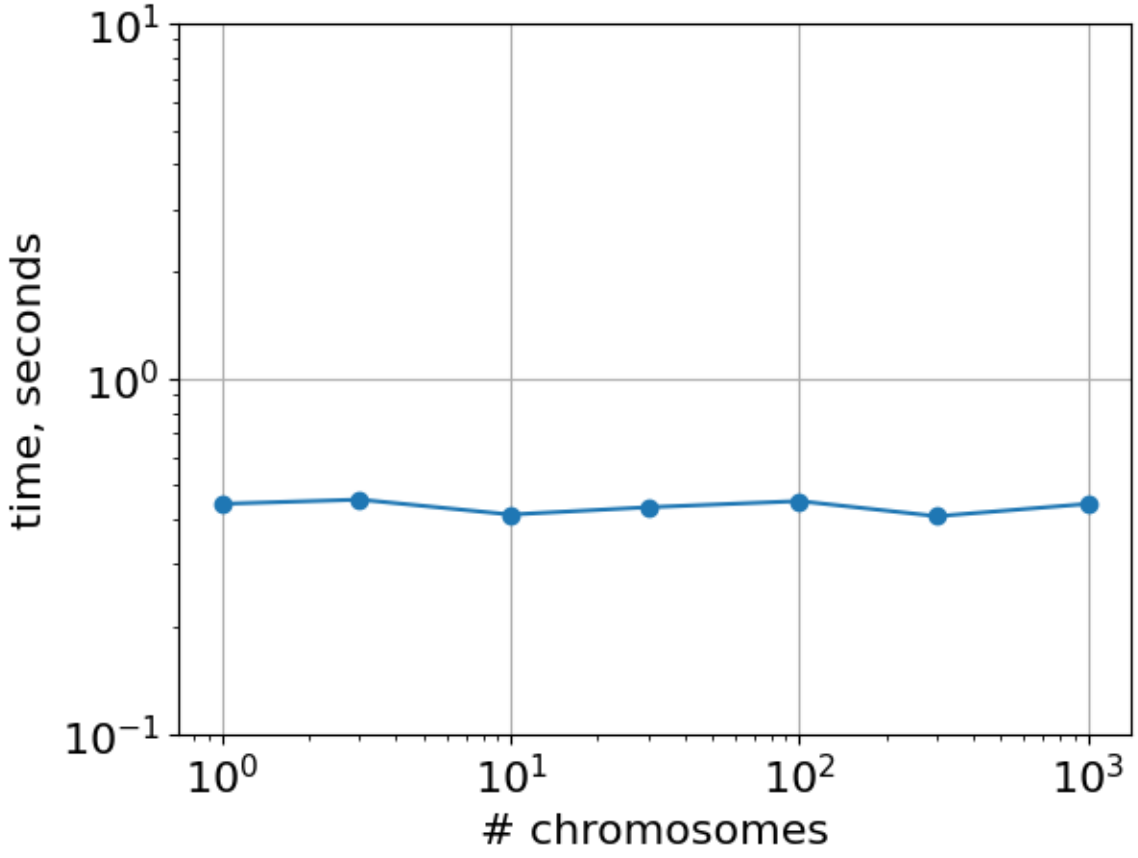
```
443 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
456 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
414 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
434 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
451 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
409 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
443 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Note this test preserves the number of intersections, which is likely why performance remains similar over the considered range.

```
n_intersections
```

```
{1: 810572,
 3: 810871,
 10: 809463,
 30: 815322,
 100: 808166,
 300: 802130,
 1000: 787235}
```

```
plt.loglog(
    list(timings.keys()),
    list([r.average for r in timings.values()]),
    "o-",
)
plt.ylim([1e-1, 10])
plt.xlabel("# chromosomes")
plt.ylabel("time, seconds")
# plt.gca().set_aspect(1.0)
plt.grid()
```



4.1.4 vs other parameters: join type, sorted or categorical inputs

Note that default for overlap: `how='left', keep_order=True`, and the returned dataframe is sorted after the overlaps have been ascertained. Also note that `keep_order=True` is only a valid argument for `how='left'` as the order is not well-defined for inner or outer overlaps.

```
df = make_random_intervals()
df2 = make_random_intervals()
%timeit -r 1 bioframe.overlap(df, df2)
%timeit -r 1 bioframe.overlap(df, df2, how='left', keep_order=False)
```

```
418 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
274 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
df = make_random_intervals()
df2 = make_random_intervals()

%timeit -r 1 bioframe.overlap(df, df2, how='outer')
%timeit -r 1 bioframe.overlap(df, df2, how='inner')
%timeit -r 1 bioframe.overlap(df, df2, how='left', keep_order=False)
```

```
329 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
151 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
247 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Note below that detection of overlaps takes a relatively small fraction of the execution time. The majority of the time the user-facing function spends on formatting the output table.

```
df = make_random_intervals()
df2 = make_random_intervals()

%timeit -r 1 bioframe.overlap(df, df2)
%timeit -r 1 bioframe.overlap(df, df2, how='inner')
%timeit -r 1 bioframe.ops._overlap_intidxs(df, df2)
%timeit -r 1 bioframe.ops._overlap_intidxs(df, df2, how='inner')
```

```
449 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
148 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
61 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
62.4 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
```

Note that sorting inputs provides a moderate speedup, as well as storing chromosomes as categoricals

```
print("Default inputs (outer/inner joins):")
df = make_random_intervals()
df2 = make_random_intervals()

%timeit -r 1 bioframe.overlap(df, df2)
%timeit -r 1 bioframe.overlap(df, df2, how='inner')

print("Sorted inputs (outer/inner joins):")
df_sorted = make_random_intervals(sort=True)
df2_sorted = make_random_intervals(sort=True)
```

(continues on next page)

(continued from previous page)

```
%timeit -r 1 bioframe.overlap(df_sorted, df2_sorted)
%timeit -r 1 bioframe.overlap(df_sorted, df2_sorted, how='inner')

print("Categorical chromosomes (outer/inner joins):")
df_cat = make_random_intervals(categorical_chroms=True)
df2_cat = make_random_intervals(categorical_chroms=True)

%timeit -r 1 bioframe.overlap(df_cat, df2_cat)
%timeit -r 1 bioframe.overlap(df_cat, df2_cat, how='inner')
```

```
Default inputs (outer/inner joins):
440 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
149 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
Sorted inputs (outer/inner joins):
331 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
137 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
Categorical chromosomes (outer/inner joins):
333 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
90 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
```

4.2 Vs Pyranges (and, optionally, pybedtools)

4.2.1 Default arguments

The core intersection function of PyRanges is faster, since PyRanges object splits intervals by chromosomes at the object construction stage

```
def df2pr(df):
    return pyranges.PyRanges(
        chromosomes=df.chrom,
        starts=df.start,
        ends=df.end,
    )
```

```
timings_bf = {}
timings_pr = {}
for n in [1e2, 1e3, 1e4, 1e5, 1e6, 3e6]:
    df = make_random_intervals(n=n, n_chroms=1)
    df2 = make_random_intervals(n=n, n_chroms=1)
    pr = df2pr(df)
    pr2 = df2pr(df2)
    timings_bf[n] = %timeit -o -r 1 bioframe.overlap(df, df2,how='inner')
    timings_pr[n] = %timeit -o -r 1 pr.join(pr2)
```

```
2.36 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
1.49 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1,000 loops each)
2.94 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
1.9 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1,000 loops each)
```

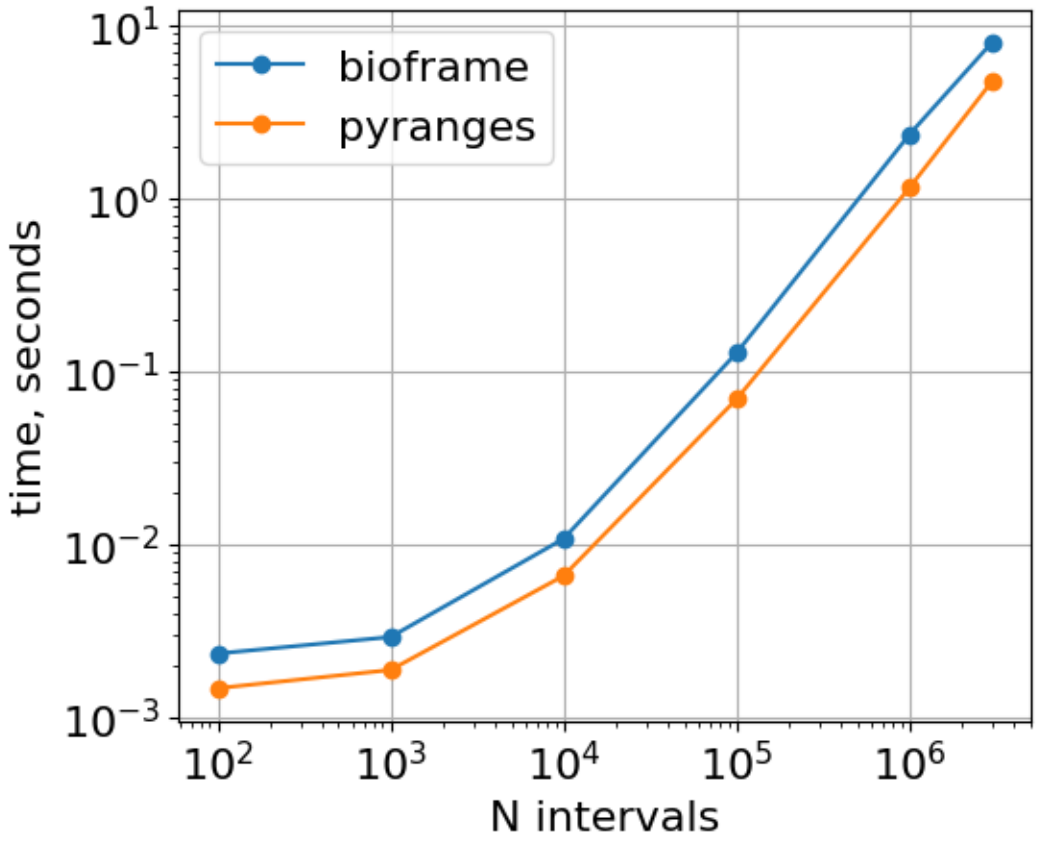
(continues on next page)

(continued from previous page)

```
10.8 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
6.63 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
128 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
69.4 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
2.35 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
1.16 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
7.97 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
4.75 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
plt.loglog(
    list(timings_bf.keys()),
    list([r.average for r in timings_bf.values()]),
    "o-",
    label="bioframe",
)
plt.loglog(
    list(timings_pr.keys()),
    list([r.average for r in timings_pr.values()]),
    "o-",
    label="pyranges",
)

plt.gca().set(
    xlabel="N intervals",
    ylabel="time, seconds",
    aspect=1.0,
    xticks=10 ** np.arange(2, 6.1),
)
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fe3556e94b0>
```



4.2.2 With roundtrips to dataframes

Note that pyranges performs useful calculations at the stage of creating a PyRanges object. Thus a direct comparison for one-off operations on pandas DataFrames between bioframe and pyranges should take this step into account. This roundrip is handled by `pyranges_intersect_dfs` below.

```
def pyranges_intersect_dfs(df, df2):
    return df2pr(df).intersect(df2pr(df2)).as_df()


if include_pybedtools:

    def pybedtools_intersect_dfs(bed1, bed2):
        return bed1.intersect(bed2).to_dataframe()
```

```
timings_bf = {}
timings_pr = {}
if include_pybedtools:
    timings_pb = {}

for n in [1e2, 1e3, 1e4, 1e5, 1e6, 3e6]:
    df = make_random_intervals(n=n, n_chroms=1)
    df2 = make_random_intervals(n=n, n_chroms=1)
    timings_bf[n] = %timeit -o -r 1 bioframe.overlap(df, df2, how='inner')
```

(continues on next page)

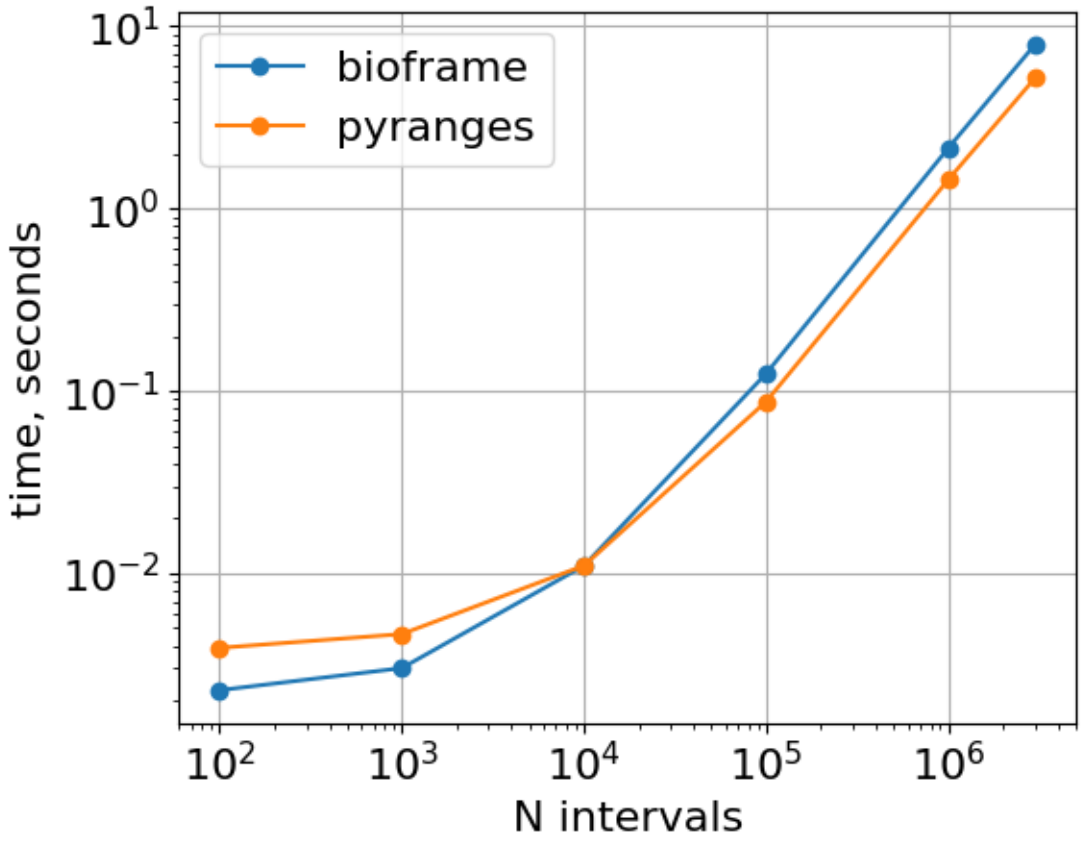
(continued from previous page)

```
    timings_pr[n] = %timeit -o -r 1 pyranges_intersect_dfs(df, df2)
    if include_pybedtools:
        bed1 = pybedtools.BedTool.from_dataframe(df)
        bed2 = pybedtools.BedTool.from_dataframe(df2)
        timings_pb[n] = %timeit -o -r 1 pybedtools_intersect_dfs(bed1, bed2)
```

```
2.28 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
3.9 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
3.02 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
4.64 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
11 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
11 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
125 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
87.4 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
2.15 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
1.44 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
7.98 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
5.23 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

```
plt.loglog(
    list(timings_bf.keys()),
    list([r.average for r in timings_bf.values()]),
    "o-",
    label="bioframe",
)
plt.loglog(
    list(timings_pr.keys()),
    list([r.average for r in timings_pr.values()]),
    "o-",
    label="pyranges",
)
if include_pybedtools:
    plt.loglog(
        list(timings_pb.keys()),
        list([r.average for r in timings_pb.values()]),
        "o-",
        label="pybedtools",
    )
plt.gca().set(xlabel="N intervals", ylabel="time, seconds", aspect=1.0)
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fe355f4d420>
```



4.2.3 Memory usage

```
import time

from memory_profiler import memory_usage


def sleep_before_after(func, sleep_sec=0.5):
    """
    Wrapper that allows to report background interpreter's memory consumption
    for the first 5 time intervals (if increment is 0.1 abd sleep_sec=0.5):
    https://github.com/pythonprofilers/memory_profiler#api
    """

    def _f(*args, **kwargs):
        time.sleep(sleep_sec)
        func(*args, **kwargs)
        time.sleep(sleep_sec)

    return _f


mem_usage_bf = {}
mem_usage_pr = {}
```

(continues on next page)

(continued from previous page)

```
if include_pybedtools:
    mem_usage_pb = {}

for n in [1e2, 1e3, 1e4, 1e5, 1e6, 3e6]:
    df = make_random_intervals(n=n, n_chroms=1)
    df2 = make_random_intervals(n=n, n_chroms=1)
    mem_usage_bf[n] = memory_usage(
        (sleep_before_after(bioframe.overlap), (df, df2), dict(how="inner")),
        backend="psutil_pss",
        include_children=True,
        interval=0.1,
    )
    mem_usage_pr[n] = memory_usage(
        (sleep_before_after(pyranges_intersect_dfs), (df, df2), dict()),
        backend="psutil_pss",
        include_children=True,
        interval=0.1,
    )
    if include_pybedtools:
        bed1 = pybedtools.BedTool.from_dataframe(df)
        bed2 = pybedtools.BedTool.from_dataframe(df2)
        mem_usage_pb[n] = memory_usage(
            (sleep_before_after(pybedtools_intersect_dfs), (bed1, bed2), dict()),
            backend="psutil_pss",
            include_children=True,
            interval=0.1,
        )
```

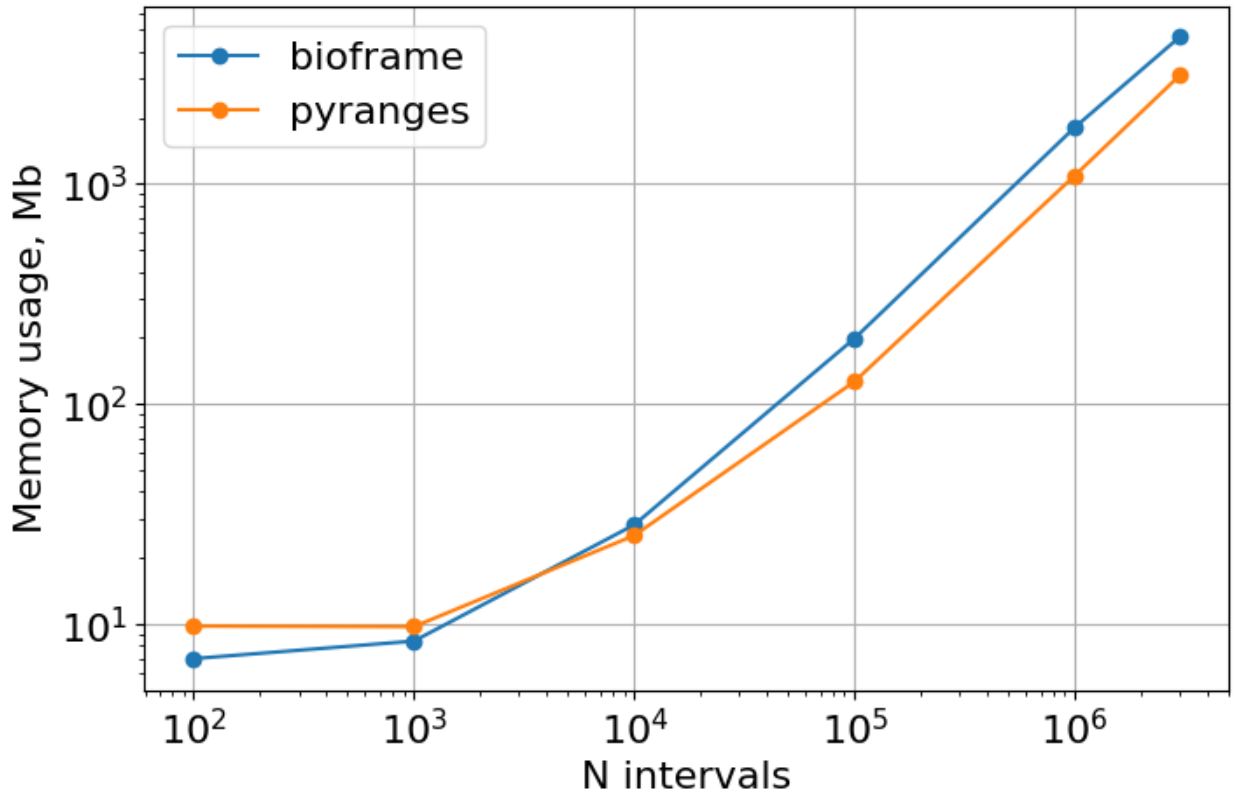
```
# Note that r[4] is the background memory usage of Python interpreter,
# and max(r) is the maximum memory usage (that must be from the
# bioframe/pyranges functions)
plt.figure(figsize=(8, 6))
plt.loglog(
    list(mem_usage_bf.keys()),
    list([max(r) - r[4] for r in mem_usage_bf.values()]),
    "o-",
    label="bioframe",
)

plt.loglog(
    list(mem_usage_pr.keys()),
    list([max(r) - r[4] for r in mem_usage_pr.values()]),
    "o-",
    label="pyranges",
)

if include_pybedtools:
    plt.loglog(
        list(mem_usage_pb.keys()),
        list([max(r) - r[4] for r in mem_usage_pb.values()]),
        "o-",
        label="pybedtools",
```

(continues on next page)

(continued from previous page)

```
    )

plt.gca().set(xlabel="N intervals", ylabel="Memory usage, Mb", aspect=1.0)
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fe38f4014b0>
```




```
print("Bioframe dtypes:")
display(df.dtypes)
print()

print("Pyranges dtypes:")
display(df2pr(df).dtypes)

if include_pybedtools:
    print("Pybedtools dtypes:")
    bed1 = pybedtools.BedTool.from_dataframe(df)
    display(bed1.to_dataframe().dtypes)
```

```
Bioframe dtypes:
```

```
chrom    object
start     int64
```

(continues on next page)

(continued from previous page)

```
end       int64
dtype: object
```

```
Pyranges dtypes:
```

```
Chromosome    category
Start            int64
End              int64
dtype: object
```

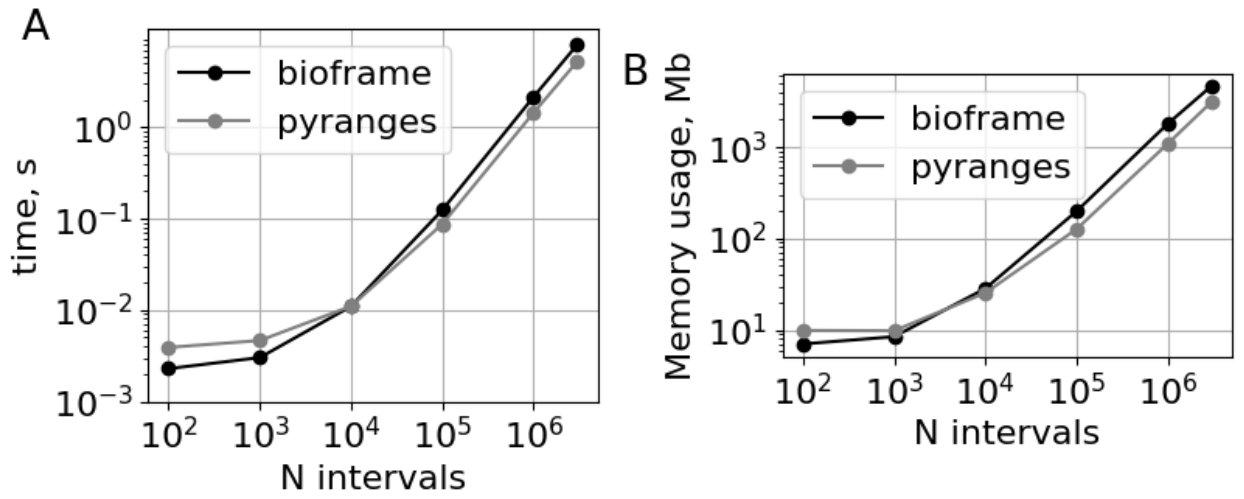
```
### Combined performance figure.

fig, axs = plt.subplot_mosaic("AAA.BBB", figsize=(9.0, 4))

plt.sca(axs["A"])

plt.text(
    -0.25,
    1.0,
    "A",
    horizontalalignment="center",
    verticalalignment="center",
    transform=plt.gca().transAxes,
    fontsize=19,
)

plt.loglog(
    list(timings_bf.keys()),
    list([r.average for r in timings_bf.values()]),
    "o-",
    color="k",
    label="bioframe",
)
plt.loglog(
    list(timings_pr.keys()),
    list([r.average for r in timings_pr.values()]),
    "o-",
    color="gray",
    label="pyranges",
)
if include_pybedtools:
    plt.loglog(
        list(timings_pb.keys()),
        list([r.average for r in timings_pb.values()]),
        "o-",
        color="lightgray",
        label="pybedtools",
    )

plt.gca().set(
    xlabel="N intervals",
```

(continues on next page)

(continued from previous page)

```
    ylabel="time, s",
    aspect=1.0,
    xticks=10 ** np.arange(2, 6.1),
    yticks=10 ** np.arange(-3, 0.1),
)

plt.grid()
plt.legend()

plt.sca(axs["B"])
plt.text(
    -0.33,
    1.0,
    "B",
    horizontalalignment="center",
    verticalalignment="center",
    transform=plt.gca().transAxes,
    fontsize=19,
)
plt.loglog(
    list(mem_usage_bf.keys()),
    list([max(r) - r[4] for r in mem_usage_bf.values()]),
    "o-",
    color="k",
    label="bioframe",
)

plt.loglog(
    list(mem_usage_pr.keys()),
    list([max(r) - r[4] for r in mem_usage_pr.values()]),
    "o-",
    color="gray",
    label="pyranges",
)
if include_pybedtools:
    plt.loglog(
        list(mem_usage_pb.keys()),
        list([max(r) - r[4] for r in mem_usage_pb.values()]),
        "o-",
        color="lightgray",
        label="pybedtools",
    )

plt.gca().set(
    xlabel="N intervals",
    ylabel="Memory usage, Mb",
    aspect=1.0,
    xticks=10 ** np.arange(2, 6.1),
)

plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fe3548d5120>
```



4.2.4 Slicing

```
timings_slicing_bf = {}
timings_slicing_pr = {}


for n in [1e2, 1e3, 1e4, 1e5, 1e6, 3e6]:
    df = make_random_intervals(n=n, n_chroms=1)
    timings_slicing_bf[n] = %timeit -o -r 1 bioframe.select(df, ('chr1', n//2, n//4*3))
    pr = df2pr(df)
    timings_slicing_pr[n] = %timeit -o -r 1 pr['chr1', n//2:n//4*3]
```

```
334 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1,000 loops each)
468 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1,000 loops each)
346 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1,000 loops each)
593 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1,000 loops each)
668 µs ± 0 ns per loop (mean ± std. dev. of 1 run, 1,000 loops each)
1.92 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1,000 loops each)
3.54 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
18.1 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 100 loops each)
40.1 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
222 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
118 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 10 loops each)
804 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

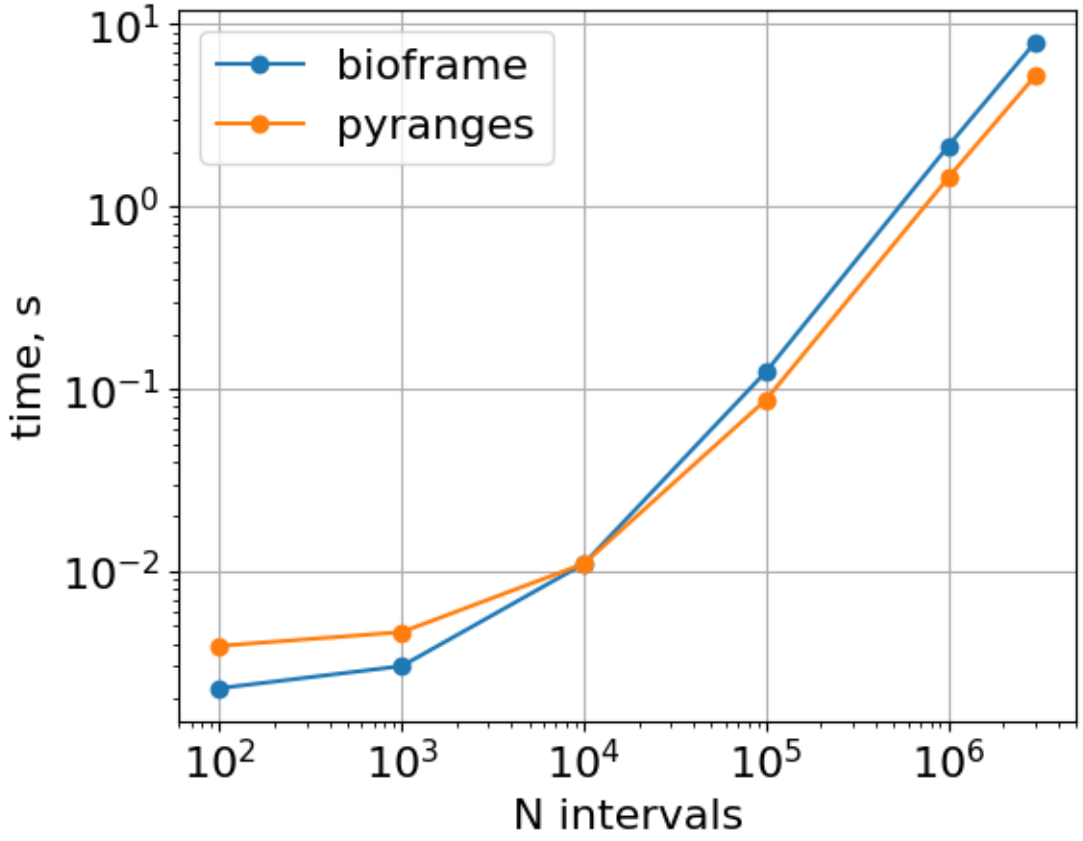
```
plt.loglog(
    list(timings_slicing_bf.keys()),
    list([r.average for r in timings_bf.values()]),
    "o-",
    label="bioframe",
)
```

(continues on next page)

(continued from previous page)

```
plt.loglog(
    list(timings_slicing_pr.keys()),
    list([r.average for r in timings_pr.values()]),
    "o-",
    label="pyranges",
)
plt.gca().set(xlabel="N intervals", ylabel="time, s", aspect=1.0)
plt.grid()
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fe3174e49d0>
```



CHAPTER

FIVE

HOW DO I

5.1 Obtain overlapping intervals with matching strandedness?

Use overlap with the on argument:

```
df = bf.overlap(df1, df2, on=['strand'])
```

5.2 Obtain overlapping intervals with opposite strandedness?

Overlap then filter pairs of opposite strandedness:

```
df = bf.overlap(df1, df2)
df = df.loc[df["strand"]!=df["strand_"]]
```

5.3 Obtain intervals that exceed 50% coverage by another set of intervals?

Coverage, then filter pairs by fractional coverage:

```
df = bf.coverage(df1, df2)
df = df[ ( df["coverage"] / (df["end"]-df["start"]) ) >=0.50]
```

5.4 Shift all intervals on the positive strand by 10bp?

Use pandas indexing:

```
df.loc[df.strand=="+",["start", "end"]] += 10
```

5.5 Obtain intervals overlapped by at least 2 intervals from another set?

Count overlaps, then filter:

```
df = bf.count_overlaps(df1, df2)
df = df[ df["count"] >= 2]
```

5.6 Find strand-specific downstream genomic features?

Use closest after filtering by strand, and passing the `ignore_upsream=True` argument.

```
bioframe.closest(df1.loc[df1['strand']=='+'], df2, ignore_upstream=True)
```

For gener, the upstream/downstream direction might be defined by the direction of transcription. Use `direction_col='strand'` to set up the direction:

```
bioframe.closest(df1, df2, ignore_upstream=True, direction_col='strand')
```

5.7 Drop non-autosomes from a bedframe?

Use pandas DataFrame.isin(values):

```
df[ ~df.chrom.isin(['chrX','chrY'])]
```

CHAPTER

SIX

DEFINITIONS

Interval:

- An *interval* is a tuple of integers (start, end) with start <= end.
- Coordinates are assumed to be 0-based and intervals half-open (1-based ends) i.e. [start, end).
- An interval has a *length* equal to (end - start).
- A special case where start and end are the same, i.e. [X, X), is interpreted as a *point* (aka an *empty interval*, i.e. an edge between 1-bp bins). A point has zero length.
- Negative coordinates are permissible for both ends of an interval.

Properties of a pair of intervals:

- Two intervals can either *overlap*, or not. The overlap length = max(0, min(end1, end2) - max(start1, start2)). Empty intervals can have overlap length = 0.
- When two intervals overlap, the shorter of the two intervals is said to be *contained* in the longer one if the length of their overlap equals the length of the shorter interval. This property is often referred to as nestedness, but we use the term "contained" as it is less ambiguous when describing the relationship of sets of intervals to one interval.
- If two intervals do not overlap, they have a *distance* = max(0, max(start1, start2) - min(end1, end2)).
- If two intervals have overlap=0 and distance=0, they are said to be *abutting*.

Scaffold:

- A chromosome, contig or, more generally, a *scaffold* is an interval defined by a unique string and has a length>=0, with start=0 and end=length, implicitly defining an interval [0, length).

Genome assembly:

- The complete set of scaffolds associated with a genome is called an *assembly* (e.g. defined by the reference sequence from NCBI, etc.).

Genomic interval:

- A *genomic interval* is an interval with an associated scaffold, or chromosome, defined by a string, i.e. a triple (chrom, start, end).
- Genomic intervals on different scaffolds never overlap and do not have a defined distance.
- Genomic intervals can extend beyond their associated scaffold (e.g. with negative values or values greater than the scaffold length), as this can be useful in downstream applications. If they do, they are not contained by their associated scaffold.
- A *base-pair* is a special case of a genomic interval with length=1, i.e. (chrom, start, start+1)

- *strand* is an (optional) property of a genomic interval which specifies an interval's orientation on its scaffold. Note start and end are still defined with respect to the scaffold's reference orientation (positive strand), even if the interval lies on the negative strand. Intervals on different strands can either be allowed to overlap or not.

View (i.e. a set of Genomic Regions):

- A genomic *view* is an ordered set of non-overlapping genomic intervals each having a unique name defined by a string. Individual named intervals in a view are *regions*, defined by a quadruple, e.g. (chrom, start, end, name).
- A view thus specifies a unified 1D coordinate system, i.e. a projection of multiple genomic regions onto a single axis.
- We define views separately from the scaffolds that make up a genome assembly, as a set of more constrained and ordered genomic regions are often useful for downstream analysis and visualization.
- An assembly is a special case of a view, where the individual regions correspond to the assembly's entire scaffolds.

Associating genomic intervals with views

- Similarly to how genomic intervals are associated with a scaffold, they can also be associated with a region from a view with an additional string, making a quadruple (chrom, start, end, view_region). This string must be *cataloged* in the view, i.e. it must match the name of a region in the view. Typically the interval would be contained in its associated view region, or, at the minimum, have a greater overlap with that region than other view regions.
- If each interval in a set is contained in their associated view region, the set is *contained* in the view.
- A set of intervals *covers* a view if each region in the view is contained by the union of its associated intervals. Conversely, if a set does not cover all of view regions, the interval set will have *gaps* relative to that view (stretches of bases not covered by an interval).

Properties of sets of genomic intervals:

- A set of genomic intervals may have overlaps or not. If it does not, it is said to be *overlap-free*.
- A set of genomic intervals is *tiling* if it: (i) covers the associated view, (ii) is contained in that view, and (iii) is overlap-free. Equivalently, a tiling set of intervals (a) has an initial interval that begins at the start of each region and (b) a final interval that terminates at the end of each region, and (c) every base pair is associated with a unique interval.

CHAPTER

SEVEN

SPECIFICATIONS

BedFrame (i.e. genomic intervals stored in a pandas dataframe):

- In a BedFrame, three required columns specify the set of genomic intervals (default column names = ('chrom', 'start', 'end')).
- Other reserved but not required column names: ('strand', 'name', 'view_region').
  - entries in column 'name' are expected to be unique
  - 'view_region' is expected to point to an associated region in a view with a matching name
  - 'strand' is expected to be encoded with strings ('+', '-', '.').
- Additional columns are allowed: 'zodiac_sign', 'soundcloud', 'twitter_name', etc.
- Repeated intervals are allowed.
- The native pandas DataFrame index is not intended to be used as an immutable lookup table for genomic intervals in BedFrame. This is because many common genomic interval operations change the number of intervals stored in a BedFrame.
- Two useful sorting schemes for BedFrames are:
  - scaffold-sorted: on (chrom, start, end), where chrom is sorted lexicographically.
  - view-sorted: on (view_region, start, end) where view_region is sorted by order in the view.
- Null values are allowed, but only as pd.NA (using np.nan is discouraged as it results in unwanted type re-casting).
- Note if no 'view_region' is assigned to a genomic interval, then 'chrom' implicitly defines an associated region
- Note the BedFrame specification is a natural extension of the BED format (https://samtools.github.io/hts-specs/BEDv1.pdf) for pandas DataFrames.

ViewFrames (a genomic view stored in a pandas dataframe)

- BedFrame where:
  - intervals are non-overlapping
  - "name" column is mandatory and contains a set of unique strings.
- Note that a ViewFrame can potentially be indexed by the name column to serve as a lookup table. This functionality is currently not implemented, because within the current Pandas implementation indexing by a column removes the column from the table.
- Note that views can be defined by:
  - dictionary of string:ints (start=0 assumed) or string:tuples (start,end)

- pandas series of chromsizes (start=0, name=chrom)

CHAPTER

EIGHT

BIOFRAME FOR BEDTOOLS USERS

Bioframe is built around the analysis of genomic intervals as a pandas DataFrame in memory, rather than working with tab-delimited text files saved on disk.

Bioframe supports reading a number of standard genomics text file formats via `read_table`, including BED files (see schemas), which will load them as pandas DataFrames, a complete list of helper functions is *available here*.

Any DataFrame object with `'chrom'`, `'start'`, and `'end'` columns will support the genomic *interval operations in bioframe*. The names of these columns can also be customized via the `cols=` arguments in bioframe functions.

For example, with gtf files, you do not need to turn them into bed files, you can directly read them into pandas (with e.g. gtfparse). For gtfs, it is often convenient to rename the `'seqname'` column to `'chrom'`, the default column name used in bioframe.

Finally, if needed, bioframe provides a convenience function to write dataframes to a standard BED file using `to_bed`.

8.1 bedtools intersect

8.1.1 Select unique entries from the first bed overlapping the second bed -u

```
bedtools intersect -u -a A.bed -b B.bed > out.bed
```

```
overlap = bf.overlap(A, B, how='inner', suffixes=('_1','_2'), return_index=True)
out = A.loc[overlap['index_1'].unique()]
```

8.1.2 Report the number of hits in B -c

Reports 0 for A entries that have no overlap with B.

```
bedtools intersect -c -a A.bed -b B.bed > out.bed
```

```
out = bf.count_overlaps(A, B)
```

8.1.3 Return entries from both beds for each overlap `-wa -wb`

```
bedtools intersect -wa -wb -a A.bed -b B.bed > out.bed
```

```
out = bf.overlap(A, B, how='inner')
```

Note: This is called an "inner join", and is analogous to an inner pandas join or merge. The default column suffixes in the output dataframe are `''` (nothing) for A's columns and `'_'` for B's columns.

8.1.4 Include all entries from the first bed, even if no overlap `-loj`

```
bedtools intersect -wa -wb -loj -a A.bed -b B.bed > out.bed
```

```
out = bf.overlap(A, B, how='left')
```

Note: This is called a "left-outer join".

8.1.5 Select entries from the first bed for each overlap `-wa`

```
bedtools intersect -wa -a A.bed -b B.bed > out.bed
```

```
overlap = bf.overlap(A, B, how='inner', suffixes=('_1','_2'), return_index=True)
out = A.loc[overlap['index_1']]

# Alternatively
out = bf.overlap(A, B, how='inner')[A.columns]
```

> **Note:** This gives one row per overlap and can contain duplicates. The output dataframe of the former method will use the same pandas index as the input dataframe `A`, while the latter result — the join output — will have an integer range index, like a pandas merge.

8.1.6 Select entries from the second bed for each overlap `-wb`

```
bedtools intersect -wb -a A.bed -b B.bed > out.bed
```

```
overlap = bf.overlap(A, B, how='inner', suffixes=('_1','_2'), return_index=True)
out = B.loc[overlap['index_2']]

# Alternatively
out = bf.overlap(A, B, how='inner', suffixes=('_', ''))[B.columns]
```

> **Note:** This gives one row per overlap and can contain duplicates. The output dataframe of the former method will use the same pandas index as the input dataframe `B`, while the latter result — the join output — will have an integer range index, like a pandas merge.

8.1.7 Intersect multiple beds against A

```
bedtools intersect -wa -a A.bed -b B.bed C.bed D.bed > out.bed
```

```
others = pd.concat([B, C, D])
overlap = bf.overlap(A, others, how='inner', suffixes=('_1','_2'), return_index=True)
out = A.loc[overlap['index_1']]
```

8.1.8 Return everything in A that doesn't overlap with B `-v`

```
bedtools intersect -wa -a A.bed -b B.bed -v > out.bed
```

```
out = bf.setdiff(A, B)
```

Note: We call this a set difference.

8.1.9 Force strandedness `-s`

For intersection

```
bedtools intersect -wa -a A.bed -b B.bed -s > out.bed
```

```
overlap = bf.overlap(A, B, on=['strand'], suffixes=('_1','_2'), return_index=True, how=
↪'inner')
out = A.loc[overlap['index_1']]
```

For non-intersection `-v`

```
bedtools intersect -wa -a A.bed -b B.bed -v -s > out.bed
```

```
out = bf.setdiff(A, B, on=['strand'])
```

8.1.10 Minimum overlap as a fraction of A `-f`

We want to keep rows of A that are covered at least 70% by elements from B

```
bedtools intersect -wa -a A.bed -b B.bed -f 0.7 > out.bed
```

```
cov = bf.coverage(A, B)
out = A.loc[cov['coverage'] / (cov['end'] - cov['start']) ) >= 0.70]

# Alternatively
out = bf.coverage(A, B).query('coverage / (end - start) >= 0.7')[A.columns]
```

CHAPTER

NINE

HOW TO: ASSIGN TF MOTIFS TO CHIP-SEQ PEAKS

This tutorial demonstrates one way to assign CTCF motifs to CTCF ChIP-seq peaks using bioframe.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

import bioframe
```

```
base_dir = "/tmp/bioframe_tutorial_data/"
assembly = "GRCh38"
```

9.1 Load CTCF ChIP-seq peaks for HFF from ENCODE

This approach makes use of the `narrowPeak` schema for bioframe.read_table .

```
ctcf_peaks = bioframe.read_table(
    "https://www.encodeproject.org/files/ENCFF401MQL/@@download/ENCFF401MQL.bed.gz",
    schema="narrowPeak",
)
ctcf_peaks[0:5]
```

```
   chrom      start        end name  score strand       fc  -log10p  -log10q  \
0  chr19   48309541   48309911    .   1000      .  5.04924     -1.0  0.00438
1   chr4  130563716  130564086    .    993      .  5.05052     -1.0  0.00432
2   chr1  200622507  200622877    .    591      .  5.05489     -1.0  0.00400
3   chr5  112848447  112848817    .    869      .  5.05841     -1.0  0.00441
4   chr1  145960616  145960986    .    575      .  5.05955     -1.0  0.00439

   relSummit
0        185
1        185
2        185
3        185
4        185
```

9.2 Get CTCF motifs from JASPAR

```
### CTCF motif: http://jaspar.genereg.net/matrix/MA0139.1/
jaspar_url = "http://expdata.cmmt.ubc.ca/JASPAR/downloads/UCSC_tracks/2022/hg38/"
jaspar_motif_file = "MA0139.1.tsv.gz"
ctcf_motifs = bioframe.read_table(
    jaspar_url + jaspar_motif_file, schema="jaspar", skiprows=1
)
ctcf_motifs[0:4]
```

	chrom	start	end	name	score	pval	strand
0	chr1	11163	11182	CTCF	811	406	-
1	chr1	11222	11241	CTCF	959	804	-
2	chr1	11280	11299	CTCF	939	728	-
3	chr1	11339	11358	CTCF	837	455	-

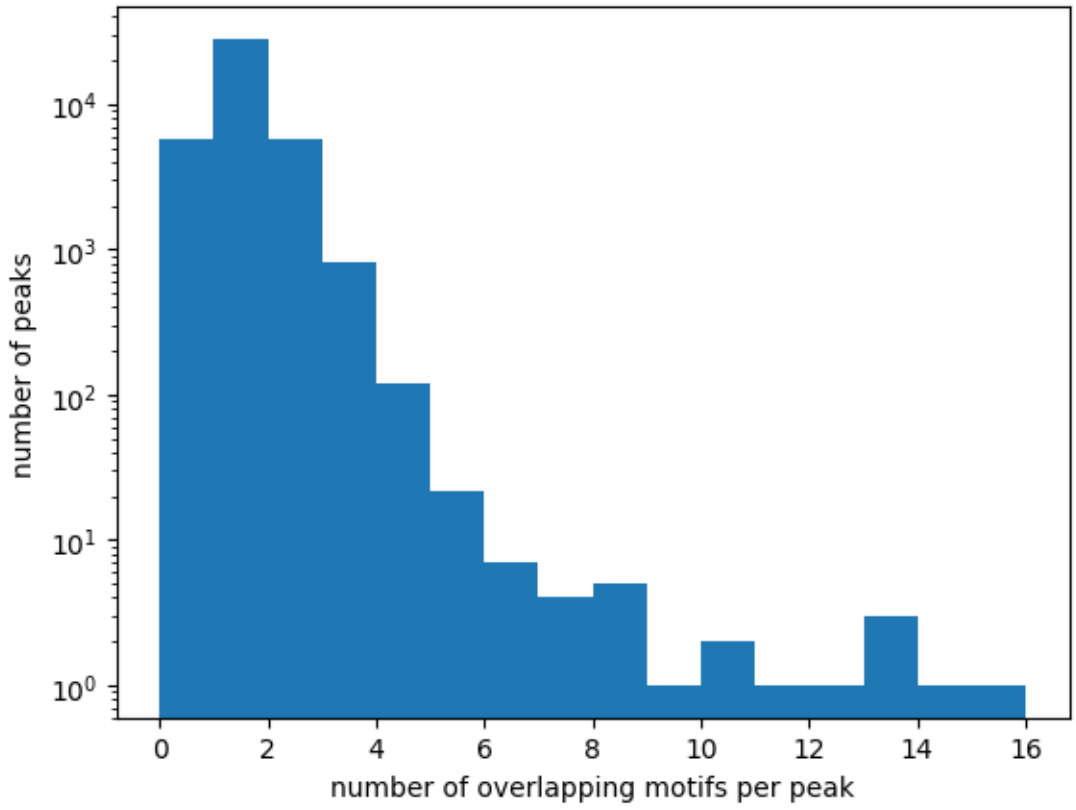
9.3 Overlap peaks & motifs

```
df_peaks_motifs = bioframe.overlap(
    ctcf_peaks, ctcf_motifs, suffixes=("_1", "_2"), return_index=True
)
```

There are often multiple motifs overlapping one ChIP-seq peak, and a substantial number of peaks without motifs:

```
# note that counting motifs per peak can also be handled directly with
# bioframe.count_overlaps but since we re-use df_peaks_motifs below we
# instead use the pandas operations directly
motifs_per_peak = df_peaks_motifs.groupby(["index_1"])["index_2"].count().values

plt.hist(motifs_per_peak, np.arange(0, np.max(motifs_per_peak)))
plt.xlabel("number of overlapping motifs per peak")
plt.ylabel("number of peaks")
plt.semilogy()

print(
    f"fraction of peaks without motifs "
    f"{np.round(np.sum(motifs_per_peak==0)/len(motifs_per_peak), 2)}"
)
```

```
fraction of peaks without motifs 0.14
```



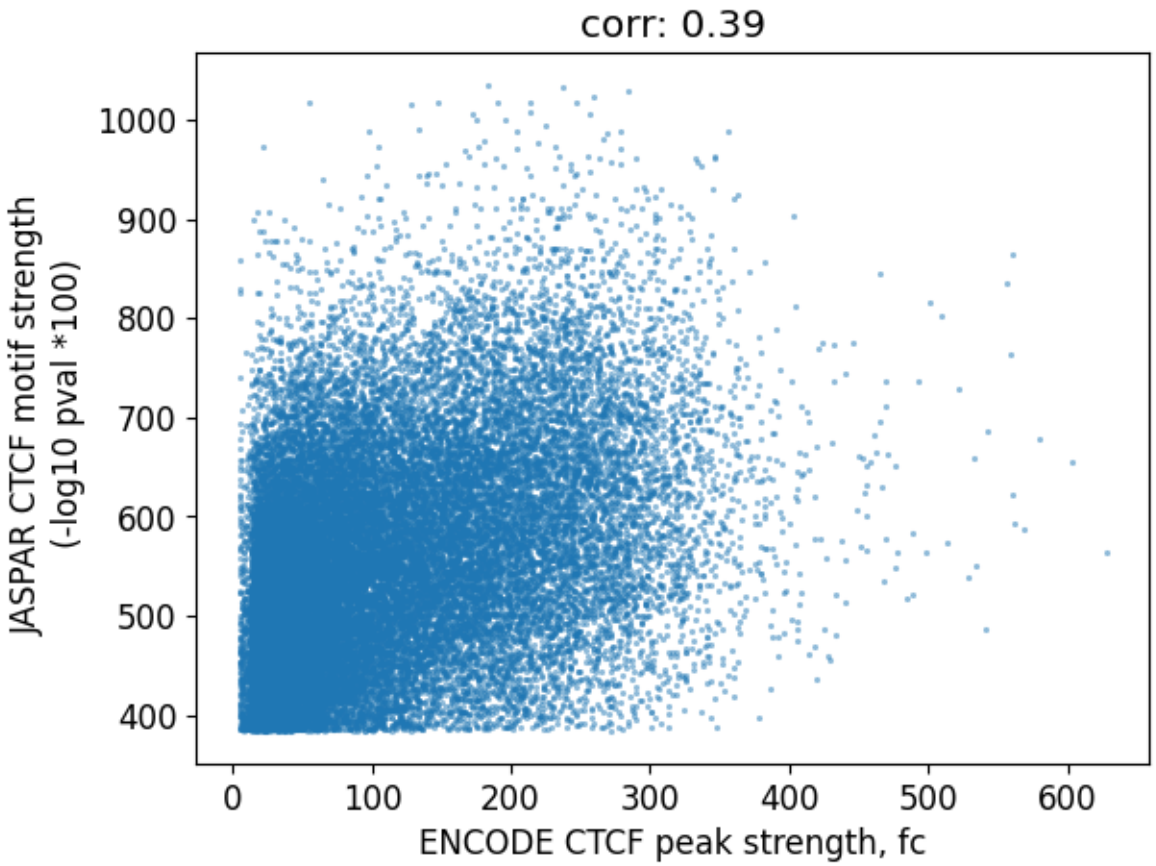
9.3.1 Assign the strongest motif to each peak

```
# since idxmax does not currently take NA, fill with -1
df_peaks_motifs["pval_2"] = df_peaks_motifs["pval_2"].fillna(-1)
idxmax_peaks_motifs = (
    df_peaks_motifs.groupby(["chrom_1", "start_1", "end_1"])["pval_2"].idxmax().values
)
df_peaks_maxmotif = df_peaks_motifs.loc[idxmax_peaks_motifs]
df_peaks_maxmotif["pval_2"].replace(-1, np.nan, inplace=True)
```

```
/tmp/ipykernel_750/2373793568.py:7: FutureWarning: A value is trying to be set on a copy␣
↪of a DataFrame or Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the␣
↪intermediate object on which we are setting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method(
↪{col: value}, inplace=True)' or df[col] = df[col].method(value) instead, to perform␣
↪the operation inplace on the original object.


  df_peaks_maxmotif["pval_2"].replace(-1, np.nan, inplace=True)
```

stronger peaks tend to have stronger motifs:

```
plt.rcParams["font.size"] = 12
df_peaks_maxmotif["fc_1"] = df_peaks_maxmotif["fc_1"].values.astype("float")
plt.scatter(
    df_peaks_maxmotif["fc_1"].values,
    df_peaks_maxmotif["pval_2"].values,
    5,
    alpha=0.5,
    lw=0,
)
plt.xlabel("ENCODE CTCF peak strength, fc")
plt.ylabel("JASPAR CTCF motif strength \n (-log10 pval *100)")
plt.title(
    "corr: "
    + str(np.round(df_peaks_maxmotif["fc_1"].corr(df_peaks_maxmotif["pval_2"]), 2))
);
```



We can also ask the reverse question: how many motifs overlap a ChIP-seq peak?

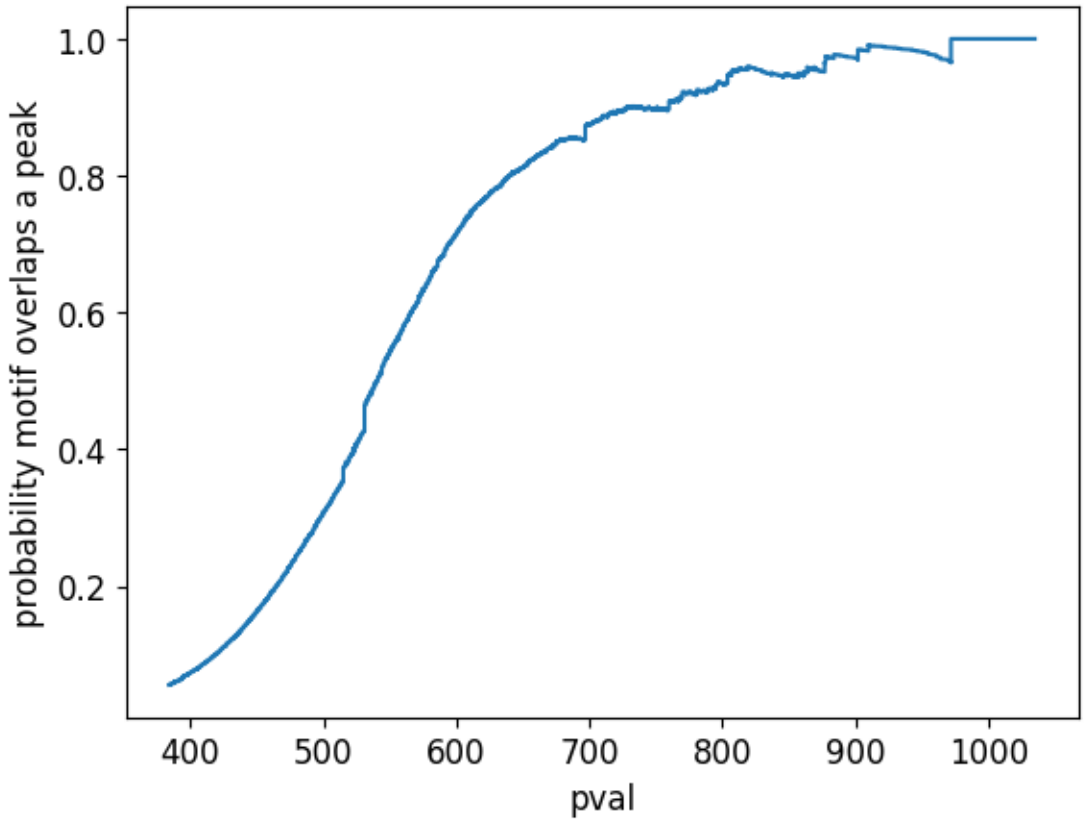
```
df_motifs_peaks = bioframe.overlap(
    ctcf_motifs, ctcf_peaks, how="left", suffixes=("_1", "_2")
)
```

```
m = df_motifs_peaks.sort_values("pval_1")
plt.plot(
```

(continues on next page)

(continued from previous page)

```
    m["pval_1"].values[::-1],
    np.cumsum(pd.isnull(m["chrom_2"].values[::-1]) == 0) / np.arange(1, len(m) + 1),
)
plt.xlabel("pval")
plt.ylabel("probability motif overlaps a peak");
```



9.3.2 filter peaks overlapping blacklisted regions

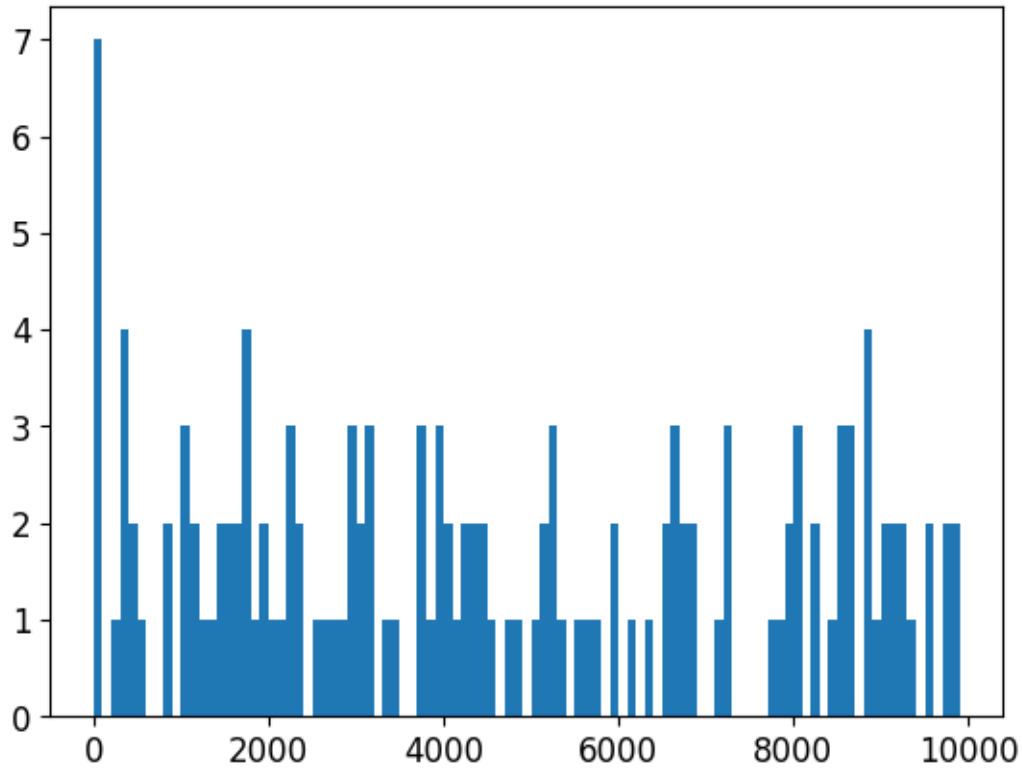
do any of our peaks overlap blacklisted genomic regions?

```
blacklist = bioframe.read_table(
    "https://www.encodeproject.org/files/ENCFF356LFX/@@download/ENCFF356LFX.bed.gz",
    schema="bed3",
)
blacklist[0:3]
```

```
  chrom    start      end
0  chr1   628903   635104
1  chr1  5850087  5850571
2  chr1  8909610  8910014
```

there appears to be a small spike in the number of peaks close to blacklist regions

```
closest_to_blacklist = bioframe.closest(ctcf_peaks, blacklist)
plt.hist(
    closest_to_blacklist["distance"].astype("Float64").astype("float"),
    np.arange(0, 1e4, 100),
);
```



to be safe, let's remove anything +/- 1kb from a blacklisted region

```
# first let's select the columns we want for our final dataframe of peaks
# with motifs
df_peaks_maxmotif = df_peaks_maxmotif[
    [
        "chrom_1",
        "start_1",
        "end_1",
        "fc_1",
        "chrom_2",
        "start_2",
        "end_2",
        "pval_2",
        "strand_2",
    ]
]
#  then rename columns for convenience when subtracting
for i in df_peaks_maxmotif.keys():
    if "_1" in i:
        df_peaks_maxmotif.rename(columns={i: i.split("_")[0]}, inplace=True)
```

(continues on next page)

(continued from previous page)

```
# now subtract, expanding the blacklist by 1kb
df_peaks_maxmotif_clean = bioframe.subtract(
    df_peaks_maxmotif, bioframe.expand(blacklist, 1000)
)
```

there it is! we now have a dataframe containing positions of CTCF ChIP peaks, including the strongest motif underlying that peak, and after conservative filtering for proximity to blacklisted regions

```
df_peaks_maxmotif_clean.iloc[7:15]
```

```
    chrom      start        end       fc chrom_2    start_2      end_2  \
7    chr9  124777413  124777783  5.06479    chr9  124777400  124777419
8    chr1   67701045   67701415  5.06708    None       <NA>       <NA>
9   chr10  119859586  119859956  5.08015   chr10  119859591  119859610
10   chr3   66816327   66816697  5.08233    chr3   66816332   66816351
11  chr16   50248791   50249161  5.08249    None       <NA>       <NA>
12  chr19   41431677   41432047  5.11060   chr19   41431802   41431821
13   chr4  131644839  131645209  5.11204    None       <NA>       <NA>
14   chr2  203239519  203239889  5.11817    None       <NA>       <NA>

    pval_2 strand_2
7    450.0        +
8      NaN     None
9    611.0        -
10   741.0        -
11     NaN     None
12   477.0        +
13     NaN     None
14     NaN     None
```

CHAPTER

TEN

HOW TO: ASSIGN CHIP-SEQ PEAKS TO GENES

This tutorial demonstrates one way to assign CTCF ChIP-seq peaks to the nearest genes using bioframe.

```
import matplotlib.pyplot as plt
import numpy as np

import bioframe
```

```
base_dir = "/tmp/bioframe_tutorial_data/"
assembly = "hg38"
```

10.1 Load chromosome sizes

```
chromsizes = bioframe.fetch_chromsizes(assembly)
chromsizes.tail()
```

```
name
chr21     46709983
chr22     50818468
chrX     156040895
chrY      57227415
chrM         16569
Name: length, dtype: int64
```

```
chromosomes = bioframe.make_viewframe(chromsizes)
```

10.2 Load CTCF ChIP-seq peaks for HFF from ENCODE

This approach makes use of the `narrowPeak` schema for bioframe.read_table .

```
ctcf_peaks = bioframe.read_table(
    "https://www.encodeproject.org/files/ENCFF401MQL/@@download/ENCFF401MQL.bed.gz",
    schema="narrowPeak",
)
ctcf_peaks.head()
```

```
   chrom      start        end name  score strand       fc  -log10p  -log10q  \
0  chr19   48309541   48309911    .   1000      .  5.04924     -1.0  0.00438
1   chr4  130563716  130564086    .    993      .  5.05052     -1.0  0.00432
2   chr1  200622507  200622877    .    591      .  5.05489     -1.0  0.00400
3   chr5  112848447  112848817    .    869      .  5.05841     -1.0  0.00441
4   chr1  145960616  145960986    .    575      .  5.05955     -1.0  0.00439

   relSummit
0        185
1        185
2        185
3        185
4        185
```

```
# Filter for selected chromosomes:
ctcf_peaks = bioframe.overlap(ctcf_peaks, chromosomes).dropna(subset=["name_"])[
    ctcf_peaks.columns
]
```

10.3 Get list of genes from UCSC

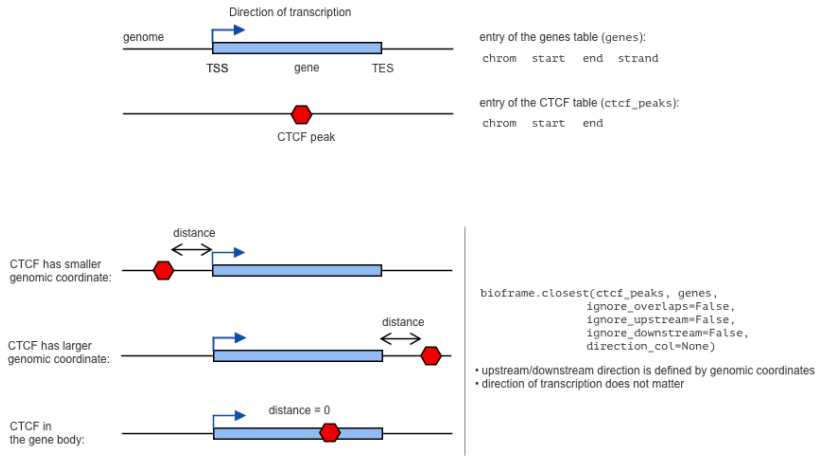
UCSC genes are stored in .gtf format.

```
genes_url = (
    "https://hgdownload.soe.ucsc.edu/goldenpath/hg38/bigZips/genes/hg38.ensGene.gtf.gz"
)
genes = bioframe.read_table(genes_url, schema="gtf").query('feature=="CDS"')

genes.head()

## Note this functions to parse the attributes of the genes:
# import bioframe.sandbox.gtf_io
# genes_attr = bioframe.sandbox.gtf_io.parse_gtf_attributes(genes['attributes'])
```

```
    chrom   source feature   start     end score strand frame  \
47   chr1  ensGene     CDS   69091   70005     .      +     0
112  chr1  ensGene     CDS  182709  182746     .      +     0
114  chr1  ensGene     CDS  183114  183240     .      +     1
116  chr1  ensGene     CDS  183922  184155     .      +     0
122  chr1  ensGene     CDS  185220  185350     .      -     2

                                          attributes
47   gene_id "ENSG00000186092"; transcript_id "ENST...
112  gene_id "ENSG00000279928"; transcript_id "ENST...
114  gene_id "ENSG00000279928"; transcript_id "ENST...
116  gene_id "ENSG00000279928"; transcript_id "ENST...
122  gene_id "ENSG00000279457"; transcript_id "ENST...
```

```
# Filter for selected chromosomes:
genes = bioframe.overlap(genes, chromosomes).dropna(subset=["name_"])[genes.columns]
```

10.4 Assign each peak to the gene

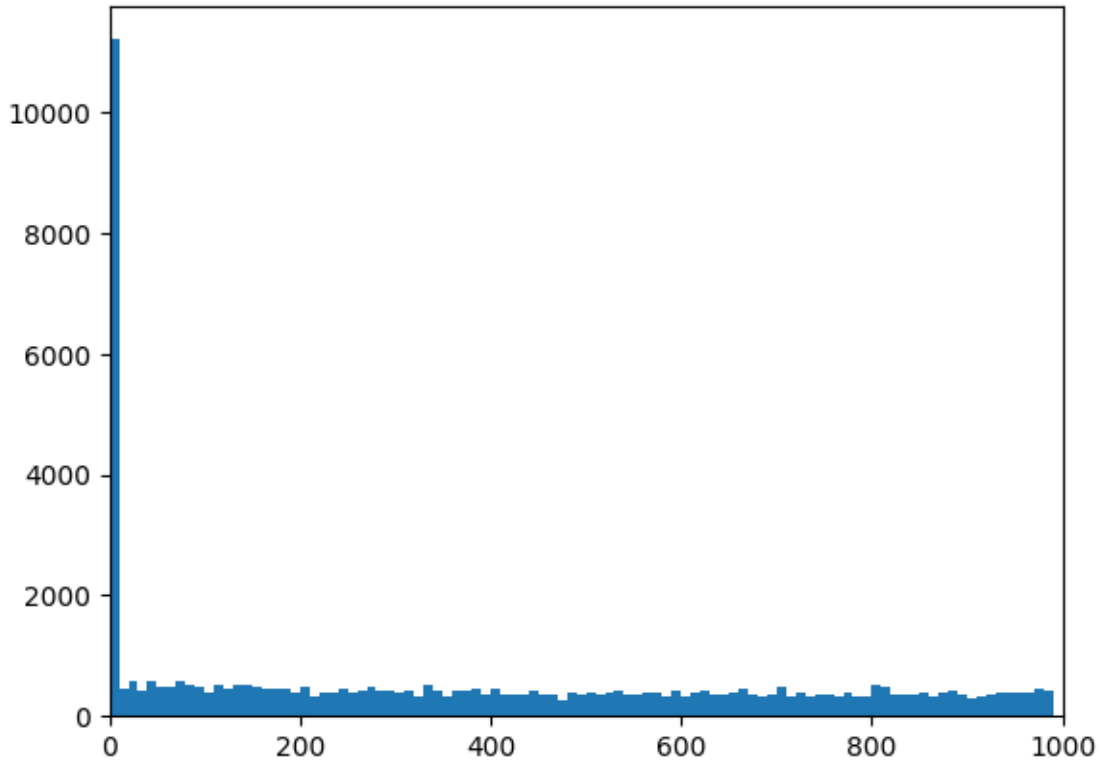


Here, we want to assign each peak (feature) to a gene (input table).

```
peaks_closest = bioframe.closest(genes, ctcf_peaks)
```

```
# Plot the distribution of distances from peaks to genes:
plt.hist(peaks_closest["distance"], np.arange(0, 1e3, 10))
plt.xlim([0, 1e3])
```

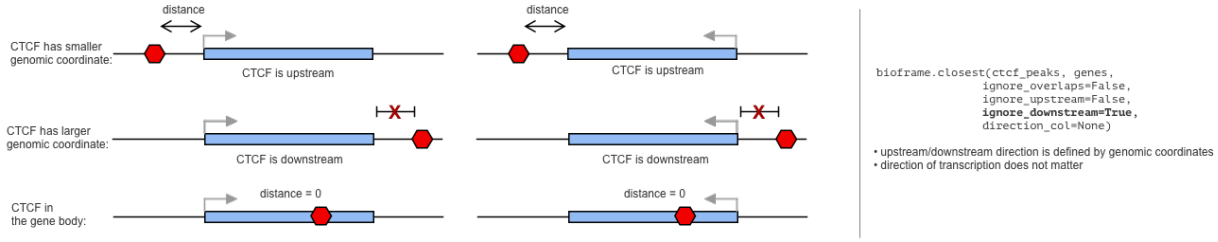
```
(0.0, 1000.0)
```



10.5 Ignore upstream/downstream peaks from genes (strand-indifferent version)

Sometimes you may want to ignore all the CTCFs upstream from the genes.

By default, `bioframe.overlap` does not know the orintation of the genes, and thus assumes that the upstream/downstream is defined by the genomic coordinate (upstream is the direction towards the smaller coordinate):




```
peaks_closest_upstream_nodir = bioframe.closest(
    genes,
    ctcf_peaks,
    ignore_overlaps=False,
    ignore_upstream=False,
    ignore_downstream=True,
    direction_col=None,
```

(continues on next page)

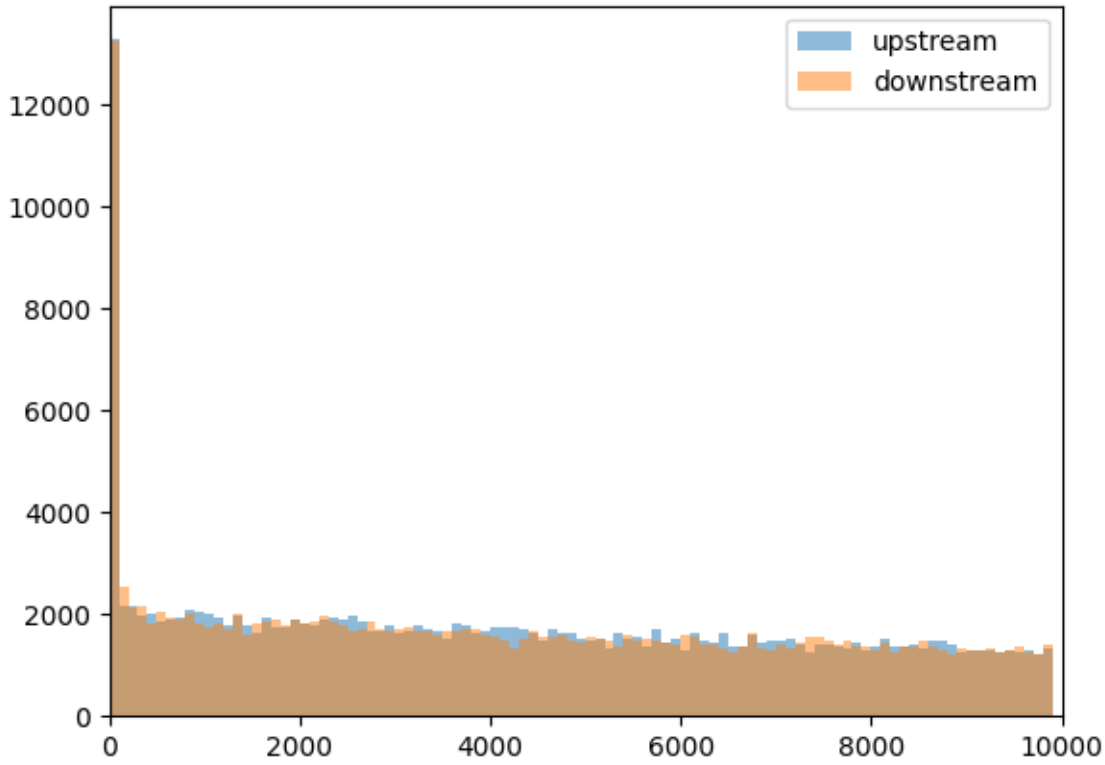
(continued from previous page)

```
)

peaks_closest_downstream_nodir = bioframe.closest(
    genes,
    ctcf_peaks,
    ignore_overlaps=False,
    ignore_upstream=True,
    ignore_downstream=False,
    direction_col=None,
)
```

Note that distribution did not change much, and upstream and downstream distances are very similar:

```
plt.hist(
    peaks_closest_upstream_nodir["distance"],
    np.arange(0, 1e4, 100),
    alpha=0.5,
    label="upstream",
)
plt.hist(
    peaks_closest_downstream_nodir["distance"],
    np.arange(0, 1e4, 100),
    alpha=0.5,
    label="downstream",
)
plt.xlim([0, 1e4])
plt.legend()
```

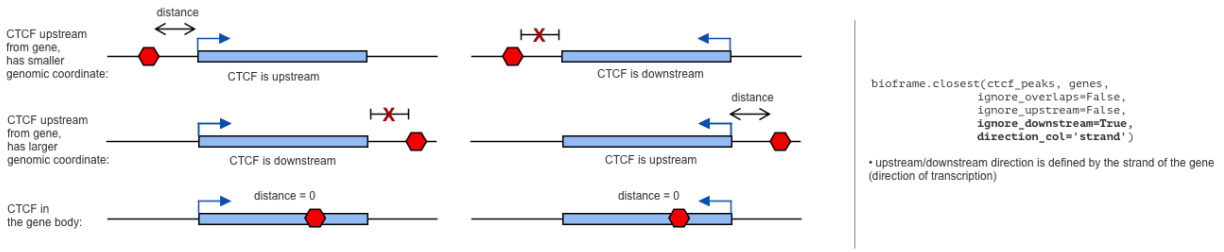
```
<matplotlib.legend.Legend at 0x7f33a44537f0>
```



10.6 Ignore upstream/downstream peaks from genes (strand-aware version)

More biologically relevant approach will be to **define upstream/downstream by strand of the gene**. CTCF upstream of transcription start site might play different role than CTCF after transcription end site.

`bioframe.closest` has the parameter `direction_col` to control for that:




```
# Note that "strand" here is the column name in genes table:
peaks_closest_upstream_dir = bioframe.closest(
    genes,
    ctcf_peaks,
    ignore_overlaps=False,
    ignore_upstream=False,
    ignore_downstream=True,
```

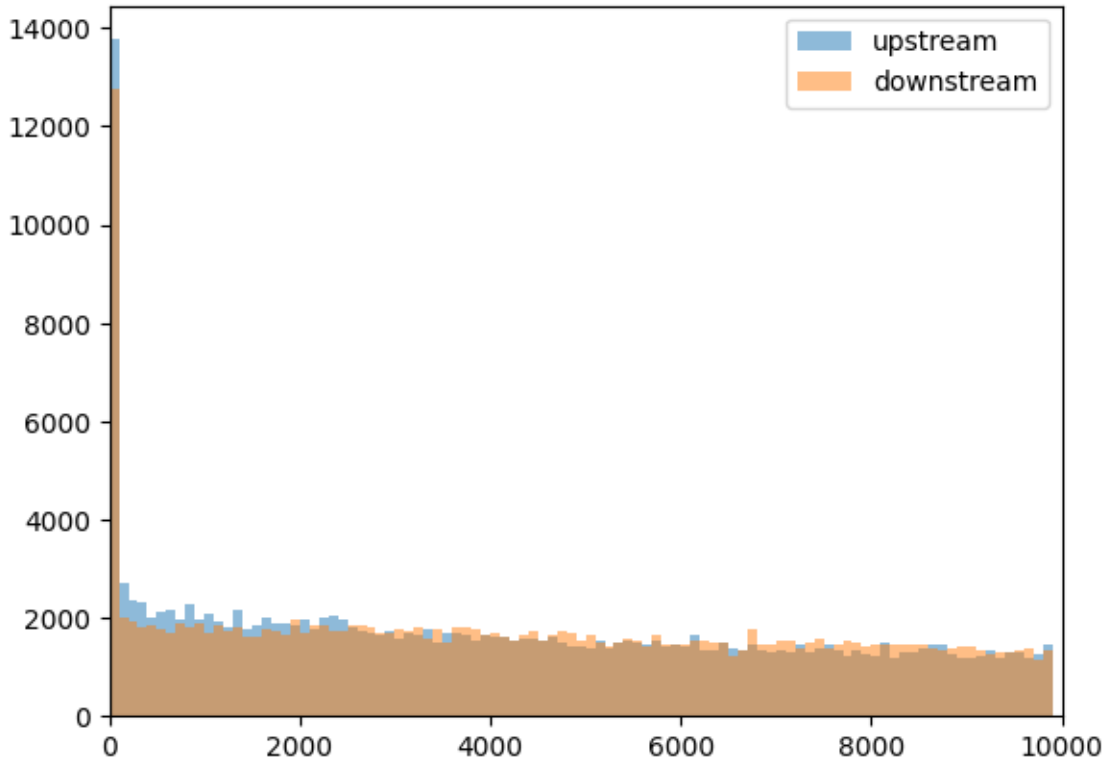
(continues on next page)

(continued from previous page)

```
    direction_col="strand",
)

peaks_closest_downstream_dir = bioframe.closest(
    genes,
    ctcf_peaks,
    ignore_overlaps=False,
    ignore_upstream=True,
    ignore_downstream=False,
    direction_col="strand",
)
```

```
plt.hist(
    peaks_closest_upstream_dir["distance"],
    np.arange(0, 1e4, 100),
    alpha=0.5,
    label="upstream",
)
plt.hist(
    peaks_closest_downstream_dir["distance"],
    np.arange(0, 1e4, 100),
    alpha=0.5,
    label="downstream",
)
plt.xlim([0, 1e4])
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f338e3e5660>
```



CTCF peaks upstream of the genes are more enriched at short distances to TSS, if we take the strand into account.

CHAPTER

ELEVEN

CONSTRUCTION

Functions:

from_any(regions[, fill_null, name_col, cols])	Attempts to make a genomic interval dataframe with columns [chr, start, end, name_col] from a variety of input types.
from_dict(regions[, cols])	Makes a dataframe from a dictionary of {str,int} pairs, interpreted as chromosome names.
make_viewframe(regions[, check_bounds, ...])	Makes and validates a dataframe *view_df* out of regions.
sanitize_bedframe(df1[, recast_dtypes, ...])	Attempts to clean a genomic interval dataframe to be a valid bedframe.

from_any(*regions*, *fill_null=False*, *name_col='name'*, *cols=None*)

Attempts to make a genomic interval dataframe with columns [chr, start, end, name_col] from a variety of input types.

Parameters

- **regions** (*supported input*) – Currently supported inputs:
  - dataframe
  - series of UCSC strings
  - dictionary of {str:int} key value pairs
  - pandas series where the index is interpreted as chromosomes and values are interpreted as end
  - list of tuples or lists, either [(chrom,start,end)] or [(chrom,start,end,name)]
  - tuple of tuples or lists, either [(chrom,start,end)] or [(chrom,start,end,name)]
- **fill_null** (*False or dictionary*) – Accepts a dictionary of {str:int} pairs, interpreted as chromosome sizes. Kept or backwards compatibility. Default False.
- **name_col** (*str*) – Column name. Only used if 4 column list is provided. Default "name".
- **cols** (*(str,str,str)*) – Names for dataframe columns. Default None sets them with get_default_colnames().

Returns
out_df

Return type
dataframe

from_dict(*regions*, *cols=None*)

Makes a dataframe from a dictionary of {str,int} pairs, interpreted as chromosome names.

Note that {str,(int,int)} dictionaries of tuples are no longer supported!

Parameters

- **regions** (*dict*)
- **name_col** (*str*) – Default 'name'.
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.

Returns
df

Return type
pandas.DataFrame

make_viewframe(*regions*, *check_bounds=None*, *name_style=None*, *view_name_col='name'*, *cols=None*)

Makes and validates a dataframe *view_df* out of regions.

Parameters

- **regions** (*supported input type*) – Currently supported input types:
  - a dictionary where keys are strings and values are integers {str:int}, specifying regions (chrom, 0, end, chrom)
  - a pandas series of chromosomes lengths with index specifying region names
  - a list of tuples [(chrom,start,end), ...] or [(chrom,start,end,name), ...]
  - a pandas DataFrame, skips to validation step
- **name_style** (*None or "ucsc"*) – If None and no column view_name_col, propagate values from cols[0] If "ucsc" and no column view_name_col, create UCSC style names
- **check_bounds** (*None, or chromosome sizes provided as any of valid formats above*) – Optional, if provided checks if regions in the view are contained by regions supplied in check_bounds, typically provided as a series of chromosome sizes. Default None.
- **view_name_col** (*str*) – Specifies column name of the view regions. Default 'name'.
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.

Returns
view_df

Return type
dataframe satisfying properties of a view

sanitize_bedframe(*df1*, *recast_dtypes=True*, *drop_null=False*, *start_exceed_end_action=None*, *cols=None*)

Attempts to clean a genomic interval dataframe to be a valid bedframe.

Parameters

- **df1** (*pandas.DataFrame*)

- **recast_dtypes** (*bool*) – Whether to attempt to recast column dtypes to pandas nullable dtypes.
- **drop_null** (*bool*) – Drops rows with pd.NA. Default False.
- **start_exceed_end_action** (*str or None*) – Options: 'flip' or 'drop' or None. Default None.
  - If 'flip', attempts to sanitize by flipping intervals with start>end.
  - If 'drop' attempts to sanitize dropping intervals with start>end.
  - If None, does not alter these intervals if present.
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.

Returns
out_df – Sanitized dataframe satisfying the properties of a bedframe.

Return type
pandas.DataFrame

Notes

The option `start_exceed_end_action='flip'` may be useful for gff files with strand information but starts > ends.

CHAPTER

TWELVE

VALIDATION

Functions:

is_bedframe(df[, raise_errors, cols])	Checks that required bedframe properties are satisfied for dataframe *df*.
is_cataloged(df, view_df[, raise_errors, ...])	Tests if all region names in *df[df_view_col]* are present in *view_df[view_name_col]*.
is_contained(df, view_df[, raise_errors, ...])	Tests if all genomic intervals in a bioframe *df* are cataloged and do not extend beyond their associated region in the view *view_df*.
is_covering(df, view_df[, view_name_col, ...])	Tests if a view *view_df* is covered by the set of genomic intervals in the bedframe *df*.
is_overlapping(df[, cols])	Tests if any genomic intervals in a bioframe *df* overlap.
is_sorted(df[, view_df, reset_index, ...])	Tests if a bedframe is changed by sorting.
is_tiling(df, view_df[, raise_errors, ...])	Tests if a view *view_df* is tiled by the set of genomic intervals in the bedframe *df*.
is_viewframe(region_df[, raise_errors, ...])	Checks that *region_df* is a valid viewFrame.

is_bedframe(*df*, *raise_errors=False*, *cols=None*)

Checks that required bedframe properties are satisfied for dataframe *df*.

This includes:

- chrom, start, end columns
- columns have valid dtypes
- **for each interval, if any of chrom, start, end are null, then all are** null
- all starts < ends.

Parameters

- **df** (*pandas.DataFrame*)
- **raise_errors** (*bool, optional [default: False]*) – If True, raises errors instead of returning a boolean False for invalid properties.
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.

Returns

is_bedframe

Return type
bool

Notes

Valid dtypes for chrom are object, string, or categorical. Valid dtypes for start and end are int/Int64Dtype.

is_cataloged(*df*, *view_df*, *raise_errors=False*, *df_view_col='view_region'*, *view_name_col='name'*)

Tests if all region names in *df[df_view_col]* are present in *view_df[view_name_col]*.

Parameters

- **df** (*pandas.DataFrame*)
- **view_df** (*pandas.DataFrame*)
- **raise_errors** (*bool*) – If True, raises errors instead of returning a boolean False for invalid properties. Default False.
- **df_view_col** (*str*) – Name of column from df that indicates region in view.
- **view_name_col** (*str*) – Name of column from view that specifies region name.

Returns
is_cataloged

Return type
bool

Notes

Does not check if names in *view_df[view_name_col]* are unique.

is_contained(*df*, *view_df*, *raise_errors=False*, *df_view_col=None*, *view_name_col='name'*, *cols=None*, *cols_view=None*)

Tests if all genomic intervals in a bioframe *df* are cataloged and do not extend beyond their associated region in the view *view_df*.

Parameters

- **df** (*pandas.DataFrame*)
- **view_df** (*pandas.DataFrame*) – Valid viewframe.
- **raise_errors** (*bool*) – If True, raises errors instead of returning a boolean False for invalid properties. Default False.
- **df_view_col** – Column from df used to associate interviews with view regions. Default *view_region*.
- **view_name_col** – Column from view_df with view region names. Default *name*.
- **cols** (*(str, str, str)*) – Column names for chrom, start, end in df.
- **cols_view** (*(str, str, str)*) – Column names for chrom, start, end in view_df.

Returns
is_contained

Return type
bool

is_covering(*df*, *view_df*, *view_name_col='name'*, *cols=None*, *cols_view=None*)

Tests if a view *view_df* is covered by the set of genomic intervals in the bedframe *df*.

This test is true if `complement(df,view_df)` is empty. Also note this test ignores regions assigned to intervals in *df* since regions are re-assigned in *bioframe.ops.complement()*.

Parameters

- **df** (*pandas.DataFrame*)
- **view_df** (*pandas.DataFrame*) – Valid viewFrame.
- **view_name_col** – Column from view_df with view region names. Default *name*.
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.
- **cols_view** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals in view_df, provided separately for each set. The default values are 'chrom', 'start', 'end'.

Returns
is_covering

Return type
bool

is_overlapping(*df*, *cols=None*)

Tests if any genomic intervals in a bioframe *df* overlap.

Also see *bioframe.ops.merge()*.

Parameters

- **df** (*pandas.DataFrame*)
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.

Returns
is_overlapping

Return type
bool

is_sorted(*df*, *view_df=None*, *reset_index=True*, *df_view_col=None*, *view_name_col='name'*, *cols=None*, *cols_view=None*)

Tests if a bedframe is changed by sorting.

Also see *bioframe.ops.sort_bedframe()*.

Parameters

- **df** (*pandas.DataFrame*)
- **view_df** (*pandas.DataFrame | dict-like*) – Optional view to pass to `sort_bedframe`. When it is dict-like :func:'bioframe.make_viewframe' will be used to convert to viewframe. If view_df is not provided df is assumed to be sorted by chrom and start.
- **reset_index** (*bool*) – Optional argument to pass to `sort_bedframe`.

- **df_view_col** (*None | str*) – Name of column from df that indicates region in view. If None, :func:'bioframe.assign_view' will be used to assign view regions. Default None.
- **view_name_col** (*str*) – Name of column from view that specifies unique region name.
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.
- **cols_view** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals in view_df, provided separately for each set. The default values are 'chrom', 'start', 'end'.

Returns
is_sorted

Return type
bool

is_tiling(*df*, *view_df*, *raise_errors=False*, *df_view_col='view_region'*, *view_name_col='name'*, *cols=None*, *cols_view=None*)

Tests if a view *view_df* is tiled by the set of genomic intervals in the bedframe *df*.

This is true if:

- df is not overlapping
- df is covering view_df
- df is contained in view_df

Parameters

- **df** (*pandas.DataFrame*)
- **view_df** (*pandas.DataFrame*) – valid viewFrame
- **raise_errors** (*bool*) – If True, raises errors instead of returning a boolean False for invalid properties. Default False.
- **df_view_col** (*str*) – Name of column from df that indicates region in view.
- **view_name_col** (*str*) – Name of column from view that specifies unique region name.
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.
- **cols_view** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals in view_df, provided separately for each set. The default values are 'chrom', 'start', 'end'.

Returns
is_tiling

Return type
bool

is_viewframe(*region_df*, *raise_errors=False*, *view_name_col='name'*, *cols=None*)

Checks that *region_df* is a valid viewFrame.

This includes:

- it satisfies requirements for a bedframe, including columns for ('chrom', 'start', 'end')

- it has an additional column, view_name_col, with default 'name'
- it does not contain null values
- entries in the view_name_col are unique.
- intervals are non-overlapping

Parameters

- **region_df** (*pandas.DataFrame*) – Dataframe of genomic intervals to be tested.
- **raise_errors** (*bool*) – If True, raises errors instead of returning a boolean False for invalid properties. Default False.
- **view_name_col** (*str*) – Specifies column name of the view regions. Default 'name'.
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.

Returns
is_viewframe

Return type
bool

CHAPTER

THIRTEEN

INTERVAL OPERATIONS

Functions:

assign_view(df, view_df[, drop_unassigned, ...])	Associates genomic intervals in bedframe `df` with regions in viewframe `view_df`, based on their largest overlap.
closest(df1[, df2, k, ignore_overlaps, ...])	For every interval in dataframe *df1* find k closest genomic intervals in dataframe *df2*.
cluster(df[, min_dist, cols, on, ...])	Cluster overlapping intervals into groups.
complement(df[, view_df, view_name_col, ...])	Find genomic regions in a viewFrame 'view_df' that are not covered by any interval in the dataFrame 'df'.
count_overlaps(df1, df2[, suffixes, ...])	Count number of overlapping genomic intervals.
coverage(df1, df2[, suffixes, return_input, ...])	Quantify the coverage of intervals from 'df1' by intervals from 'df2'.
expand(df[, pad, scale, side, cols])	Expand each interval by an amount specified with *pad*.
merge(df[, min_dist, cols, on])	Merge overlapping intervals.
overlap(df1, df2[, how, return_input, ...])	Find pairs of overlapping genomic intervals.
select(df, region[, cols])	Return all genomic intervals in a dataframe that overlap a genomic region.
select_indices(df, region[, cols])	Return integer indices of all genomic intervals that overlap a query range.
select_labels(df, region[, cols])	Return pandas Index labels of all genomic intervals that overlap a query range.
select_mask(df, region[, cols])	Return boolean mask for all genomic intervals that overlap a query range.
setdiff(df1, df2[, cols1, cols2, on])	Generate a new dataframe of genomic intervals by removing any interval from the first dataframe that overlaps an interval from the second dataframe.
sort_bedframe(df[, view_df, reset_index, ...])	Sorts a bedframe 'df'.
subtract(df1, df2[, return_index, suffixes, ...])	Generate a new set of genomic intervals by subtracting the second set of genomic intervals from the first.
trim(df[, view_df, df_view_col, ...])	Trim each interval to fall within regions specified in the viewframe 'view_df'.

assign_view(*df*, *view_df*, *drop_unassigned=False*, *df_view_col='view_region'*, *view_name_col='name'*, *cols=None*, *cols_view=None*)

Associates genomic intervals in bedframe `df` with regions in viewframe `view_df`, based on their largest overlap.

Parameters

- **df** (*pandas.DataFrame*)

- **view_df** (*pandas.DataFrame*) – ViewFrame specifying region start and ends for assignment. Attempts to convert dictionary and pd.Series formats to viewFrames.
- **drop_unassigned** (*bool*) – If True, drop intervals in df that do not overlap a region in the view. Default False.
- **df_view_col** (*str*) – The column of `df` used to specify view regions. The associated region in view_df is then used for trimming. If no view_df is provided, uses the chrom column, `df[cols[0]]`. Default "view_region".
- **view_name_col** (*str*) – Column of `view_df` with region names. Default 'name'.
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals. The default values are 'chrom', 'start', 'end'.
- **cols_view** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals in the view. The default values are 'chrom', 'start', 'end'.

Returns

- **out_df** (*dataframe with an associated view region for each interval in*)
- `out_df[view_name_col]`.

Notes

Resets index.

closest(*df1*, *df2=None*, *k=1*, *ignore_overlaps=False*, *ignore_upstream=False*, *ignore_downstream=False*, *direction_col=None*, *tie_breaking_col=None*, *return_input=True*, *return_index=False*, *return_distance=True*, *return_overlap=False*, *suffixes=('', '_')*, *cols1=None*, *cols2=None*)

For every interval in dataframe *df1* find k closest genomic intervals in dataframe *df2*.

Currently, we are not taking the feature strands into account for filtering. However, the strand can be used for definition of upstream/downstream of the feature (direction).

Note that, unless specified otherwise, overlapping intervals are considered as closest. When multiple intervals are located at the same distance, the ones with the lowest index in *df2* are returned.

Parameters

- **df1** (*pandas.DataFrame*) – Two sets of genomic intervals stored as a DataFrame. If *df2* is None, find closest non-identical intervals within the same set.
- **df2** (*pandas.DataFrame*) – Two sets of genomic intervals stored as a DataFrame. If *df2* is None, find closest non-identical intervals within the same set.
- **k** (*int*) – The number of the closest intervals to report.
- **ignore_overlaps** (*bool*) – If True, ignore overlapping intervals and return the closest non-overlapping interval.
- **ignore_upstream** (*bool*) – If True, ignore intervals in *df2* that are upstream of intervals in *df1*, relative to the reference strand or the strand specified by direction_col.
- **ignore_downstream** (*bool*) – If True, ignore intervals in *df2* that are downstream of intervals in *df1*, relative to the reference strand or the strand specified by direction_col.
- **direction_col** (*str*) – Name of direction column that will set upstream/downstream orientation for each feature. The column should contain bioframe-compliant strand values ("+", "-", ".").

- **tie_breaking_col** (*str*) – A column in *df2* to use for breaking ties when multiple intervals are located at the same distance. Intervals with *lower* values will be selected.
- **return_input** (*bool*) – If True, return input
- **return_index** (*bool*) – If True, return indices
- **return_distance** (*bool*) – If True, return distances. Returns zero for overlaps.
- **return_overlap** (*bool*) – If True, return columns: 'have_overlap', 'overlap_start', and 'overlap_end'. Fills df_closest['overlap_start'] and df['overlap_end'] with None if non-overlapping. Default False.
- **suffixes** (*(str, str)*) – The suffixes for the columns of the two sets.
- **cols1** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.
- **cols2** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.

Returns
df_closest – If no intervals found, returns none.

Return type
pandas.DataFrame

Notes

By default, direction is defined by the reference genome: everything with smaller coordinate is considered upstream, everything with larger coordinate is considered downstream.

If `direction_col` is provided, upstream/downstream are relative to the direction column in `df1`, i.e. features marked "+" and "." strand will define upstream and downstream as above, while features marked "-" have upstream and downstream reversed: smaller coordinates are downstream and larger coordinates are upstream.

cluster(*df*, *min_dist=0*, *cols=None*, *on=None*, *return_input=True*, *return_cluster_ids=True*, *return_cluster_intervals=True*)

Cluster overlapping intervals into groups.

Can return numeric ids for these groups (with *return_cluster_ids `=True) and/or their genomic coordinates (with `return_cluster_intervals `=True). Also see :func: `merge()*, which discards original intervals and returns a new set.

Parameters

- **df** (*pandas.DataFrame*)
- **min_dist** (*float or None*) – If provided, cluster intervals separated by this distance or less. If `None`, do not cluster non-overlapping intervals. Since bioframe uses semi-open intervals, interval pairs [0,1) and [1,2) do not overlap, but are separated by a distance of 0. Such adjacent intervals are not clustered when `min_dist=None`, but are clustered when `min_dist=0`.
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals. The default values are 'chrom', 'start', 'end'.

- **on** (*None or list*) – List of column names to perform clustering on independently, passed as an argument to df.groupby before clustering. Default is `None`. An example useage would be to pass `on=['strand']`.
- **return_input** (*bool*) – If True, return input
- **return_cluster_ids** (*bool*) – If True, return ids for clusters
- **return_cluster_invervals** (*bool*) – If True, return clustered interval the original interval belongs to

Returns
df_clustered

Return type
pd.DataFrame

complement(*df*, *view_df=None*, *view_name_col='name'*, *cols=None*, *cols_view=None*)

Find genomic regions in a viewFrame 'view_df' that are not covered by any interval in the dataFrame 'df'.

First assigns intervals in 'df' to region in 'view_df', splitting intervals in 'df' as necessary.

Parameters

- **df** (*pandas.DataFrame*)
- **view_df** (*pandas.Dataframe*) – If none, attempts to infer the view from chroms (i.e. df[cols[0]]).
- **view_name_col** (*str*) – Name of column in view_df with unique reigon names. Default 'name'.
- **cols** (*(str, str, str)*) – The names of columns containing the chromosome, start and end of the genomic intervals. The default values are 'chrom', 'start', 'end'.
- **cols_view** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals in the view. The default values are 'chrom', 'start', 'end'.

Returns
df_complement

Return type
pandas.DataFrame

Notes

Discards null intervals in input, and df_complement has regular int dtype.

count_overlaps(*df1*, *df2*, *suffixes=('', '_')*, *return_input=True*, *cols1=None*, *cols2=None*, *on=None*)

Count number of overlapping genomic intervals.

Parameters

- **df1** (*pandas.DataFrame*) – Two sets of genomic intervals stored as a DataFrame.
- **df2** (*pandas.DataFrame*) – Two sets of genomic intervals stored as a DataFrame.
- **suffixes** (*(str, str)*) – The suffixes for the columns of the two overlapped sets.
- **return_input** (*bool*) – If True, return columns from input dfs. Default True.

- **cols1** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.
- **cols2** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.
- **on** (*list*) – List of additional shared columns to consider as separate groups when considering overlaps. A common use would be passing on=['strand']. Default is None.

Returns
df_counts

Return type
pandas.DataFrame

Notes

Resets index.

coverage(*df1*, *df2*, *suffixes=('', '_')*, *return_input=True*, *cols1=None*, *cols2=None*)

Quantify the coverage of intervals from 'df1' by intervals from 'df2'.

For every interval in 'df1' find the number of base pairs covered by intervals in 'df2'. Note this only quantifies whether a basepair in 'df1' was covered, as 'df2' is merged before calculating coverage.

Parameters

- **df1** (*pandas.DataFrame*) – Two sets of genomic intervals stored as a DataFrame.
- **df2** (*pandas.DataFrame*) – Two sets of genomic intervals stored as a DataFrame.
- **suffixes** (*(str, str)*) – The suffixes for the columns of the two overlapped sets.
- **return_input** (*bool*) – If True, return input as well as computed coverage
- **cols1** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.
- **cols2** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.

Returns
df_coverage

Return type
pandas.DataFrame

Notes

Resets index.

expand(*df*, *pad=None*, *scale=None*, *side='both'*, *cols=None*)

Expand each interval by an amount specified with *pad*.

Negative values for pad shrink the interval, up to the midpoint. Multiplicative rescaling of intervals enabled with scale. Only one of pad or scale can be provided. Often followed by *trim()*.

Parameters

- **df** (*pandas.DataFrame*)
- **pad** (*int, optional*) – The amount by which the intervals are additively expanded *on each side*. Negative values for pad shrink intervals, but not beyond the interval midpoint. Either *pad* or *scale* must be supplied.
- **scale** (*float, optional*) – The factor by which to scale intervals multiplicatively on each side, e.g `scale=2` doubles each interval, `scale=0` returns midpoints, and `scale=1` returns original intervals. Default False. Either *pad* or *scale* must be supplied.
- **side** (*str, optional*) – Which side to expand, possible values are 'left', 'right' and 'both'. Default 'both'.
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals. Default values are 'chrom', 'start', 'end'.

Returns
df_expanded

Return type
pandas.DataFrame

Notes

See `bioframe.trim()` for trimming interals after expansion.

merge(*df*, *min_dist=0*, *cols=None*, *on=None*)

Merge overlapping intervals.

This returns a new dataframe of genomic intervals, which have the genomic coordinates of the interval cluster groups from the input dataframe. Also *cluster()*, which returns the assignment of intervals to clusters prior to merging.

Parameters

- **df** (*pandas.DataFrame*)
- **min_dist** (*float or None*) – If provided, merge intervals separated by this distance or less. If None, do not merge non-overlapping intervals. Using `min_dist=0` and `min_dist=None` will bring different results. bioframe uses semi-open intervals, so interval pairs [0,1) and [1,2) do not overlap, but are separated by a distance of 0. Adjacent intervals are not merged when `min_dist=None`, but are merged when `min_dist=0`.
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals. The default values are 'chrom', 'start', 'end'.
- **on** (*None or list*) – List of column names to perform clustering on independently, passed as an argument to df.groupby before clustering. Default is None. An example useage would be to pass `on=['strand']`.

Returns
df_merged – A pandas dataframe with coordinates of merged clusters.

Return type
pandas.DataFrame

Notes

Resets index.

overlap(*df1*, *df2*, *how='left'*, *return_input=True*, *return_index=False*, *return_overlap=False*, *suffixes=('', '_')*, *keep_order=None*, *cols1=None*, *cols2=None*, *on=None*, *ensure_int=True*)

Find pairs of overlapping genomic intervals.

Parameters

- **df1** (*pandas.DataFrame*) – Two sets of genomic intervals stored as a DataFrame.
- **df2** (*pandas.DataFrame*) – Two sets of genomic intervals stored as a DataFrame.
- **how** (*{'left', 'right', 'outer', 'inner'}, default 'left'*) – How to handle the overlaps on the two dataframes. left: use the set of intervals in df1 right: use the set of intervals in df2 outer: use the union of the set of intervals from df1 and df2 inner: use intersection of the set of intervals from df1 and df2
- **return_input** (*bool, optional*) – If True, return columns from input dfs. Default True.
- **return_index** (*bool, optional*) – If True, return indicies of overlapping pairs as two new columns ('index'+suffixes[0] and 'index'+suffixes[1]). Default False.
- **return_overlap** (*bool, optional*) – If True, return overlapping intervals for the overlapping pairs as two additional columns (*overlap_start*, *overlap_end*). When *cols1* is modified, *start* and *end* are replaced accordingly. When *return_overlap* is a string, its value is used for naming the overlap columns: *return_overlap* + *"_start"*, *return_overlap* + *"_end"*. Default False.
- **suffixes** (*(str, str), optional*) – The suffixes for the columns of the two overlapped sets.
- **keep_order** (*bool, optional*) – If True and how='left', sort the output dataframe to preserve the order of the intervals in df1. Cannot be used with how='right'/'outer'/'inner'. Default True for how='left', and None otherwise. Note that it relies on sorting of index in the original dataframes, and will reorder the output by index.
- **cols1** (*(str, str, str) or None, optional*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.
- **cols2** (*(str, str, str) or None, optional*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.
- **on** (*list or None, optional*) – List of additional shared columns to consider as separate groups when considering overlaps. A common use would be passing on=['strand']. Default is None.
- **ensure_int** (*bool, optional [default: True]*) – If True, ensures that the output dataframe uses integer dtypes for start and end coordinates. This may involve converting coordinate columns to nullable types in outer joins. Default True.

Returns
df_overlap

Return type
pandas.DataFrame

Notes

If `ensure_int` is False, inner joins will preserve coordinate dtypes from the input dataframes, but outer joins will be subject to native type casting rules if missing data is introduced. For example, if *df1* uses a NumPy integer dtype for *start* and/or *end*, the output dataframe will use the same dtype after an inner join, but, due to casting rules, may produce `float64` after a left/right/outer join with missing data stored as `NaN`. On the other hand, if *df1* uses Pandas nullable dtypes, the corresponding coordinate columns will preserve the same dtype in the output, with missing data stored as `NA`.

select(*df*, *region*, *cols=None*)

Return all genomic intervals in a dataframe that overlap a genomic region.

Parameters

- **df** (*pandas.DataFrame*)
- **region** (*str or tuple*) – The genomic region to select from the dataframe in UCSC-style genomic region string, or triple (chrom, start, end).
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals. The default values are 'chrom', 'start', 'end'.

Returns
df

Return type
pandas.DataFrame

Notes

See *core.stringops.parse_region()* for more information on region formatting.

See also:

select_mask(), *select_indices()*, *select_labels()*

select_indices(*df*, *region*, *cols=None*)

Return integer indices of all genomic intervals that overlap a query range.

Parameters

- **df** (*pandas.DataFrame*)
- **region** (*str or tuple*) – The genomic region to select from the dataframe in UCSC-style genomic region string, or triple (chrom, start, end).
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals. The default values are 'chrom', 'start', 'end'.

Return type
1D array of int

select_labels(*df*, *region*, *cols=None*)

Return pandas Index labels of all genomic intervals that overlap a query range.

Parameters

- **df** (*pandas.DataFrame*)
- **region** (*str or tuple*) – The genomic region to select from the dataframe in UCSC-style genomic region string, or triple (chrom, start, end).
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals. The default values are 'chrom', 'start', 'end'.

Return type
pandas.Index

select_mask(*df*, *region*, *cols=None*)

Return boolean mask for all genomic intervals that overlap a query range.

Parameters

- **df** (*pandas.DataFrame*)
- **region** (*str or tuple*) – The genomic region to select from the dataframe in UCSC-style genomic region string, or triple (chrom, start, end).
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals. The default values are 'chrom', 'start', 'end'.

Return type
Boolean array of shape (len(df),)

setdiff(*df1*, *df2*, *cols1=None*, *cols2=None*, *on=None*)

Generate a new dataframe of genomic intervals by removing any interval from the first dataframe that overlaps an interval from the second dataframe.

Parameters

- **df1** (*pandas.DataFrame*) – Two sets of genomic intervals stored as DataFrames.
- **df2** (*pandas.DataFrame*) – Two sets of genomic intervals stored as DataFrames.
- **cols1** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each dataframe. The default values are 'chrom', 'start', 'end'.
- **cols2** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each dataframe. The default values are 'chrom', 'start', 'end'.
- **on** (*None or list*) – Additional column names to perform clustering on independently, passed as an argument to df.groupby when considering overlaps and must be present in both dataframes. Examples for additional columns include 'strand'.

Returns
df_setdiff

Return type
pandas.DataFrame

sort_bedframe(*df*, *view_df=None*, *reset_index=True*, *df_view_col=None*, *view_name_col='name'*, *cols=None*, *cols_view=None*)

Sorts a bedframe 'df'.

If 'view_df' is not provided, sorts by `cols` (e.g. "chrom", "start", "end"). If 'view_df' is provided and 'df_view_col' is not provided, uses *bioframe.ops.assign_view()* with `df_view_col='view_region'` to assign intervals to the view regions with the largest overlap and then sorts. If 'view_df' and 'df_view_col' are both provided, checks if the latter are cataloged in 'view_name_col', and then sorts.

df
[pandas.DataFrame] Valid bedframe.

view_df
[pandas.DataFrame | dict-like] Valid input to make a viewframe. When it is dict-like :func:'bioframe.make_viewframe' will be used to convert to viewframe. If view_df is not provided df is sorted by chrom and start.

reset_index
[bool] Default True.

df_view_col: None | str
Column from 'df' used to associate intervals with view regions. The associated region in 'view_df' is then used for sorting. If None, :func:'bioframe.assign_view' will be used to assign view regions. Default None.

view_name_col: str
Column from view_df with names of regions. Default *name*.

cols
[(str, str, str) or None] The names of columns containing the chromosome, start and end of the genomic intervals. The default values are 'chrom', 'start', 'end'.

cols_view
[(str, str, str) or None] The names of columns containing the chromosome, start and end of the genomic intervals in the view. The default values are 'chrom', 'start', 'end'.

Returns
out_df

Return type
sorted bedframe

Notes

df_view_col is currently returned as an ordered categorical

`subtract`(*df1*, *df2*, *return_index=False*, *suffixes=('', '_')*, *cols1=None*, *cols2=None*)

Generate a new set of genomic intervals by subtracting the second set of genomic intervals from the first.

Parameters

- **df1** (*pandas.DataFrame*) – Two sets of genomic intervals stored as a DataFrame.
- **df2** (*pandas.DataFrame*) – Two sets of genomic intervals stored as a DataFrame.
- **return_index** (*bool*) – Whether to return the indices of the original intervals ('index'+suffixes[0]), and the indices of any sub-intervals split by subtraction ('sub_index'+suffixes[1]). Default False.
- **suffixes** (*(str,str)*) – Suffixes for returned indices. Only alters output if return_index is True. Default ("","_").

- **cols1** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.
- **cols2** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.

Returns
df_subtracted

Return type
pandas.DataFrame

Notes

Resets index, drops completely subtracted (null) intervals, and casts to pd.Int64Dtype().

trim(*df*, *view_df=None*, *df_view_col=None*, *view_name_col='name'*, *return_view_columns=False*, *cols=None*, *cols_view=None*)

Trim each interval to fall within regions specified in the viewframe 'view_df'.

Intervals that fall outside of view regions are replaced with nulls. If no 'view_df' is provided, intervals are truncated at zero to avoid

negative values.

Parameters

- **df** (*pandas.DataFrame*)
- **view_df** (*None or pandas.DataFrame*) – View specifying region start and ends for trimming. Attempts to convert dictionary and pd.Series formats to viewFrames.
- **df_view_col** (*str or None*) – The column of 'df' used to specify view regions. The associated region in 'view_df' is then used for trimming. If None, :func:'bioframe.ops.assign_view' will be used to assign view regions. If no 'view_df' is provided, uses the 'chrom' column, df[cols[0]]. Default None.
- **view_name_col** (*str*) – Column of df with region names. Default 'name'.
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals. The default values are 'chrom', 'start', 'end'.
- **cols_view** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals in the view. The default values are 'chrom', 'start', 'end'.

Returns
df_trimmed

Return type
pandas.DataFrame

CHAPTER

FOURTEEN

FILE I/O

Functions:

load_fasta(filepath_or[, engine])	Load lazy fasta sequences from an indexed fasta file (optionally compressed) or from a collection of uncompressed fasta files.
read_bam(fp[, chrom, start, end])	Read bam records into a DataFrame.
read_bigbed(path, chrom[, start, end, engine])	Read intervals from a bigBed file.
read_bigwig(path, chrom[, start, end, engine])	Read intervals from a bigWig file.
read_chromsizes(filepath_or[, ...])	Read a `<db>.chrom.sizes` or `<db>.chromInfo.txt` file from the UCSC database, where `db` is a genome assembly name, as a *pandas.Series*.
read_pairix(fp, region1[, region2, ...])	Read a pairix-indexed file into DataFrame.
read_tabix(fp[, chrom, start, end])	Read a tabix-indexed file into dataFrame.
read_table(filepath_or[, schema, ...])	Read a tab-delimited file into a data frame.
to_bigbed(df, chromsizes, outpath[, schema, ...])	Save a bedGraph-like dataframe as a binary BigWig track.
to_bigwig(df, chromsizes, outpath[, ...])	Save a bedGraph-like dataframe as a binary BigWig track.

load_fasta(*filepath_or*, *engine='pysam'*, ***kwargs*)

Load lazy fasta sequences from an indexed fasta file (optionally compressed) or from a collection of uncompressed fasta files.

Parameters

- **filepath_or** (*str or iterable*) – If a string, a filepath to a single *.fa* or *.fa.gz* file. Assumed to be accompanied by a *.fai* index file. Depending on the engine, the index may be created on the fly, and some compression formats may not be supported. If not a string, an iterable of fasta file paths each assumed to contain a single sequence.
- **engine** (*{'pysam', 'pyfaidx'}, optional*) – Module to use for loading sequences.
- **kwargs** (*optional*) – Options to pass to `pysam.FastaFile` or `pyfaidx.Fasta`.

Return type

OrderedDict of (lazy) fasta records.

Notes

- pysam/samtools can read .fai and .gzi indexed files, I think.
- pyfaidx can handle uncompressed and bgzf compressed files.

read_bam(*fp*, *chrom=None*, *start=None*, *end=None*)

Read bam records into a DataFrame.

read_bigbed(*path*, *chrom*, *start=None*, *end=None*, *engine='auto'*)

Read intervals from a bigBed file.

Parameters

- **path** (*str*) – Path or URL to a bigBed file
- **chrom** (*str*)
- **start** (*int, optional*) – Start and end coordinates. Defaults to 0 and chromosome length.
- **end** (*int, optional*) – Start and end coordinates. Defaults to 0 and chromosome length.
- **engine** (*{"auto", "pybbi", "pybigwig"}*) – Library to use for querying the bigBed file.

Return type
DataFrame

read_bigwig(*path*, *chrom*, *start=None*, *end=None*, *engine='auto'*)

Read intervals from a bigWig file.

Parameters

- **path** (*str*) – Path or URL to a bigWig file
- **chrom** (*str*)
- **start** (*int, optional*) – Start and end coordinates. Defaults to 0 and chromosome length.
- **end** (*int, optional*) – Start and end coordinates. Defaults to 0 and chromosome length.
- **engine** (*{"auto", "pybbi", "pybigwig"}*) – Library to use for querying the bigWig file.

Return type
DataFrame

read_chromsizes(*filepath_or*, *filter_chroms=True*, *chrom_patterns=('^chr[0-9]+$', '^chr[XY]$', '^chrM$')*, *natsort=True*, *as_bed=False*, ***kwargs*)

Read a <db>.chrom.sizes or <db>.chromInfo.txt file from the UCSC database, where db is a genome assembly name, as a *pandas.Series*.

Parameters

- **filepath_or** (*str or file-like*) – Path or url to text file, or buffer.
- **filter_chroms** (*bool, optional*) – Filter for chromosome names given in chrom_patterns.
- **chrom_patterns** (*sequence, optional*) – Sequence of regular expressions to capture desired sequence names.
- **natsort** (*bool, optional*) – Sort each captured group of names in natural order. Default is True.

- **as_bed** (*bool, optional*) – If True, return chromsizes as an interval dataframe (chrom, start, end).
- ****kwargs** – Passed to `pandas.read_csv()`

Return type
Series of integer bp lengths indexed by sequence name or an interval dataframe.

Notes

Mention name patterns

See also:

- UCSC assembly terminology: <http://genome.ucsc.edu/FAQ/FAQdownloads.html#download9>
- NCBI assembly terminology: <https://www.ncbi.nlm.nih.gov/grc/help/definitions>

read_pairix(*fp*, *region1*, *region2=None*, *chromsizes=None*, *columns=None*, *usecols=None*, *dtypes=None*, ***kwargs*)

Read a pairix-indexed file into DataFrame.

read_tabix(*fp*, *chrom=None*, *start=None*, *end=None*)

Read a tabix-indexed file into dataFrame.

read_table(*filepath_or*, *schema=None*, *schema_is_strict=False*, ***kwargs*)

Read a tab-delimited file into a data frame.

Equivalent to `pandas.read_table()` but supports an additional *schema* argument to populate column names for common genomic formats.

Parameters

- **filepath_or** (*str, path object or file-like object*) – Any valid string path is acceptable. The string could be a URL
- **schema** (*str*) – Schema to use for table column names.
- **schema_is_strict** (*bool*) – Whether to check if columns are filled with NAs.

Returns
df

Return type
pandas.DataFrame of intervals

to_bigbed(*df*, *chromsizes*, *outpath*, *schema='bed6'*, *path_to_binary=None*)

Save a bedGraph-like dataframe as a binary BigWig track.

Parameters

- **df** (*pandas.DataFrame*) – Data frame with columns 'chrom', 'start', 'end' and one or more value columns
- **chromsizes** (*pandas.Series*) – Series indexed by chromosome name mapping to their lengths in bp
- **outpath** (*str*) – The output BigWig file path
- **value_field** (*str, optional*) – Select the column label of the data frame to generate the track. Default is to use the fourth column.

- **path_to_binary** (*str, optional*) – Provide system path to the bedGraphToBigWig binary.

to_bigwig(*df*, *chromsizes*, *outpath*, *value_field=None*, *path_to_binary=None*)

Save a bedGraph-like dataframe as a binary BigWig track.

Parameters

- **df** (*pandas.DataFrame*) – Data frame with columns 'chrom', 'start', 'end' and one or more value columns
- **chromsizes** (*pandas.Series*) – Series indexed by chromosome name mapping to their lengths in bp
- **outpath** (*str*) – The output BigWig file path
- **value_field** (*str, optional*) – Select the column label of the data frame to generate the track. Default is to use the fourth column.
- **path_to_binary** (*str, optional*) – Provide system path to the bedGraphToBigWig binary.

to_bed(*df: DataFrame*, *path: str | Path | None = None*, ***, *schema: str = 'infer'*, *validate_fields: bool = True*, *require_sorted: bool = False*, *chromsizes: dict | Series | None = None*, *strict_score: bool = False*, *replace_na: bool = True*, *na_rep: str = 'nan'*) → str | None

Write a DataFrame to a BED file.

Parameters

- **df** (*pd.DataFrame*) – DataFrame to write.
- **path** (*str or Path, optional*) – Path to write the BED file to. If `None`, the serialized BED file is returned as a string.
- **schema** (*str, optional [default: "infer"]*) – BED schema to use. If `"infer"`, the schema is inferred from the DataFrame's columns.
- **validate_fields** (*bool, optional [default: True]*) – Whether to validate the fields of the BED file.
- **require_sorted** (*bool, optional [default: False]*) – Whether to require the BED file to be sorted.
- **chromsizes** (*dict or pd.Series, optional*) – Chromosome sizes to validate against.
- **strict_score** (*bool, optional [default: False]*) – Whether to strictly enforce validation of the score field (0-1000).
- **replace_na** (*bool, optional [default: True]*) – Whether to replace null values of standard BED fields with compliant uninformative values.
- **na_rep** (*str, optional [default: "nan"]*) – String representation of null values if written.

Returns

The serialized BED file as a string if `path` is `None`, otherwise `None`.

Return type

str or None

CHAPTER

FIFTEEN

RESOURCES

15.1 Genome assembly metadata

Bioframe provides a collection of genome assembly metadata for commonly used genomes. These are accessible through a convenient dataclass interface via `bioframe.assembly_info()`.

The assemblies are listed in a manifest YAML file, and each assembly has a mandatory companion file called *seqinfo* that contains the sequence names, lengths, and other information. The records in the manifest file contain the following fields:

- `organism`: the organism name
- `provider`: the genome assembly provider (e.g, ucsc, ncbi)
- `provider_build`: the genome assembly build name (e.g., hg19, GRCh37)
- `release_year`: the year of the assembly release
- `seqinfo`: path to the seqinfo file
- `cytobands`: path to the cytoband file, if available
- `default_roles`: default molecular roles to include from the seqinfo file
- `default_units`: default assembly units to include from the seqinfo file
- `url`: URL to where the corresponding sequence files can be downloaded

The *seqinfo* file is a TSV file with the following columns (with header):

- `name`: canonical sequence name
- `length`: sequence length
- `role`: role of the sequence or scaffold (e.g., "assembled", "unlocalized", "unplaced")
- `molecule`: name of the molecule that the sequence belongs to, if placed
- `unit`: assembly unit of the chromosome (e.g., "primary", "non-nuclear", "decoy")
- `aliases`: comma-separated list of aliases for the sequence name

We currently do not include sequences with "alt" or "patch" roles in *seqinfo* files, but we do support the inclusion of additional decoy sequences (as used by so-called NGS *analysis sets* for human genome assemblies) by marking them as members of a "decoy" assembly unit.

The *cytoband* file is an optional TSV file with the following columns (with header):

- `chrom`: chromosome name
- `start`: start position

- end: end position
- band: cytogenetic coordinate (name of the band)
- stain: Giesma stain result

The order of the sequences in the *seqinfo* file is treated as canonical. The ordering of the chromosomes in the *cytobands* file should match the order of the chromosomes in the *seqinfo* file.

The manifest and companion files are stored in the bioframe/io/data directory. New assemblies can be requested by opening an issue on GitHub or by submitting a pull request. **Functions:**

assemblies_available()	Get a list of available genome assembly metadata in local storage.
assembly_info(name[, roles, units])	Get information about a genome assembly.

assemblies_available() → DataFrame

Get a list of available genome assembly metadata in local storage.

Returns
A dataframe with metadata fields for available assemblies, including 'provider', 'provider_build', 'default_roles', 'default_units', and names of seqinfo and cytoband files.

Return type
pandas.DataFrame

assembly_info(*name: str*, *roles: List | Tuple | Literal['all'] | None = None*, *units: List | Tuple | Literal['all'] | None = None*) → *GenomeAssembly*

Get information about a genome assembly.

Parameters

- **name** (*str*) – Name of the assembly. If the name contains a dot, it is interpreted as a provider name and a build, e.g. "hg38". Otherwise, the provider is inferred if the build name is unique.
- **roles** (*list or tuple or "all", optional*) – Sequence roles to include in the assembly info. If not specified, only sequences with the default sequence roles for the assembly are shown. e.g. "assembled", "unlocalized", "unplaced"
- **units** (*list or tuple or "all", optional*) – Assembly units to include in the assembly info. If not specified, only sequences from the default units for the assembly are shown. e.g. "primary", "non-nuclear", "decoy"

Returns
A dataclass containing information about the assembly.

Return type
GenomeAssembly

Raises
ValueError – If the assembly name is not found or is not unique.

Examples

```
>>> hg38 = assembly_info("hg38")
>>> hg38.chromsizes
name
chr1    248956422
chr2    242193529
chr3    198295559
...     ...
```

```
>>> assembly_info("hg38", roles=("assembled", "non-nuclear"))
```

```
>>> assembly_info("ucsc.hg38", units=("unplaced",))
```

class GenomeAssembly(*organism: str*, *provider: str*, *provider_build: str*, *release_year: str*, *seqinfo: DataFrame*, *cytobands: DataFrame | None = None*, *url: str | None = None*, *alias_dict: Dict[str, str] | None = None*)

A dataclass containing information about sequences in a genome assembly.

alias_dict: Dict[str, str] = None

property chromnames: List[str]

property chromsizes: Series

cytobands: DataFrame = None

organism: str

provider: str

provider_build: str

release_year: str

seqinfo: DataFrame

url: str = None

property viewframe: DataFrame

15.2 Remote resources

These functions now default to using the local data store, but can be used to obtain chromsizes or centromere positions from UCSC by setting `provider="ucsc"`. **Functions:**

fetch_centromeres(db[, provider])	Extract centromere locations for a given assembly 'db' from a variety of file formats in UCSC (cytoband, centromeres) depending on availability, returning a DataFrame.
fetch_chromsizes(db, *[, provider, as_bed, ...])	Fetch chromsizes from local storage or the UCSC database.

fetch_centromeres(*db: str*, *provider: str = 'local'*) → DataFrame

Extract centromere locations for a given assembly 'db' from a variety of file formats in UCSC (cytoband, centromeres) depending on availability, returning a DataFrame.

Parameters

- **db** (*str*) – Assembly name.
- **provider** (*str, optional [default: "local"]*) – The provider of centromere data. Either "local" for local storage or "ucsc".

Return type
DataFrame with centromere 'chrom', 'start', 'end', 'mid'.

Notes

When provider="local", centromeres are derived from cytoband tables in local storage.

Whe provider="ucsc", the fallback priority goes as follows: - UCSC cytoBand - UCSC cytoBandIdeo - UCSC centromeres.txt

Note that UCSC "gap" files no longer provide centromere information.

Currently only works for human assemblies.

See also:

`bioframe.assembly_info`, `bioframe.UCSCClient`

fetch_chromsizes(*db: str*, ***, *provider: str = 'local'*, *as_bed: bool = False*, *filter_chroms: bool = True*, *chrom_patterns: tuple = ('^chr[0-9]+$', '^chr[XY]$', '^chrM$')*, *natsort: bool = True*, ***kwargs*) → Series | DataFrame

Fetch chromsizes from local storage or the UCSC database.

Parameters

- **db** (*str*) – Assembly name.
- **provider** (*str, optional [default: "local"]*) – The provider of chromsizes. Either "local" for local storage or "ucsc".
- **as_bed** (*bool, optional*) – If True, return chromsizes as an interval DataFrame (chrom, start, end) instead of a Series.
- **provider="ucsc".** (*The remaining options only apply to*)
- **filter_chroms** (*bool, optional*) – Filter for chromosome names given in `chrom_patterns`.
- **chrom_patterns** (*sequence, optional*) – Sequence of regular expressions to capture desired sequence names.
- **natsort** (*bool, optional*) – Sort each captured group of names in natural order. Default is True.
- ****kwargs** – Passed to `pandas.read_csv()`

Return type
Series of integer bp lengths indexed by sequence name or BED3 DataFrame.

Notes

For more fine-grained control over the chromsizes from local storage, use `bioframe.assembly_info()`.

Examples

```
>>> fetch_chromsizes("hg38")
name
chr1     248956422
chr2     242193529
chr3     198295559
...      ...
chrX     156040895
chrY      57227415
chrM         16569
Name: length, dtype: int64
```

```
>>> fetch_chromsizes("hg38", as_bed=True)
        chrom      start        end
0        chr1          0  248956422
1        chr2          0  242193529
2        chr3          0  198295559
...      ...
21       chrX          0  156040895
22       chrY          0   57227415
23       chrM          0      16569
```

See also:

`bioframe.assembly_info`, `bioframe.UCSCClient`

CHAPTER

SIXTEEN

ADDITIONAL TOOLS

Functions:

binnify(chromsizes, binsize[, rel_ids])	Divide a genome into evenly sized bins.
digest(fasta_records, enzyme)	Divide a genome into restriction fragments.
frac_gc(df, fasta_records[, mapped_only, ...])	Calculate the fraction of GC basepairs for each interval in a dataframe.
frac_gene_coverage(df, ucsc_mrna)	Calculate number and fraction of overlaps by predicted and verified RNA isoforms for a set of intervals stored in a dataframe.
frac_mapped(df, fasta_records[, return_input])	Calculate the fraction of mapped base-pairs for each interval in a dataframe.
make_chromarms(chromsizes, midpoints[, ...])	Split chromosomes into chromosome arms.
pair_by_distance(df, min_sep, max_sep[, ...])	From a dataframe of genomic intervals, find all unique pairs of intervals that are between `min_sep` and `max_sep` bp separated from each other.
seq_gc(seq[, mapped_only])	Calculate the fraction of GC basepairs for a string of nucleotides.

binnify(*chromsizes*, *binsize*, *rel_ids=False*)

Divide a genome into evenly sized bins.

Parameters

- **chromsizes** (*Series*) – pandas Series indexed by chromosome name with chromosome lengths in bp.
- **binsize** (*int*) – size of bins in bp

Returns
bintable

Return type
pandas.DataFrame with columns: 'chrom', 'start', 'end'.

digest(*fasta_records*, *enzyme*)

Divide a genome into restriction fragments.

Parameters

- **fasta_records** (*OrderedDict*) – Dictionary of chromosome names to sequence records. Created by: bioframe.load_fasta('/path/to/fasta.fa')
- **enzyme** (*str*) – Name of restriction enzyme.

Returns
Dataframe with columns

Return type
'chrom', 'start', 'end'.

frac_gc(*df*, *fasta_records*, *mapped_only=True*, *return_input=True*)

Calculate the fraction of GC basepairs for each interval in a dataframe.

Parameters

- **df** (*pandas.DataFrame*) – A sets of genomic intervals stored as a DataFrame.
- **fasta_records** (*OrderedDict*) – Dictionary of chromosome names to sequence records. Created by: bioframe.load_fasta('/path/to/fasta.fa')
- **mapped_only** (*bool*) – if True, ignore 'N' in the fasta_records for calculation. if True and there are no mapped base-pairs in an interval, return np.nan.
- **return_input** (*bool*) – if False, only return Series named frac_mapped.

Returns
df_mapped – Original dataframe with new column 'GC' appended.

Return type
pd.DataFrame

frac_gene_coverage(*df*, *ucsc_mrna*)

Calculate number and fraction of overlaps by predicted and verified RNA isoforms for a set of intervals stored in a dataframe.

Parameters

- **df** (*pd.DataFrame*) – Set of genomic intervals stored as a dataframe.
- **ucsc_mrna** (*str or DataFrame*) – Name of UCSC genome or all_mrna.txt dataframe from UCSC or similar.

Returns
df_gene_coverage

Return type
pd.DataFrame

frac_mapped(*df*, *fasta_records*, *return_input=True*)

Calculate the fraction of mapped base-pairs for each interval in a dataframe.

Parameters

- **df** (*pandas.DataFrame*) – A sets of genomic intervals stored as a DataFrame.
- **fasta_records** (*OrderedDict*) – Dictionary of chromosome names to sequence records. Created by: bioframe.load_fasta('/path/to/fasta.fa')
- **return_input** (*bool*) – if False, only return Series named frac_mapped.

Returns
df_mapped – Original dataframe with new column 'frac_mapped' appended.

Return type
pd.DataFrame

make_chromarms(*chromsizes*, *midpoints*, *cols_chroms=('chrom', 'length')*, *cols_mids=('chrom', 'mid')*, *suffixes=('_p', '_q')*)

Split chromosomes into chromosome arms.

Parameters

- **chromsizes** (*pandas.Dataframe or dict-like*) – If dict or pandas.Series, a map from chromosomes to lengths in bp. If pandas.Dataframe, a dataframe with columns defined by cols_chroms. If cols_chroms is a triplet (e.g. 'chrom','start','end'), then values in chromsizes[cols_chroms[1]].values must all be zero.
- **midpoints** (*pandas.Dataframe or dict-like*) – Mapping of chromosomes to midpoint (aka centromere) locations. If dict or pandas.Series, a map from chromosomes to midpoints in bp. If pandas.Dataframe, a dataframe with columns defined by cols_mids.
- **cols_chroms** (*(str, str) or (str, str, str)*) – Two columns
- **suffixes** (*tuple, optional*) – Suffixes to name chromosome arms. Defaults to p and q.

Returns

4-column BED-like DataFrame (chrom, start, end, name). Arm names are chromosome names + suffix. Any chromosome not included in `mids` will be not be split.

Return type

df_chromarms

pair_by_distance(*df*, *min_sep*, *max_sep*, *min_intervening=None*, *max_intervening=None*, *relative_to='midpoints'*, *cols=None*, *return_index=False*, *keep_order=False*, *suffixes=('_1', '_2')*)

From a dataframe of genomic intervals, find all unique pairs of intervals that are between `min_sep` and `max_sep` bp separated from each other.

Parameters

- **df** (*pandas.DataFrame*) – A BED-like dataframe.
- **min_sep** (*int*) – Minimum and maximum separation between intervals in bp. Min > 0 and Max >= Min.
- **max_sep** (*int*) – Minimum and maximum separation between intervals in bp. Min > 0 and Max >= Min.
- **min_intervening** (*int*) – Minimum and maximum number of intervening intervals separating pairs. Min > 0 and Max >= Min.
- **max_intervening** (*int*) – Minimum and maximum number of intervening intervals separating pairs. Min > 0 and Max >= Min.
- **relative_to** (*str,*) – Whether to calculate distances between interval "midpoints" or "endpoints". Default "midpoints".
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.
- **return_index** (*bool*) – If True, return indicies of pairs as two new columns ('index'+suffixes[0] and 'index'+suffixes[1]). Default False.
- **keep_order** (*bool, optional*) – If True, sort the output dataframe to preserve the order of the intervals in df1. Default False. Note that it relies on sorting of index in the original dataframes, and will reorder the output by index.

- **suffixes** (*(str, str), optional*) – The column name suffixes for the two interval sets in the output. The first interval of each output pair is always upstream of the second.

Returns
A BEDPE-like dataframe of paired intervals from `df`.

Return type
pandas.DataFrame

seq_gc(*seq*, *mapped_only=True*)

Calculate the fraction of GC basepairs for a string of nucleotides.

Parameters

- **seq** (*str*) – Basepair input
- **mapped_only** (*bool*) – if True, ignore 'N' in the sequence for calculation. if True and there are no mapped base-pairs, return np.nan.

Returns
gc – calculated gc content.

Return type
float

CHAPTER

SEVENTEEN

PLOTTING

Functions:

plot_intervals(df[, levels, labels, colors, ...])	Plot a collection of intervals, one plot per chromosome.
to_ucsc_colorstring(color)	Convert any matplotlib color identifier into a UCSC itemRgb color string.

plot_intervals(*df*, *levels=None*, *labels=None*, *colors=None*, *xlim=None*, *show_coords=False*, *figsize=(10, 2)*)

Plot a collection of intervals, one plot per chromosome.

Parameters

- **df** (*pandas.DataFrame*) – A collection of intervals.
- **levels** (*iterable or None*) – The level of each interval, i.e. the y-coordinate at which the interval must be plotted. If None, it will be determined automatically.
- **labels** (*str or iterable or None*) – The label of each interval.
- **colors** (*str or iterable or None.*) – The color of each interval.
- **xlim** (*(float, float) or None*) – The x-span of the plot.
- **show_coords** (*bool*) – If True, plot x-ticks.
- **figsize** (*(float, float) or None.*) – The size of the figure. If None, plot within the current figure.

to_ucsc_colorstring(*color: str | tuple*) → str

Convert any matplotlib color identifier into a UCSC itemRgb color string.

Parameters

color (*str or tuple*) – Any valid matplotlib color representation (e.g. 'red', 'tomato', '#ff0000', '#ff00', "#ff000055", (1, 0, 0), (1, 0, 0, 0.5))

Returns

A UCSC itemRgb colorstring of the form "r,g,b" where r, g, and b are integers between 0 and 255, inclusive.

Return type

str

Notes

The alpha (opacity) channel is ignored if represented in the input.

Null values are converted to “0”, which is shorthand for “0,0,0” (black). Note that BED9+ files with uninformative itemRgb values should use “0” as the itemRgb value on every data line.

Examples

```
>>> to_ucsc_colorstring("red")
'255,0,0'
>>> to_ucsc_colorstring("tomato")
'255,99,71'
>>> df["itemRgb"] = df["color"].apply(to_ucsc_colorstring)
>>> df
chrom  start  end  color  itemRgb
chr1   0      10   red    255,0,0
chr1   10     20   blue   0,0,255
chr2   0      10   green  0,128,0
chr2   10     20   None   0
```

CHAPTER

EIGHTEEN

LOW-LEVEL API

18.1 Array operations

Low level operations that are used to implement the genomic interval operations. **Functions:**

arange_multi(starts[, stops, lengths])	Create concatenated ranges of integers for multiple start/length.
closest_intervals(starts1, ends1[, starts2, ...])	For every interval in set 1, return the indices of k closest intervals from set 2.
interweave(a, b)	Interweave two arrays.
merge_intervals(starts, ends[, min_dist])	Merge overlapping intervals.
overlap_intervals(starts1, ends1, starts2, ends2)	Take two sets of intervals and return the indices of pairs of overlapping intervals.
overlap_intervals_outer(starts1, ends1, ...)	Take two sets of intervals and return the indices of pairs of overlapping intervals, as well as the indices of the intervals that do not overlap any other interval.
sum_slices(arr, starts, ends)	Calculate sums of slices of an array.

arange_multi(*starts*, *stops=None*, *lengths=None*)

Create concatenated ranges of integers for multiple start/length.

Parameters

- **starts** (*numpy.ndarray*) – Starts for each range
- **stops** (*numpy.ndarray*) – Stops for each range
- **lengths** (*numpy.ndarray*) – Lengths for each range. Either stops or lengths must be provided.

Returns

concat_ranges – Concatenated ranges.

Return type

numpy.ndarray

Notes

See the following illustrative example:

starts = np.array([1, 3, 4, 6]) stops = np.array([1, 5, 7, 6])

print arange_multi(starts, lengths) >>> [3 4 4 5 6]

From: https://codereview.stackexchange.com/questions/83018/vectorized-numpy-version-of-arange-with-multiple-start-stop

closest_intervals(*starts1*, *ends1*, *starts2=None*, *ends2=None*, *k=1*, *tie_arr=None*, *ignore_overlaps=False*, *ignore_upstream=False*, *ignore_downstream=False*, *direction=None*)

For every interval in set 1, return the indices of k closest intervals from set 2.

Parameters

- **starts1** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored. If start2 and ends2 are None, find closest intervals within the same set.
- **ends1** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored. If start2 and ends2 are None, find closest intervals within the same set.
- **starts2** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored. If start2 and ends2 are None, find closest intervals within the same set.
- **ends2** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored. If start2 and ends2 are None, find closest intervals within the same set.
- **k** (*int*) – The number of neighbors to report.
- **tie_arr** (*numpy.ndarray or None*) – Extra data describing intervals in set 2 to break ties when multiple intervals are located at the same distance. Intervals with *lower* tie_arr values will be given priority.
- **ignore_overlaps** (*bool*) – If True, ignore set 2 intervals that overlap with set 1 intervals.
- **ignore_upstream** (*bool*) – If True, ignore set 2 intervals upstream/downstream of set 1 intervals.
- **ignore_downstream** (*bool*) – If True, ignore set 2 intervals upstream/downstream of set 1 intervals.
- **direction** (*numpy.ndarray with dtype bool or None*) – Strand vector to define the upstream/downstream orientation of the intervals.

Returns

closest_ids – An Nx2 array containing the indices of pairs of closest intervals. The 1st column contains ids from the 1st set, the 2nd column has ids from the 2nd set.

Return type

numpy.ndarray

interweave(*a*, *b*)

Interweave two arrays.

Parameters

- **a** (*numpy.ndarray*) – Arrays to interweave, must have the same length/
- **b** (*numpy.ndarray*) – Arrays to interweave, must have the same length/

Returns
out – Array of interweaved values from a and b.

Return type
numpy.ndarray

Notes

From https://stackoverflow.com/questions/5347065/interweaving-two-numpy-arrays

`merge_intervals`(*starts*, *ends*, *min_dist=0*)

Merge overlapping intervals.

Parameters

- **starts** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored.
- **ends** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored.
- **min_dist** (*float or None*) – If provided, merge intervals separated by this distance or less. If None, do not merge non-overlapping intervals. Using min_dist=0 and min_dist=None will bring different results. bioframe uses semi-open intervals, so interval pairs [0,1) and [1,2) do not overlap, but are separated by a distance of 0. Such intervals are not merged when min_dist=None, but are merged when min_dist=0.

Returns

- **cluster_ids** (*numpy.ndarray*) – The indices of interval clusters that each interval belongs to.
- **cluster_starts** (*numpy.ndarray*)
- **cluster_ends** (*numpy.ndarray*) – The spans of the merged intervals.

Notes

From https://stackoverflow.com/questions/43600878/merging-overlapping-intervals/58976449#58976449

`overlap_intervals`(*starts1*, *ends1*, *starts2*, *ends2*, *closed=False*, *sort=False*)

Take two sets of intervals and return the indices of pairs of overlapping intervals.

Parameters

- **starts1** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored.
- **ends1** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored.
- **starts2** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored.
- **ends2** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored.
- **closed** (*bool*) – If True, then treat intervals as closed and report single-point overlaps.

Returns
overlap_ids – An Nx2 array containing the indices of pairs of overlapping intervals. The 1st column contains ids from the 1st set, the 2nd column has ids from the 2nd set.

Return type
numpy.ndarray

overlap_intervals_outer(*starts1*, *ends1*, *starts2*, *ends2*, *closed=False*)

Take two sets of intervals and return the indices of pairs of overlapping intervals, as well as the indices of the intervals that do not overlap any other interval.

Parameters

- **starts1** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored.
- **ends1** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored.
- **starts2** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored.
- **ends2** (*numpy.ndarray*) – Interval coordinates. Warning: if provided as pandas.Series, indices will be ignored.
- **closed** (*bool*) – If True, then treat intervals as closed and report single-point overlaps.

Returns

- **overlap_ids** (*numpy.ndarray*) – An Nx2 array containing the indices of pairs of overlapping intervals. The 1st column contains ids from the 1st set, the 2nd column has ids from the 2nd set.
- **no_overlap_ids1, no_overlap_ids2** (*numpy.ndarray*) – Two 1D arrays containing the indices of intervals in sets 1 and 2 respectively that do not overlap with any interval in the other set.

sum_slices(*arr*, *starts*, *ends*)

Calculate sums of slices of an array.

Parameters

- **arr** (*numpy.ndarray*)
- **starts** (*numpy.ndarray*) – Starts for each slice
- **ends** (*numpy.ndarray*) – Stops for each slice

Returns
sums – Sums of the slices.

Return type
numpy.ndarray

18.2 Specifications

Functions:

is_chrom_dtype(chrom_dtype)	Returns True if dtype is any of the allowed bioframe chrom dtypes, False otherwise.

is_chrom_dtype(*chrom_dtype*)

Returns True if dtype is any of the allowed bioframe chrom dtypes, False otherwise.

Unexposed functions:

specs.**_verify_column_dtypes**(*cols=None*, *return_as_bool=False*)

Checks that dataframe *df* has chrom, start, end columns with valid dtypes. Raises TypeErrors if cols have invalid dtypes.

Parameters

- **df** (*pandas.DataFrame*)
- **cols** (*(str, str, str) or None*) – The names of columns containing the chromosome, start and end of the genomic intervals, provided separately for each set. The default values are 'chrom', 'start', 'end'.
- **return_as_bool** (*bool*) – If true, returns as a boolean instead of raising errors. Default False.

specs.**_verify_columns**(*colnames*, *unique_cols=False*, *return_as_bool=False*)

Raises ValueError if columns with colnames are not present in dataframe df.

Parameters

- **df** (*pandas.DataFrame*)
- **colnames** (*list of column names*)
- **return_as_bool** (*bool*) – If True, returns as a boolean instead of raising errors. Default False.

specs.**_get_default_colnames**()

Returns default column names.

These defaults be updated with `update_default_colnames()`.

Returns
colnames

Return type
triplet (str, str, str)

18.3 String operations

Functions:

is_complete_ucsc_string(s)	Returns True if a string can be parsed into a completely informative (chrom, start, end) format.
parse_region(grange[, chromsizes, check_bounds])	Coerce a genomic range string or sequence type into a triple.
parse_region_string(s)	Parse a UCSC-style genomic range string into a triple.
to_ucsc_string(grange)	Convert a grange to a UCSC string.

is_complete_ucsc_string(*s: str*) → bool

Returns True if a string can be parsed into a completely informative (chrom, start, end) format.

Parameters
s (*str*)

Returns
True if able to be parsed and end is known.

Return type
bool

parse_region(*grange: str | tuple*, *chromsizes: dict | Series | None = None*, ***, *check_bounds: bool = True*) → Tuple[str, int, int]

Coerce a genomic range string or sequence type into a triple.

Parameters

- **grange** (*str or tuple*) –
  - A UCSC-style genomic range string, e.g. "chr5:10,100,000-30,000,000".
  - A triple (chrom, start, end), where `start` or `end` may be `None`.
  - A quadruple or higher-order tuple, e.g. (chrom, start, end, name). `name` and other fields will be ignored.
- **chromsizes** (*dict or Series, optional*) – Lookup table of sequence lengths for bounds checking and for filling in a missing end coordinate.
- **check_bounds** (*bool, optional [default: True]*) – If True, check that the genomic range is within the bounds of the sequence.

Returns
A well-formed genomic range triple (str, int, int).

Return type
tuple

Notes

Genomic ranges are interpreted as half-open intervals (0-based starts, 1-based ends) along the length coordinate of a sequence.

Sequence names may contain any character except for whitespace and colon.

The start coordinate should be 0 or greater and the end coordinate should be less than or equal to the length of the sequence, if the latter is known. These are enforced when `check_bounds` is `True`.

If the start coordinate is missing, it is assumed to be 0. If the end coordinate is missing and chromsizes are provided, it is replaced with the length of the sequence.

The end coordinate **must** be greater than or equal to the start.

The start and end coordinates may be suffixed with k(b), M(b), or G(b) multipliers, case-insentive. e.g. "chr1:1K-2M" is equivalent to "chr1:1000-2000000".

parse_region_string(*s: str*) → Tuple[str, int, int]

Parse a UCSC-style genomic range string into a triple.

Parameters
s (*str*) – UCSC-style genomic range string, e.g. "chr5:10,100,000-30,000,000".

Returns
(str, int or None, int or None)

Return type
tuple

See also:

parse_region

to_ucsc_string(*grange: Tuple[str, int, int]*) → str

Convert a grange to a UCSC string.

Parameters
grange (*tuple or other iterable*) – chrom, start, end

Returns
UCSC-style genomic range string, '{chrom}:{start}-{end}'

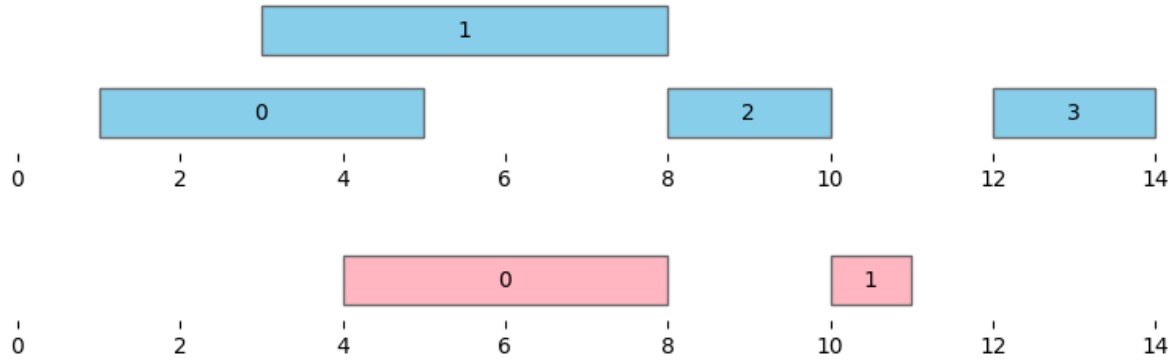
Return type
str

Low level array-based operations are used to implement the genomic interval operations on dataframes.

```
import itertools

import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import pandas as pd

import bioframe as bf
import bioframe.vis


from bioframe.core import arrops
```

```
starts1, ends1 = np.array([
    [1,5],
    [3,8],
    [8,10],
    [12,14]
]).T

starts2, ends2 = np.array([
    [4,8],
    [10,11],
]).T
```

```
bf.vis.plot_intervals_arr(
    starts = starts1,
    ends = ends1,
    xlim = (-0.5,14.5),
    labels = np.arange(0,starts1.shape[0]),
    show_coords = True)

bf.vis.plot_intervals_arr(
    starts = starts2,
    ends = ends2,
    colors = 'lightpink',
    xlim = (-0.5,14.5),
    labels = np.arange(0,starts2.shape[0]),
    show_coords = True)
```




```
arrops.overlap_intervals(starts1, ends1, starts2, ends2)
```

```
array([[0, 0],
       [1, 0]])
```

```
arrops.overlap_intervals_outer(starts1, ends1, starts2, ends2)
```

```
(array([[0, 0],
        [1, 0]]),
 array([2, 3]),
 array([1]))
```

```
arrops.merge_intervals(starts1, ends1, min_dist=0)
```

```
(array([0, 0, 0, 1]), array([ 1, 12]), array([10, 14]))
```

```
arrops.merge_intervals(starts1, ends1, min_dist=None)
```

```
(array([0, 0, 1, 2]), array([ 1,  8, 12]), array([ 8, 10, 14]))
```

```
arrops.merge_intervals(starts1, ends1, min_dist=2)
```

```
(array([0, 0, 0, 0]), array([1]), array([14]))
```

```
arrops.complement_intervals(starts1, ends1)
```

```
(array([ 0, 10, 14]),
 array([                  1,                  12, 9223372036854775807]))
```

CHAPTER

NINETEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

Symbols

A

B

C

D

E

F

G

I