GFFutils Documentation

Release 0.5

Ryan Dale

March 21, 2010

CONTENTS

Contents:

CHAPTER

ONE

OVERVIEW

This module is used for doing things with GFF files that are too complicated for a simple `awk` or `grep` command line call.

For example, to get a BED file of genes from a GFF file, you can use something simple like:

```
grep "gene" chr2.gff | awk '{print $1,$4,$5}'
```

But how would you use commandline tools to get a BED file of 3' exons from genes longer than 5 kb? Or how would you get the average number of isoforms for genes on the plus strand? These more complex questions are actually quite easy to answer using `GFFutils` – see the Examples for how it's done.

CHAPTER

TWO

INSTALLATION

Unzip the source code, and from the source directory run (with root priveliges):

```
python setup.py install
```

Now you're ready to create a GFF database and interact with it from a Python shell like IPython.

CHAPTER

THREE

CREATING A GFFDB

For each GFF file you would like to use you need to create a GFF database. This database is stored in a file, and is simply a sqlite3 database. You only have to do this once for each GFF file. As long as you don't delete the database from your hard drive, you don't have to do this time-consuming step again.

The database will take roughly twice as much hard drive space as the original text file. This is the cost of interactivity and fast lookups.

```
import GFFutils

# downloaded from, e.g., FlyBase
gff_filename = '/data/annotations/dm3.gff'

# the database that will be created
db_filename = '/data/dm3.db'

# do it! (this will take ~2 min to run)
GFFutils.create_gffdb(gfffn, db_filename)
```

Now `dm3.db` is the sqlite3 database that can be used in all sorts of weird and wonderful ways, outlined below.

CHAPTER

FOUR

USING THE DATABASE INTERACTIVELY

```
import GFFutils

# Set up a GFFDB object, telling it the filename of your database as
# created above
G = GFFutils.GFFDB('dm3.db')
```

For performance, most of the `GFFDB` class methods return iterators. In practice, you will need to either convert them to a list or iterate through them in a list comprehension or a for-loop. You can also grab the next item in an iterator with its `.next()` method. All four ways of getting info from an iterator are shown below in the examples.

4.1 Identifying what's in the database

What sorts of features are in the db? The `GFFDB.features()` method returns an iterator of the featuretypes that were in the GFF file (and which are now in the `featuretype` field of the sqlite3 database, which this method accesses).

Most methods in a `GFFDB` object return iterators for performance.

Since this is the first example of using the iterators returned by a `GFFDB` object, here are a few different ways to get the results from the iterator it returns.

Method 0: Convert iterator to a list:

```
featuretype_iterator = G.features()
featuretypes = list(featuretype_iterator)
```

Method 1: Use iterator in a for-loop (preferred):

```
featuretype_iterator = G.features()
for featuretype in featuretype_iterator:
    print featuretype
```

Method 2: Call `next()` incrementally on the iterator:

```
featuretype_iterator = G.features()
featuretype_1 = featuretype_iterator.next()
featuretype_2 = featuretype_iterator.next()
featuretype_3 = featuretype_iterator.next()
```

```
featuretype_4 = featuretype_iterator.next()
...
...

featuretypes = [featuretype1, featuretype2, ...]
```

It's mostly a matter of preference which method you use. However, using the for-loop approach is most memory-efficient, since only a single featuretype is in memory at one time. This is not too important for iterating through featuretypes (of which there are usually <50; typically 3-10). But when you want to iterate through 15,000 genes it can be useful.

In any case, we get something like the following. This depends entirely on the GFF file that you created your database from:

```
['BAC_cloned_genomic_insert',
 'CDS',
 'DNA_motif',
 'breakpoint',
 'chromosome_arm',
 'chromosome_band',
 'complex_substitution',
 'deletion',
 'enhancer',
 'exon',
 'five_prime_UTR',
 'gene',
 'insertion_site',
 'intron',
 ...
 ...
  'tRNA',
 'tandem_repeat',
 'three_prime_UTR',
 'transposable_element',
 'transposable_element_insertion_site',
 'uncharacterized_change_in_nucleotide_sequence']
```

4.2 Retrieving specific feature types

To retrieve just genes, just exons, or any other feature type that was in the GFF file, use the `GFFDB.features_of_type()` method. This will return an iterator of `GFFFeature` objects. These objects are described in more detail in another section below.

`'gene'` was in the list of `featuretypes` above. Let's find out how many genes there were:

```
gene_iterator = G.features_of_type('gene')

# convert iterator to list so we can get a length
gene_list = list(gene_iterator)

print len(gene_list)
```

Here's a more memory-efficient way to do the same thing. In this method, we're not bringing ALL the genes into a giant list – we'll just increment a counter. Only a single `GFFFeature` object is in memory at a time, which is the advantage of iterators . . .

```
gene_count = 0
for gene in G.features_of_type('gene'):
    gene_count += 1
print gene_count
```

Feature types not found in the db will not return an error (maybe they should, eventually?); they just don't return anything:

```
ncabbages = len(list(G.features_of_type('cabbage')))
print ncabbages  # zero cabbages.
```

Already know the ID of a feature? Get the `GFFFeature` object for that gene directly like this:

```
my_favorite_feature = G['FBgn0002121']
```

CHAPTER

FIVE

GFFFEATURES IN MORE DETAIL

Just to make sure we're on the same page, here's the setup for this section:

```
import GFFutils
G = GFFutils.GFFDB('dm3.db')
```

Let's get a single `GFFFeature` to work with. Since I don't know any accessions off the top of my head, let's just get the first gene in the iterator:

```
genes_iterator = G.features_of_type('gene')
gene = genes_iterator.next()
```

`GFFFeature` objects, when printed, show useful information:

```
GFFFeature gene 'FBgn0031208': chr2L:7529-9484 (+)
#           ^          ^               ^         ^
#           |          |               |         |
# featuretype      accession   genomic coords   strand
```

`GFFFeature` objects have an attribute, `id`, which contains the accession in the attributes field of the original GFF file:

```
print gene.id
```

```
'FBgn0031208'
```

They also have many other properties:

```
print gene.start
print gene.stop
print gene.chr
print gene.featuretype
print gene.strand
```

You can get the length of a gene with:

```
gene_len = gene.stop - gene.start
```

or you can use the perhaps-more-convenient:

```
gene_len = len(gene)
```

In a `GFFFeature` object, the `GFFFeature.attributes` attribute holds all the info that was in the attributes column of your GFF file. This will vary based on what was in your original GFF file. You can get a list of this with:

```
print gene.attributes._attrs
```

and you can access any of the attributes with a dot, then the attribute name. For example, in the GFF file I used, the above code returned:

```
['ID', 'Name', 'Ontology_term', 'Dbxref', 'derived_computed_cyto', 'gbunit']
```

So we could get the ontology terms for this gene with:

```
print gene.attributes.Ontology_term
```

Or the DBxref for the gene with:

```
print gene.attributes.Dbxref
```

You can parse this info out yourself; parsing these into sub-attributes of a `GFFFeature.Attribute` object is something I haven't implemented yet...

You now know enough to be able to generate a line for a BED-format file:

```
line = '%s\t%s\t%s\t%s\t%s\t%s\n' % (gene.chr,
                                     gene.start,
                                     gene.stop,
                                     gene.id,
                                     gene.value,
                                     gene.strand)
print line
```

But `GFFFeature` objects have a convenience function, `to_bed()`, which also accepts a number from 3 to 6 so you can tell it how many BED fields you want returned (3 fields is the default).

So you could write a BED file of all the genes like so:

```
fout = open('genes.bed','w')  # open a file for writing
for i in G.features_of_type('gene'):
    fout.write(i.to_bed())
fout.close()
```

This can be extremely useful for downstream processing by, for example, BEDtools.

CHAPTER

SIX

NAVIGATING THE HIERARCHY OF FEATURES

Here's how to find the transcripts belonging to a gene. The `GFFFeature.children()` and `GFFFeature.parents()` methods need a feature ID as an argument, which is stored in the `GFFFeature.id` attribute:

```
for i in G.children(gene.id):
    print i
```

Here's how to find the exons belonging to a gene. By default, level=1, which means a 'hierarchy distance' of 1 (direct parent/children). level=2 is analagous to grandparent/grandchild, which is used for the relationship between genes/exons. level=3 not currently implemented (not clear where it would be used):

```
for i in G.children(gene_name, level=2):
    print i
```

Note that, depending on your GFF file, you may have more than just exons as the children of genes (e.g., 3' UTRs, introns, 5' UTRs). If you just want the exons, then you can filter by feature type:

```
for i in G.children(gene.id, level=2):
    if i.featuretype == 'exon':
        print i
```

CHAPTER

SEVEN

FILE FORMAT CONVERSIONS

Converting features to BED files was described above; briefly:

```
fout = open('genes.bed','w')
for gene in G.features_of_type('gene'):
    fout.write(gene.to_bed())
fout.close()
```

Exporting a refFlat entry for one gene:

```
print G.refFlat(gene_name)
```

Create a new file, writing a refFlat entry for each gene. Note that the `refFlat()` method is set up such that it will return `None` if there were no CDSs for a particular gene. We don't want to write these to file, but do want to keep track of them.

This will take a few seconds to run:

```
missing_cds = []
fout = open('mydatabase.refFlat','w')
for gene in G.features_of_type('gene'):
    rflt = G.refFlat(gene.id)
    if rflt is not None:
        fout.write(rflt)
    else:
        missing_cds.append(gene)

fout.close()
```

So, what were those genes that didn't have CDSs? Check the first 25:

```
for g in missing_cds[:25]:
    print g.attributes.Name[0]
```

Ahhhhh . . . a bunch of snoRNAs, tRNAs, etc. Makes sense!

`GFFFeatures` have a `GFFFeature.tostring()` method which prints back the GFF file entry as a string (with the newline included). This makes it very easy to write new GFF files containing a subset of the features in the original GFF file:

```
# new GFF file with genes > 5kb
fout = open('big-genes.gff','w')
for gene in G.features_of_type('gene'):
```

```
    if len(gene) < 5000:
        fout.write(gene.tostring())
fout.close()
```

CHAPTER

EIGHT

EXAMPLES

In each case, assume the following setup:

```
import GFFutils
GFFutils.create_gffdb('dm3.gff','dm3.db')
G = GFFutils.GFFDB('dm3.db')
```

8.1 Gene count

```
gene_count = 0
for gene in G.features_of_type('three_prime_UTR'):
    gene_count += 1
print gene_count
```

8.2 Average gene length

```
gene_lengths = 0
gene_count = 0
for gene in G.features_of_type('gene'):
    gene_lengths += len(gene)
    gene_count += 1
mean_gene_length = float(gene_lengths) / gene_count
print mean_gene_length
```

8.3 Longest gene

```
maxlen = 0
for gene in G.features_of_type('gene'):
    gene_len = len(gene)
    if gene_len > maxlen:
        maxlen = gene_len
print maxlen
```

8.4 Average exon count

```
exon_count = 0
gene_count = 0
for gene in G.features_of_type('gene'):
    gene_exon_count = 0
    for child in G.children(gene.id,2):
        if child.featuretype == 'exon':
            gene_exon_count += 1
    exon_count += gene_exon_count
    gene_count += 1
mean_exon_count = float(exon_count) / gene_count
print mean_exon_count
```

8.5 BED file of 3' exons from genes longer than 5 kb

```
fout = open('3prime-exons.bed','w')
fout.write('track name="3-prime exons\n"')
for gene in G.features_of_type('gene'):
    if len(gene) < 5000:
        continue
    children = [i for i in G.children(gene.id,2) if i.featuretype=='exon']
    if gene.strand == '+':
        three_prime_exon = children[0]
    else:
        three_prime_exon = children[-1]
    fout.write(three_prime_exon.to_bed())
fout.close()
```

8.6 Histogram of exon lengths

(Assumes you have matplotlib installed)

```
from matplotlib import pyplot as p
lengths = [i.stop-i.start for i in G.features_of_type('exon')]
p.hist(lengths,bins=50)
p.show()
```

8.7 Average number of isoforms for genes on plus strand

```
isoform_count = 0
gene_count = 0
for gene in G.features_of_type('gene'):
    if gene.strand == '-':
        continue
    isoforms = [i for i in G.children(gene.id) if i.featuretype=='mRNA']
    isoform_count += len(isoforms)
```

```
    gene_count += 1
mean_isoform_count = float(isoform_count) / gene_count
```

CHAPTER

NINE

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*