PyCogent3 Documentation

Release 3.0a1

PyCogent3 Team

Sep 28, 2017

Contents

Contents

CHAPTER 1

Quick installation

Until the library is officially released, these installation instructions explicitly reference the bitbucket repository.

Using `pip`

Assuming you have pip installed on your system, the key steps for the minimal install are:

1. Install numpy

```
$ pip install numpy
```

2. Install PyCogent:

   (a) If you have mercurial installed:

```
$ DONT_USE_CYTHON=1 pip install hg+https://bitbucket.org/pycogent3/cogent3
```

   (b) If you don't have mercurial installed, download a zip file to your hard drive and pip install as:

```
$ DONT_USE_CYTHON=1 pip install /path/to/downloaded/archive.zip
```

Note: Use the `DONT_USE_CYTHON=1` if you want to be sure and use the `*.c` files we generated. If you don't have Cython installed, it has no effect.

Optional installs

To use the drawing and parallel computing capabilities, you will need to download a zip archive as indicated above. Then do:

```
$ pip install /path/to/downloaded/archive.zip[all]
```

Using conda

Support not yet in place, but it will be...

CHAPTER 2

The Readme

Download From bitbucket or follow the *Quick installation* instructions.

CHAPTER 3

Coding guidelines

As project size increases, consistency increases in importance. Unit testing and a consistent style are critical to having trusted code to integrate. Also, guesses about names and interfaces will be correct more often.

What should I call my variables?

- *Choose the name that people will most likely guess.* Make it descriptive, but not too long: `curr_record` is better than `c`, or `curr`, or `current_genbank_record_from_database`.
- *Good names are hard to find.* Don't be afraid to change names except when they are part of interfaces that other people are also using. It may take some time working with the code to come up with reasonable names for everything: if you have unit tests, it's easy to change them, especially with global search and replace.
- *Use singular names for individual things, plural names for collections.* For example, you'd expect `self.Name` to hold something like a single string, but `self.Names` to hold something that you could loop through like a list or dict. Sometimes the decision can be tricky: is `self.Index` an int holding a positon, or a dict holding records keyed by name for easy lookup? If you find yourself wondering these things, the name should probably be changed to avoid the problem: try `self.Position` or `self.LookUp`.
- *Don't make the type part of the name.* You might want to change the implementation later. Use `Records` rather than `RecordDict` or `RecordList`, etc. Don't use Hungarian Notation either (i.e. where you prefix the name with the type).
- *Make the name as precise as possible.* If the variable is the name of the input file, call it `infile_name`, not `input` or `file` (which you shouldn't use anyway, since they're keywords), and not `infile` (because that looks like it should be a file object, not just its name).
- *Use* `result` *to store the value that will be returned from a method or function.* Use `data` for input in cases where the function or method acts on arbitrary data (e.g. sequence data, or a list of numbers, etc.) unless a more descriptive name is appropriate.
- *One-letter variable names should only occur in math functions or as loop iterators with limited scope.* Limited scope covers things like `for k in keys: print k`, where `k` survives only a line or two. Loop iterators should refer to the variable that they're looping through: `for k in keys, i in items`, or `for key`

`in keys, item in items`. If the loop is long or there are several 1-letter variables active in the same scope, rename them.

- *Limit your use of abbreviations.* A few well-known abbreviations are OK, but you don't want to come back to your code in 6 months and have to figure out what `sptxck2` is. It's worth it to spend the extra time typing `species_taxon_check_2`, but that's still a horrible name: what's check number 1? Far better to go with something like `taxon_is_species_rank` that needs no explanation, especially if the variable is only used once or twice.

Acceptable abbreviations

The following list of abbreviations can be considered well-known and used with impunity within mixed name variables, but some should not be used by themselves as they would conflict with common functions, python built-in's, or raise an exception. Do not use the following by themselves as variable names: `dir`, `exp` (a common `math` module function), `in`, `max`, and `min`. They can, however, be used as part of a name, eg `matrix_exp`.

Full	Abbreviated
alignment	aln
archaeal	arch
auxillary	aux
bacterial	bact
citation	cite
current	curr
database	db
dictionary	dict
directory	dir
end of file	eof
eukaryotic	euk
frequency	freq
expected	exp
index	idx
input	in
maximum	max
minimum	min
mitochondrial	mt
number	num
observed	obs
original	orig
output	out
parameter	param
phylogeny	phylo
previous	prev
probability	prob
protein	prot
record	rec
reference	ref
sequence	seq
standard deviation	stdev
statistics	stats
string	str
structure	struct

Continued on next page

Table 3.1 – continued from previous page

Full	Abbreviated
temporary	temp
taxonomic	tax
variance	var

What are the naming conventions?

We aim to adhere, to a large extent, to PEP8!

How should I write comments?

- *Always update the docstring when the code changes.* Like outdated comments, outdated docstrings can waste a lot of time. "Correct examples are priceless, but incorrect examples are worse than worthless." Jim Fulton.

How should I format my code?

- *Use 4 spaces for indentation.* Do not use tabs (set your editor to convert tabs to spaces). The behaviour of tabs is not predictable across platforms, and will cause syntax errors. If we all use the same indentation, collaboration is much easier.
- *Lines should not be longer than 79 characters.* Long lines are inconvenient in some editors. Use \ for line continuation. Note that there cannot be whitespace after the \.
- *Blank lines should be used to highlight class and method definitions.* Separate class definitions by two blank lines. Separate methods by one blank line.

How should I test my code ?

Tests are an opportunity to invent the interface(s) you want. Write the test for a method before you write the method: often, this helps you figure out what you would want to call it and what parameters it should take. It's OK to write the tests a few methods at a time, and to change them as your ideas about the interface change. However, you shouldn't change them once you've told other people what the interface is.

Never treat prototypes as production code. It's fine to write prototype code without tests to try things out, but when you've figured out the algorithm and interfaces you must rewrite it *with tests* to consider it finished. Often, this helps you decide what interfaces and functionality you actually need and what you can get rid of.

"Code a little test a little". For production code, write a couple of tests, then a couple of methods, then a couple more tests, then a couple more methods, then maybe change some of the names or generalize some of the functionality. If you have a huge amount of code where 'all you have to do is write the tests', you're probably closer to 30% done than 90%. Testing vastly reduces the time spent debugging, since whatever went wrong has to be in the code you wrote since the last test suite. And remember to use python's interactive interpreter for quick checks of syntax and ideas.

Run the test suite when you change *anything*. Even if a change seems trivial, it will only take a couple of seconds to run the tests and then you'll be sure. This can eliminate long and frustrating debugging sessions where the change turned out to have been made long ago, but didn't seem significant at the time.

Some `unittest` pointers

- *Use the* `unittest` *framework with tests in a separate file for each module.* Name the test file `test_module_name.py`. Keeping the tests separate from the code reduces the temptation to change the tests when the code doesn't work, and makes it easy to verify that a completely new implementation presents the same interface (behaves the same) as the old.
- *Use* `evo.unit_test` *if you are doing anything with floating point numbers or permutations* (use `assertFloatEqual`). Do *not* try to compare floating point numbers using `assertEqual` if you value your sanity. `assertFloatEqualAbs` and `assertFloatEqualRel` can specifically test for absolute and relative differences if the default behavior is not giving you what you want. Similarly, `assertEqualItems`, `assertSameItems`, etc. can be useful when testing permutations.
- *Test the interface of each class in your code by defining at least one* `TestCase` *with the name* `ClassNameTests`. This should contain tests for everything in the public interface.
- *If the class is complicated, you may want to define additional tests with names* `ClassNameTests_test_type`. These might subclass `ClassNameTests` in order to share `setUp` methods, etc.
- *Tests of private methods should be in a separate* `TestCase` *called* `ClassNameTests_private`. Private methods may change if you change the implementation. It is not required that test cases for private methods pass when you change things (that's why they're private, after all), though it is often useful to have these tests for debugging.
- *Test 'all' the methods in your class.* You should assume that any method you haven't tested has bugs. The convention for naming tests is `test_method_name`. Any leading and trailing underscores on the method name can be ignored for the purposes of the test; however, *all tests must start with the literal substring* `test` *for* `unittest` *to find them.* If the method is particularly complex, or has several discretely different cases you need to check, use `test_method_name_suffix`, e.g. `test_init_empty`, `test_init_single`, `test_init_wrong_type`, etc. for testing `__init__`.
- *Write good docstrings for all your test methods.* When you run the test with the `-v` command-line switch for verbose output, the docstring for each test will be printed along with `...OK` or `...FAILED` on a single line. It is thus important that your docstring is short and descriptive, and makes sense in this context.

  Good docstrings:

```
NumberList.var should raise ValueError on empty or 1-item list
NumberList.var should match values from R if list has >2 items
NumberList.__init__ should raise error on values that fail float()
FrequencyDistribution.var should match corresponding NumberList var
```

  Bad docstrings:

```
var should calculate variance           # lacks class name, not␣
↪descriptive
Check initialization of a NumberList    # doesn't say what's expected
Tests of the NumberList initialization. # ditto
```

- *Module-level functions should be tested in their own* `TestCase`*, called* `modulenameTests`. Even if these functions are simple, it's important to check that they work as advertised.
- *It is much more important to test several small cases that you can check by hand than a single large case that requires a calculator.* Don't trust spreadsheets for numerical calculations – use R instead!
- *Make sure you test all the edge cases: what happens when the input is None, or '', or 0, or negative?* What happens at values that cause a conditional to go one way or the other? Does incorrect input raise the right exceptions? Can your code accept subclasses or superclasses of the types it expects? What happens with very large input?

- *To test permutations, check that the original and shuffled version are different, but that the sorted original and sorted shuffled version are the same.* Make sure that you get *different* permutations on repeated runs and when starting from different points.
- *To test random choices, figure out how many of each choice you expect in a large sample (say, 1000 or a million) using the binomial distribution or its normal approximation.* Run the test several times and check that you're within, say, 3 standard deviations of the mean.

CHAPTER 4

The data files used in the documentation

All data files referred to in the documentation can be download by clicking on the links below.

Note: not all browsers handle the file suffixes well, so you may need to rename the downloaded files in order to use them.

abglobin_aa.phylip

dists_for_phylo.pickle

long_testseqs.fasta

primate_brca1.fasta

primate_brca1.tree

primate_cdx2_promoter.fasta

test.paml

test.tree

test2.fasta

trna_profile.fasta

refseqs.fasta

refseqs_protein.fasta

inseqs.fasta

inseqs_protein.fasta

CHAPTER 5

PyCogent3 Usage Examples

A Note on the Computable Documentation

The following examples are all available as standalone text files which can be computed using the Python doctest module.

Data manipulation

Translating DNA into protein

Section author: Gavin Huttley

To translate a DNA alignment, read it in assigning the DNA alphabet. Note setting `aligned = False` is critical for loading sequences of unequal length. Different genetic codes are available in `cogent3.core.genetic_code`

```
>>> from cogent3 import LoadSeqs, DNA
>>> al = LoadSeqs('data/test2.fasta', moltype=DNA, aligned=False)
>>> pal = al.get_translation()
>>> print(pal.to_fasta())
>DogFaced
ARSQQNRWVETKETCNDRQT
>HowlerMon
ARSQHNRWAESEETCNDRQT
>Human
ARSQHNRWAGSKETCNDRRT
>Mouse
AVSQQSRWAASKGTCNDRQV
>NineBande
RQQSRWAESKETCNDRQT
```

To save this result to a file, use the `write` method.

Advanced sequence handling

Section author: Gavin Huttley

Individual sequences and alignments can be manipulated by annotations. Most value in the genome sequences arises from sequence annotations regarding specific sequence feature types, e.g. genes with introns / exons, repeat sequences. These can be applied to an alignment either using data formats available from genome portals (e.g. GFF, or GenBank annotation formats) or by custom assignments.

Annotations can be added in two ways: using either the `add_annotation` or the `add_feature` method. The distinction between these two is that `add_features` is more specialised. Features can be thought of as a type of annotation representing standard sequence properties eg introns/exons. Annotations are the more general case, such as a computed property which has, say a numerical value and a span.

For illustrative purposes we define a sequence with 2 exons and grab the 1st exon:

```
>>> from cogent3 import DNA
>>> s = DNA.make_seq("aagaagaagacccccaaaaaaaaaattttttttttaaaaaaaaaaaaa",
... name="Orig")
>>> exon1 = s.add_feature('exon', 'exon1', [(10,15)])
>>> exon2 = s.add_feature('exon', 'exon2', [(30,40)])
```

Here, 'exon' is the feature type, and 'exon#' the feature name. The feature type is used for the display formatting, which won't be illustrated here, and also for selecting all features of the same type, shown below.

We could also have created an annotation using the `add_annotation` method:

```
>>> from cogent3.core.annotation import Feature
>>> s2=DNA.make_seq("aagaagaagacccccaaaaaaaaaattttttttttaaaaaaaaaaaaa",
... name="Orig2")
>>> exon3 = s2.add_annotation(Feature, 'exon', 'exon1', [(35,40)])
```

We can use the features (eg `exon1`) to get the corresponding sequence region.

```
>>> s[exon1]
DnaSequence(CCCCC)
```

You can query annotations by type and optionally by label, receiving a list of features:

```
>>> exons = s.get_annotations_matching('exon')
>>> print(exons)
[exon "exon1" at [10:15]/48, exon "exon2" at [30:40]/48]
```

We can use this list to construct a pseudo-feature covering (or excluding) multiple features using `get_region_covering_all`. For instance, getting all exons,

```
>>> print(s.get_region_covering_all(exons))
region "exon" at [10:15, 30:40]/48
>>> s.get_region_covering_all(exons).get_slice()
DnaSequence(CCCCCTT... 15)
```

or not exons (the exon *shadow*):

```
>>> print(s.get_region_covering_all(exons).get_shadow().get_slice())
AAGAAGAAGAAAAAAAAAAATTTTTAAAAAAAA
```

The first of these essentially returns the CDS of the gene.

Features are themselves sliceable:

```
>>> exon1[0:3].get_slice()
DnaSequence(CCC)
```

This approach to sequence / alignment handling allows the user to manipulate them according to things they know about such as genes or repeat elements. Most of this annotation data can be obtained from genome portals.

The toolkit can perform standard sequence / alignment manipulations such as getting a subset of sequences or aligned columns, translating sequences, reading and writing standard formats.

Complete version of manipulating sequence annotations

Section author: Peter Maxwell, Gavin Huttley

A Sequence with a couple of exons on it.

```
>>> from cogent3 import DNA
>>> from cogent3.core.annotation import Feature
>>> s = DNA.make_seq("AAGAAGAAGACCCCCAAAAAAAAAATTTTTTTTTTAAAAAAAAAAAAA",
... name="Orig")
>>> exon1 = s.add_annotation(Feature, 'exon', 'fred', [(10,15)])
>>> exon2 = s.add_annotation(Feature, 'exon', 'trev', [(30,40)])
```

The corresponding sequence can be extracted either with slice notation or by asking the feature to do it, since the feature knows what sequence it belongs to.

```
>>> s[exon1]
DnaSequence(CCCCC)
>>> exon1.get_slice()
DnaSequence(CCCCC)
```

Usually the only way to get a `Feature` object like `exon1` is to ask the sequence for it. There is one method for querying annotations by type and optionally by name:

```
>>> exons = s.get_annotations_matching('exon')
>>> print(exons)
[exon "fred" at [10:15]/48, exon "trev" at [30:40]/48]
```

If the sequence does not have a matching feature you get back an empty list, and slicing the sequence with that returns a sequence of length 0.

```
>>> dont_exist = s.get_annotations_matching('dont_exist')
>>> dont_exist
[]
>>> s[dont_exist]
DnaSequence()
```

To construct a pseudo-feature covering (or excluding) multiple features, use `get_region_covering_all`:

```
>>> print(s.get_region_covering_all(exons))
region "exon" at [10:15, 30:40]/48
>>> print(s.get_region_covering_all(exons).get_shadow())
region "not exon" at [0:10, 15:30, 40:48]/48
```

eg: all the exon sequence:

```
>>> s.get_region_covering_all(exons).get_slice()
DnaSequence(CCCCCTT... 15)
```

or with slice notation:

```
>>> s[exon1, exon2]
DnaSequence(CCCCCTT... 15)
```

Though `.get_region_covering_all` also guarantees no overlaps within the result, slicing does not:

```
>>> print(s.get_region_covering_all(exons+exons))
region "exon" at [10:15, 30:40]/48
>>> s[exon1, exon1, exon1, exon1, exon1]
Traceback (most recent call last):
ValueError: Uninvertable. Overlap: 10 < 15
```

You can use features, maps, slices or integers, but non-monotonic slices are not allowed:

```
>>> s[15:20, 5:16]
Traceback (most recent call last):
ValueError: Uninvertable. Overlap: 15 < 16
```

Features are themselves sliceable:

```
>>> exon1[0:3].get_slice()
DnaSequence(CCC)
```

When sequences are concatenated they keep their (non-overlapping) annotations:

```
>>> c = s[exon1[4:]]+s
>>> print(len(c))
49
>>> for feat in  c.annotations:
...     print(feat)
...
exon "fred" at [-4-, 0:1]/49
exon "fred" at [11:16]/49
exon "trev" at [31:41]/49
```

Since features know their parents you can't use a feature from one sequence to slice another:

```
>>> print(c[exon1])
Traceback (most recent call last):
ValueError: Can't map exon "fred" at [10:15]/48 onto ...
```

Features are generally attached to the thing they annotate, but in those cases where a free-floating feature is created it can later be attached:

```
>>> len(s.annotations)
2
>>> region = s.get_region_covering_all(exons)
>>> len(s.annotations)
2
>>> region.attach()
>>> len(s.annotations)
3
>>> region.detach()
>>> len(s.annotations)
2
```

When dealing with sequences that can be reverse complemented (e.g. `DnaSequence`) features are **not** reversed. Features are considered to have strand specific meaning (.e.g CDS, exons) and so stay on their original strands. We

create a sequence with a CDS that spans multiple exons, and show that after getting the reverse complement we have exactly the same result from getting the CDS annotation.

```
>>> plus = DNA.make_seq("AAGGGGAAAACCCCCAAAAAAAAAATTTTTTTTTTAAA",
... name="plus")
>>> plus_cds = plus.add_annotation(Feature, 'CDS', 'gene',
...                                [(2,6),(10,15),(25,35)])
>>> print(plus_cds.get_slice())
GGGGCCCCCTTTTTTTTT
>>> minus = plus.rc()
>>> minus_cds = minus.get_annotations_matching('CDS')[0]
>>> print(minus_cds.get_slice())
GGGGCCCCCTTTTTTTTT
```

Sequence features can be accessed via a containing `Alignment`:

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs(data=[['x','-AAAAAAAAA'], ['y','TTTT--TTTT']], array_align=False)
>>> print(aln)
>x
-AAAAAAAAA
>y
TTTT--TTTT

>>> exon = aln.get_seq('x').add_annotation(Feature, 'exon', 'fred', [(3,8)])
>>> aln_exons = aln.get_annotations_from_seq('x', 'exon')
>>> aln_exons = aln.get_annotations_from_any_seq('exon')
```

But these will be returned as **alignment** features with locations in alignment coordinates.

```
>>> print(exon)
exon "fred" at [3:8]/9
>>> print(aln_exons[0])
exon "fred" at [4:9]/10
>>> print(aln_exons[0].get_slice())
>x
AAAAA
>y
--TTT

>>> aln_exons[0].attach()
>>> len(aln.annotations)
1
```

Similarly alignment features can be projected onto the aligned sequences, where they may end up falling across gaps:

```
>>> exons = aln.get_projected_annotations('y', 'exon')
>>> print(exons)
[exon "fred" at [-2-, 4:7]/8]
>>> print(aln.get_seq('y')[exons[0].map.without_gaps()])
TTT
```

We copy the annotations from another sequence,

```
>>> aln = LoadSeqs(data=[['x', '-AAAAAAAAA'], ['y', 'TTTT--CCCC']], array_align=False)
>>> s = DNA.make_seq("AAAAAAAAA", name="x")
>>> exon = s.add_annotation(Feature, 'exon', 'fred', [(3,8)])
>>> exon = aln.get_seq('x').copy_annotations(s)
```

```
>>> aln_exons = list(aln.get_annotations_from_seq('x', 'exon'))
>>> print(aln_exons)
[exon "fred" at [4:9]/10]
```

even if the name is different.

```
>>> exon = aln.get_seq('y').copy_annotations(s)
>>> aln_exons = list(aln.get_annotations_from_seq('y', 'exon'))
>>> print(aln_exons)
[exon "fred" at [3:4, 6:10]/10]
>>> print(aln[aln_exons])
>x
AAAAA
>y
TCCCC
```

If the feature lies outside the sequence being copied to, you get a lost span

```
>>> aln = LoadSeqs(data=[['x', '-AAAA'], ['y', 'TTTTT']], array_align=False)
>>> seq = DNA.make_seq('CCCCCCCCCCCCCCCCCCCC', 'x')
>>> exon = seq.add_feature('exon', 'A', [(5,8)])
>>> aln.get_seq('x').copy_annotations(seq)
>>> copied = list(aln.get_annotations_from_seq('x', 'exon'))
>>> copied
[exon "A" at [5:5, -4-]/5]
>>> copied[0].get_slice()
2 x 4 text alignment: x[----], y[----]
```

You can copy to a sequence with a different name, in a different alignment if the feature lies within the length

```
>>> aln = LoadSeqs(data=[['x', '-AAAAAAAAA'], ['y', 'TTTT--TTTT']], array_align=False)
>>> seq = DNA.make_seq('CCCCCCCCCCCCCCCCCCCC', 'x')
>>> match_exon = seq.add_feature('exon', 'A', [(5,8)])
>>> aln.get_seq('y').copy_annotations(seq)
>>> copied = list(aln.get_annotations_from_seq('y', 'exon'))
>>> copied
[exon "A" at [7:10]/10]
```

If the sequence is shorter, again you get a lost span.

```
>>> aln = LoadSeqs(data=[['x', '-AAAAAAAAA'], ['y', 'TTTT--TTTT']], array_align=False)
>>> diff_len_seq = DNA.make_seq('CCCCCCCCCCCCCCCCCCCCCCCCCCCC', 'x')
>>> nonmatch = diff_len_seq.add_feature('repeat', 'A', [(12,14)])
>>> aln.get_seq('y').copy_annotations(diff_len_seq)
>>> copied = list(aln.get_annotations_from_seq('y', 'repeat'))
>>> copied
[repeat "A" at [10:10, -6-]/10]
```

We consider cases where there are terminal gaps.

```
>>> aln = LoadSeqs(data=[['x', '-AAAAAAAAA'], ['y', '------TTTT']], array_align=False)
>>> exon = aln.get_seq('x').add_feature('exon', 'fred', [(3,8)])
>>> aln_exons = list(aln.get_annotations_from_seq('x', 'exon'))
>>> print(aln_exons)
[exon "fred" at [4:9]/10]
>>> print(aln_exons[0].get_slice())
>x
AAAAA
```

```
>y
--TTT

>>> aln = LoadSeqs(data=[['x', '-AAAAAAAAA'], ['y', 'TTTT--T---']], array_align=False)
>>> exon = aln.get_seq('x').add_feature('exon', 'fred', [(3,8)])
>>> aln_exons = list(aln.get_annotations_from_seq('x', 'exon'))
>>> print(aln_exons[0].get_slice())
>x
AAAAA
>y
--T--
```

In this case, only those residues included within the feature are covered - note the omission of the T in `y` opposite the gap in `x`.

```
>>> aln = LoadSeqs(data=[['x', 'C-CCCAAAAA'], ['y', '-T----TTTT']],
...                        moltype=DNA, array_align=False)
>>> print(aln)
>x
C-CCCAAAAA
>y
-T----TTTT

>>> exon = aln.get_seq('x').add_feature('exon', 'ex1', [(0,4)])
>>> print(exon)
exon "ex1" at [0:4]/9
>>> print(exon.get_slice())
CCCC
>>> aln_exons = list(aln.get_annotations_from_seq('x', 'exon'))
>>> print(aln_exons)
[exon "ex1" at [0:1, 2:5]/10]
>>> print(aln_exons[0].get_slice())
>x
CCCC
>y
----
```

`Feature.as_one_span()`, is applied to the exon that straddles the gap in `x`. The result is we preserve that feature.

```
>>> print(aln_exons[0].as_one_span().get_slice())
>x
C-CCC
>y
-T---
```

These properties also are consistently replicated with reverse complemented sequences.

```
>>> aln_rc = aln.rc()
>>> rc_exons = list(aln_rc.get_annotations_from_any_seq('exon'))
>>> print(aln_rc[rc_exons]) # not using as_one_span, so gap removed from x
>x
CCCC
>y
----

>>> print(aln_rc[rc_exons[0].as_one_span()])
>x
C-CCC
```

```
>y
-T---
```

Features can provide their coordinates, useful for custom analyses.

```
>>> all_exons = aln.get_region_covering_all(aln_exons)
>>> coords = all_exons.get_coordinates()
>>> assert coords == [(0,1),(2,5)]
```

Annotated regions can be masked (observed sequence characters replaced by another), either through the sequence on which they reside or by projection from the alignment. Note that `mask_char` must be a valid character for the sequence `MolType`. Either the features (multiple can be named), or their shadow, can be masked.

We create an alignment with a sequence that has two different annotation types.

```
>>> aln = LoadSeqs(data=[['x', 'C-CCCAAAAAGGGAA'], ['y', '-T----TTTTG-GTT']],
...                array_align=False)
>>> print(aln)
>x
C-CCCAAAAAGGGAA
>y
-T----TTTTG-GTT

>>> exon = aln.get_seq('x').add_feature('exon', 'norwegian', [(0,4)])
>>> print(exon.get_slice())
CCCC
>>> repeat = aln.get_seq('x').add_feature('repeat', 'blue', [(9,12)])
>>> print(repeat.get_slice())
GGG
>>> repeat = aln.get_seq('y').add_feature('repeat', 'frog', [(5,7)])
>>> print(repeat.get_slice())
GG
```

Each sequence should correctly mask either the single feature, it's shadow, or the multiple features, or shadow.

```
>>> print(aln.get_seq('x').with_masked_annotations('exon', mask_char='?'))
????AAAAAGGGAA
>>> print(aln.get_seq('x').with_masked_annotations('exon', mask_char='?',
...                                                shadow=True))
CCCC??????????
>>> print(aln.get_seq('x').with_masked_annotations(['exon', 'repeat'],
...                                                mask_char='?'))
????AAAAA???AA
>>> print(aln.get_seq('x').with_masked_annotations(['exon', 'repeat'],
...                                                mask_char='?', shadow=True))
CCCC?????GGG??
>>> print(aln.get_seq('y').with_masked_annotations('exon', mask_char='?'))
TTTTTGGTT
>>> print(aln.get_seq('y').with_masked_annotations('repeat', mask_char='?'))
TTTTT??TT
>>> print(aln.get_seq('y').with_masked_annotations('repeat', mask_char='?',
...                                                shadow=True))
?????GG??
```

The same methods can be applied to annotated Alignment's.

```
>>> print(aln.with_masked_annotations('exon', mask_char='?'))
>x
```

```
?-???AAAAAGGGAA
>y
-T----TTTTG-GTT

>>> print(aln.with_masked_annotations('exon', mask_char='?', shadow=True))
>x
C-CCC??????????
>y
-?----?????-???

>>> print(aln.with_masked_annotations('repeat', mask_char='?'))
>x
C-CCCAAAAA???AA
>y
-T----TTTT?-?TT

>>> print(aln.with_masked_annotations('repeat', mask_char='?', shadow=True))
>x
?-????????GGG??
>y
-?----????G-G??

>>> print(aln.with_masked_annotations(['repeat', 'exon'], mask_char='?'))
>x
?-???AAAAA???AA
>y
-T----TTTT?-?TT

>>> print(aln.with_masked_annotations(['repeat', 'exon'],shadow=True))
>x
C-CCC?????GGG??
>y
-?----????G-G??
```

It shouldn't matter whether annotated coordinates are entered separately, or as a series.

```
>>> data = [['human', 'CGAAACGTTT'], ['mouse', 'CTAAACGTCG']]
>>> as_series = LoadSeqs(data=data, array_align=False)
>>> as_items = LoadSeqs(data=data, array_align=False)
```

We add annotations to the sequences as a series.

```
>>> as_series.get_seq('human').add_feature('cpgsite', 'cpg', [(0,2), (5,7)])
cpgsite "cpg" at [0:2, 5:7]/10
>>> as_series.get_seq('mouse').add_feature('cpgsite', 'cpg', [(5,7), (8,10)])
cpgsite "cpg" at [5:7, 8:10]/10
```

We add the annotations to the sequences one segment at a time.

```
>>> as_items.get_seq('human').add_feature('cpgsite', 'cpg', [(0,2)])
cpgsite "cpg" at [0:2]/10
>>> as_items.get_seq('human').add_feature('cpgsite', 'cpg', [(5,7)])
cpgsite "cpg" at [5:7]/10
>>> as_items.get_seq('mouse').add_feature('cpgsite', 'cpg', [(5,7)])
cpgsite "cpg" at [5:7]/10
>>> as_items.get_seq('mouse').add_feature('cpgsite', 'cpg', [(8,10)])
cpgsite "cpg" at [8:10]/10
```

These different constructions should generate the same output.

```
>>> serial = as_series.with_masked_annotations(['cpgsite'])
>>> print(serial)
>human
??AAA??TTT
>mouse
CTAAA??T??

>>> itemwise = as_items.with_masked_annotations(['cpgsite'])
>>> print(itemwise)
>human
??AAA??TTT
>mouse
CTAAA??T??
```

Annotations should be correctly masked, whether the sequence has been reverse complemented or not. We use the plus/minus strand CDS containing sequences created above.

```
>>> print(plus.with_masked_annotations("CDS"))
AA????AAAA?????AAAAAAAAAA??????????AAA
>>> print(minus.with_masked_annotations("CDS"))
TTT??????????TTTTTTTTT?????TTTT????TT
```

Getting the reverse complement

Section author: Gavin Huttley

This is a property of DNA, and hence alignments need to be created with the appropriate `MolType`. In the following example, the alignment is truncated to just 50 bases for the sake of simplifying the presentation.

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs("data/long_testseqs.fasta", moltype=DNA)[:50]
```

The original alignment looks like this.

```
>>> print(aln)
>Human
TGTGGCACAAATACTCATGCCAGCTCATTACAGCATGAGAACAGCAGTTT
>HowlerMon
TGTGGCACAAATACTCATGCCAGCTCATTACAGCATGAGAACAGCAGTTT
>Mouse
TGTGGCACAGATGCTCATGCCAGCTCATTACAGCCTGAGACCAGCAGTTT
>NineBande
TGTGGCACAAATACTCATGCCAACTTATTACAGCATGAGAACAGCAGTTT
>DogFaced
TGTGGCACAAATACTCATGCCAACTCATTACAGCATGAGAACAGCAGTTT
```

We do reverse complement very simply.

```
>>> naln = aln.rc()
```

The reverse complemented alignment looks like this.

```
>>> print(naln)
>Human
AAACTGCTGTTCTCATGCTGTAATGAGCTGGCATGAGTATTTGTGCCACA
```

```
>HowlerMon
AAACTGCTGTTCTCATGCTGTAATGAGCTGGCATGAGTATTTGTGCCACA
>Mouse
AAACTGCTGGTCTCAGGCTGTAATGAGCTGGCATGAGCATCTGTGCCACA
>NineBande
AAACTGCTGTTCTCATGCTGTAATAAGTTGGCATGAGTATTTGTGCCACA
>DogFaced
AAACTGCTGTTCTCATGCTGTAATGAGTTGGCATGAGTATTTGTGCCACA
```

Map protein alignment gaps to DNA alignment gaps

Section author: Gavin Huttley

Although PyCogent3 provides a means for directly aligning codon sequences, you may want to use a different approach based on the translate-align-introduce gaps into the original paradigm. After you've translated your codon sequences, and aligned the resulting amino acid sequences, you want to introduce the gaps from the aligned protein sequences back into the original codon sequences. Here's how.

```
>>> from cogent3 import LoadSeqs, DNA, PROTEIN
```

First I'm going to construct an artificial example, using the seqs dict as a means to get the data into the Alignment object. The basic idea, however, is that you should already have a set of DNA sequences that are in frame (i.e. position 0 is the 1st codon position), you've translated those sequences and aligned these translated sequences. The result is an alignment of aa sequences and a set of unaligned DNA sequences from which the aa seqs were derived. If your sequences are not in frame you can adjust it by either slicing, or adding N's to the beginning of the raw string.

```
>>> seqs = {
... 'hum':
↪'AAGCAGATCCAGGAAAGCAGCGAGAATGGCAGCCTGGCCGCGCGCCAGGAGAGGCAGGCCCAGGTCAACCTCACT',
... 'mus':
↪'AAGCAGATCCAGGAGAGCGGCGAGAGCGGCAGCCTGGCCGCGCGGCAGGAGAGGCAGGCCCAAGTCAACCTCACG',
... 'rat': 'CTGAACAAGCAGCCACTTTCAAACAAGAAA'}
>>> unaligned_DNA = LoadSeqs(data=seqs, moltype=DNA, aligned=False)
>>> print(unaligned_DNA.to_fasta())
>hum
AAGCAGATCCAGGAAAGCAGCGAGAATGGCAGCCTGGCCGCGCGCCAGGAGAGGCAGGCCCAGGTCAACCTCACT
>mus
AAGCAGATCCAGGAGAGCGGCGAGAGCGGCAGCCTGGCCGCGCGGCAGGAGAGGCAGGCCCAAGTCAACCTCACG
>rat
CTGAACAAGCAGCCACTTTCAAACAAGAAA
```

In order to ensure the alignment algorithm preserves the coding frame, we align the translation of the sequences. We need to translate them first, but note that because the seqs are unaligned they we have to set `aligned=False`, or we'll get an error.

```
>>> unaligned_aa = unaligned_DNA.get_translation()
>>> print(unaligned_aa.to_fasta())
>hum
KQIQESSENGSLAARQERQAQVNLT
>mus
KQIQESGESGSLAARQERQAQVNLT
>rat
LNKQPLSNKK
```

The translated seqs can then be written to file, using the method `write`. That file then serves as input for an alignment program. The resulting alignment file can be read back in. (We won't write to file in this example.) For this example

we will specify the aligned sequences in the dict, rather than from file.

```
>>> aligned_aa_seqs = {'hum': 'KQIQESSENGSLAARQERQAQVNLT',
... 'mus': 'KQIQESGESGSLAARQERQAQVNLT',
... 'rat': 'LNKQ------PLS---------NKK'}
>>> aligned_aa = LoadSeqs(data=aligned_aa_seqs, moltype=PROTEIN)
>>> aligned_DNA = aligned_aa.replace_seqs(unaligned_DNA)
```

Just to be sure, we'll check that the DNA sequence has gaps in the right place.

```
>>> print(aligned_DNA)
>hum
AAGCAGATCCAGGAAAGCAGCGAGAATGGCAGCCTGGCCGCGCGCCAGGAGAGGCAGGCCCAGGTCAACCTCACT
>rat
CTGAACAAGCAG----------------CCACTTTCA-------------------------AACAAGAAA
>mus
AAGCAGATCCAGGAGAGCGGCGAGAGCGGCAGCCTGGCCGCGCGGCAGGAGAGGCAGGCCCAAGTCAACCTCACG
```

Creating and manipulating alignment profiles

Section author: Sandra Smit

This is an example of how to create a profile from an alignment and how to do particular tricks with it. First, import the necessary stuff.

```
>>> from cogent3.core.profile import Profile
>>> from cogent3 import LoadSeqs, RNA
```

Then load an example alignment of 20 phe-tRNA sequences which we will use to create the profile

```
>>> aln = LoadSeqs("data/trna_profile.fasta", moltype=RNA)
```

Examine the number of sequences in the alignment and the alignment length

```
>>> print(len(aln.seqs))
20
>>> print(len(aln))
77
```

Create a profile containing the counts of each base at each alignment position

```
>>> pf = aln.get_pos_freqs()
>>> print(pf.pretty_print(include_header=True, column_limit=6, col_sep='    '))
 U     C     A     G     -    H
 0     0     0    20     0    0
 0    12     0     8     0    0
 1    18     0     1     0    0
 7     9     0     4     0    0...
```

Normalize the positions to get the relative frequencies at each position

```
>>> pf.normalize_positions()
>>> print(pf.pretty_print(include_header=True, column_limit=6, col_sep='    '))
     U         C         A         G         -         B
0.0000    0.0000    0.0000    1.0000    0.0000    0.0000
0.0000    0.6000    0.0000    0.4000    0.0000    0.0000
```

```
0.0500   0.9000   0.0000   0.0500   0.0000   0.0000
0.3500   0.4500   0.0000   0.2000   0.0000   0.0000...
```

Make sure the data in the profile is valid. The method is_valid checks whether all rows add up to one and whether the profile has a valid Alphabet and CharacterOrder.

```
>>> print(pf.is_valid())
True
```

A profile can be used to calculate consensus sequences from the alignment. To illustrate the different options for consensus calculation, let's examine the frequency data at the fifth position of the alignment (index=4)

```
>>> print('\n'.join(['%s: %.3f'%(c,f) for (c,f) in\
...     zip(pf.char_order, pf.data_at(4)) if f!=0]))
U: 0.050
C: 0.400
A: 0.250
G: 0.300
```

The easiest consensus calculation will simply take the most frequent character at each position.

```
>>> print(pf.to_consensus(fully_degenerate=False))
GCCCCGGUAGCUCAGU--GGUAGAGCAGGGGACUGAAAAUCCCCGUGUCGGCGGUUCGAUUCCGUCCCGGGGCACCA
```

You can also specify to use the degenerate character needed to cover all symbols occurring at a certain alignment position (fully_degenerate=True). At index 4 in the alignment U, C, A, and G occur, thus the fully degenerate symbol needed is 'N'. Alternatively, using the cutoff value, you can ask for the degenerate symbol needed to cover a certain frequency. At a cutoff of 0.8, we need both C, G, and A at index 4 to cover this value, which results in the degenerate character 'V'. For the lower cutoff of 0.6, C and G suffice, and thus the character in the consensus sequence is 'S'.

```
>>> pf.alphabet=RNA
>>> print(pf.to_consensus(fully_degenerate=True))
GSBBNNDUAGCUCAGH??GGKAGAGCRBNVGRYUGAARAYCBNVNKGUCVBBDGWUCRAWHCHSNBHNNNVSC?CHM
>>> print(pf.to_consensus(cutoff=0.8))
GSCYVBRUAGCUCAGU??GGUAGAGCASVSGAYUGAAAAUCYBSRUGUCSSYGGUUCGAUUCCGBSYSBRGSCACCA
>>> print(pf.to_consensus(cutoff=0.6))
GCCYSGRUAGCUCAGU??GGUAGAGCAGRGGACUGAAAAUCCYCGUGUCGGYGGUUCGAUUCCGYCYCKRGGCACCA
```

A profile could also function as the description of a certain motif. As an example, let's create a profile description for the T-pseudouridine-C-loop which starts at index 54 and ends at index 59 (based on the reference structure matching the alignment).

```
>>> loop_profile = Profile(pf.Data[54:60,:], alphabet=RNA,
...                        char_order=pf.char_order)
>>> print(loop_profile.pretty_print(include_header=True, column_limit=6,
...                                 col_sep='   '))
     U        C        A        G        -        B
0.9500   0.0000   0.0500   0.0000   0.0000   0.0000
1.0000   0.0000   0.0000   0.0000   0.0000   0.0000
0.0000   1.0000   0.0000   0.0000   0.0000   0.0000
0.0000   0.0000   0.0500   0.9500   0.0000   0.0000
0.0000   0.0000   1.0000   0.0000   0.0000   0.0000
0.8500   0.0000   0.1500   0.0000   0.0000   0.0000
```

We can calculate how well this profile matches in a certain sequence (or profile) by using the score method. As an example we see where the loop profile best fits into the yeast phe-tRNA sequence. As expected, we find the best hit at index 54 (with a score of 5.75).

```
>>> yeast = RNA.make_seq(
...     'GCGGAUUUAGCUCAGUU-GGGAGAGCGCCAGACUGAAGAUCUGGAGGUCCUGUGUUCGAUCCACAGAAUUCGCACCA
↪')
>>> scores = loop_profile.score(yeast)
>>> print(scores)
[ 2.8   0.9   0.85  0.15  2.05  2.    3.75  0.95  1.2   1.    2.9   2.75
  0.    0.05  1.    2.9   2.05  1.95  0.2   1.95  0.05  1.    0.    2.
  0.15  2.    1.2   1.95  0.9   0.05  1.15  2.15  2.05  1.15  2.8   0.1
  0.9   0.    2.05  2.05  2.95  1.    1.8   0.95  0.05  0.85  2.    2.8
  0.95  1.85  2.75  1.    0.95  1.15  5.75  1.    0.    0.15  3.05  2.15
  1.    1.2   2.15  1.9   0.95  0.    0.05  1.05  4.05  1.95  1.05  0.15]
>>> print(max(scores))
5.75
>>> print(scores.argmax())
54
```

Compute the effect of a nucleotide substitution on residue polarity in two different genetic codes using GeneticCode and AAIndex

Section author: Greg Caporaso

This document illustrates how to work with a genetic code object, and compare two different genetic codes. Here we compare the change in residue polarity, as judged by the Woese Polarity Requirement index (Woese 1973), resulting from a nucleotide substitution if the sequence is translated with the standard nuclear genetic code, or the vertebrate mitochondrial genetic code.

First, we load the genetic code objects and look at how they differ from one another.

```
>>> from cogent3.core.genetic_code import GeneticCode
>>> code = 'FFLLSSSSYY**CC*WLLLLPPPPHHQQRRRRIIIMTTTTNNKKSSRRVVVVAAAADDEEGGGG'
>>> standard_nuclear_genetic_code = GeneticCode(code)
>>> code = 'FFLLSSSSYY**CCWWLLLLPPPPHHQQRRRRIIMMTTTTNNKKSS**VVVVAAAADDEEGGGG'
>>> vertebrate_mitochondrial_genetic_code = GeneticCode(code)
>>> standard_nuclear_genetic_code == vertebrate_mitochondrial_genetic_code
False
```

We'll make some synonyms for the objects for simplicity, and then look at the differences between the two codes:

```
>>> ngc = standard_nuclear_genetic_code
>>> mgc = vertebrate_mitochondrial_genetic_code

>>> differences = list(ngc.changes(mgc).items())
>>> differences.sort()
>>> differences
[('AGA', 'R*'), ('AGG', 'R*'), ('ATA', 'IM'), ('TGA', '*W')]
```

Next, let's load the Woese Polar Requirement `AAIndex` data, and find the effect of an ATA to ATG substitution with each of the two `GeneticCode` objects.

```
>>> from cogent3.parse.aaindex import getWoeseDistanceMatrix
>>> woese_distance_matrix = getWoeseDistanceMatrix()
>>> woese_distance_matrix[ngc['ATA']][ngc['ATG']]
0.39999999999999947
>>> woese_distance_matrix[mgc['ATA']][mgc['ATG']]
0.0
```

This illustrates that there is a difference in residue polarity associated with substitution only in the standard nuclear code (where ATA to ATG translates to an isoleucine to methionine substitution). In the vertebrate mitochondrial code, ATA to ATG is a synonymous substitution. Calculations of this type were central to[1] which presents the study that these modules were initially developed for.

`GeneticCode` objects can also be used to translate DNA sequences (where asterisks in the results refer to stop-translation characters):

```
>>> dna = "AAACGCTGTGTGTGAGATGAAAAA"
>>> ngc.translate(dna)
'KRCV*DEK'
>>> mgc.translate(dna)
'KRCVWDEK'
```

The standard nuclear genetic code can also be loaded as `DEFAULT`:

```
>>> from cogent3.core.genetic_code import DEFAULT
>>> DEFAULT == standard_nuclear_genetic_code
True
```

Citations

Data Manipulation using `Table`

Section author: Gavin Huttley

The toolkit has a `Table` object that can be used for manipulating tabular data. It's properties can be considered like an ordered 2 dimensional dictionary or tuple with flexible output format capabilities of use for exporting data for import into external applications. Importantly, via the restructured text format one can generate html or latex formatted tables. The `table` module is located within `cogent3.util`. The `LoadTable` convenience function is provided as a top-level `cogent3` import.

Table creation

Tables can be created directly using the Table object itself, or a convenience function that handles loading from files. We import both here:

```
>>> from cogent3 import LoadTable
>>> from cogent3.util.table import Table
```

First, if you try and create a `Table` without any data, it raises a `RuntimeError`.

```
>>> t = Table()
Traceback (most recent call last):
RuntimeError: header and rows must be provided to Table
>>> t = Table(header=[], rows=[])
Traceback (most recent call last):
RuntimeError: header and rows must be provided to Table
```

Let's create a very simple, rather nonsensical, table first. To create a table requires a header series, and a 2D series (either of type `tuple`, `list`, `dict`) or a pandas DataFrame..

[1] Caporaso, Yarus, and Knight. *Error minimization and coding triplet/binding site associations are independent features of the canonical genetic code.* J Mol Evol, 61(5):597-607, 2005.

```
>>> column_headings = ['chrom', 'stableid', 'length']
```

The string "chrom" will become the first column heading, "stableid" the second column heading, etc. The data are,

```
>>> rows = [['X', 'ENSG00000005893', 1353],
...         ['A', 'ENSG00000019485', 1827],
...         ['A', 'ENSG00000019102', 999],
...         ['X', 'ENSG00000012174', 1599],
...         ['X', 'ENSG00000010671', 1977],
...         ['A', 'ENSG00000019186', 1554],
...         ['A', 'ENSG00000019144', 4185],
...         ['X', 'ENSG00000008056', 2307],
...         ['A', 'ENSG00000018408', 1383],
...         ['A', 'ENSG00000019169', 1698]]
...
>>>
```

We create the simplest of tables.

```
>>> t = Table(header=column_headings, rows=rows)
>>> print(t)
=================================
chrom           stableid    length
---------------------------------
    X    ENSG00000005893      1353
    A    ENSG00000019485      1827
    A    ENSG00000019102       999
    X    ENSG00000012174      1599
    X    ENSG00000010671      1977
    A    ENSG00000019186      1554
    A    ENSG00000019144      4185
    X    ENSG00000008056      2307
    A    ENSG00000018408      1383
    A    ENSG00000019169      1698
---------------------------------
```

The format above is referred to as 'simple' format in the documentation. Notice that the numbers in this table have 4 decimal places, despite the fact the original data were largely strings and had `max` of 3 decimal places precision. `Table` converts string representations of numbers to their appropriate form when you do `str(table)` or print the table.

We have several things we might want to specify when creating a table: the precision and or format of floating point numbers (integer argument - `digits`), the spacing between columns (integer argument or actual string of whitespace - `space`), title (argument - `title`), and legend (argument - `legend`). Lets modify some of these and provide a title and legend.

```
>>> t = Table(column_headings, rows, title='Alignment lengths',
...           legend='Some analysis',
...           digits=2, space='        ')
>>> print(t)
Alignment lengths
========================================
chrom               stableid        length
----------------------------------------
    X        ENSG00000005893          1353
    A        ENSG00000019485          1827
    A        ENSG00000019102           999
    X        ENSG00000012174          1599
```

```
    X         ENSG00000010671          1977
    A         ENSG00000019186          1554
    A         ENSG00000019144          4185
    X         ENSG00000008056          2307
    A         ENSG00000018408          1383
    A         ENSG00000019169          1698
----------------------------------------
Some analysis
```

Note: The `repr()` of a table gives a quick summary.

```
>>> t
Table(numrows=10, numcols=3, header=['chrom', 'stableid', 'length'], rows=[['X',
 ↪'ENSG00000005893', 1353],..])
```

The Table class cannot handle arbitrary python objects, unless they are passed in as strings. Note in this case we now directly pass in the column headings list and the handling of missing data can be explicitly specified..

```
>>> t2 = Table(['abcd', 'data'], [[str(list(range(1,6))), '0'],
...                               ['x', 5.0], ['y', None]],
...            missing_data='*')
>>> print(t2)
=========================
           abcd      data
-------------------------
[1, 2, 3, 4, 5]          0
              x     5.0000
              y          *
-------------------------
```

Table column headings can be assessed from the `table.header` property

```
>>> assert t2.header == ['abcd', 'data']
```

and this is immutable (cannot be changed).

```
>>> t2.header[1] = 'Data'
Traceback (most recent call last):
RuntimeError: Table header is immutable, use with_new_header
```

If you want to change the header, use the `with_new_header` method. This can be done one column at a time, or as a batch. The returned Table is identical aside from the modified column labels.

```
>>> mod_header = t2.with_new_header('abcd', 'ABCD')
>>> assert mod_header.header == ['ABCD', 'data']
>>> mod_header = t2.with_new_header(['abcd', 'data'], ['ABCD', 'DATA'])
>>> print(mod_header)
=========================
           ABCD      DATA
-------------------------
[1, 2, 3, 4, 5]          0
              x     5.0000
              y          *
-------------------------
```

Tables may also be created from 2-dimensional dictionaries. In this case, special capabilities are provided to enforce printing rows in a particular order.

```
>>> d2D={'edge.parent': {'NineBande': 'root', 'edge.1': 'root',
... 'DogFaced': 'root', 'Human': 'edge.0', 'edge.0': 'edge.1',
... 'Mouse': 'edge.1', 'HowlerMon': 'edge.0'}, 'x': {'NineBande': 1.0,
... 'edge.1': 1.0, 'DogFaced': 1.0, 'Human': 1.0, 'edge.0': 1.0,
... 'Mouse': 1.0, 'HowlerMon': 1.0}, 'length': {'NineBande': 4.0,
... 'edge.1': 4.0, 'DogFaced': 4.0, 'Human': 4.0, 'edge.0': 4.0,
... 'Mouse': 4.0, 'HowlerMon': 4.0}, 'y': {'NineBande': 3.0, 'edge.1': 3.0,
... 'DogFaced': 3.0, 'Human': 3.0, 'edge.0': 3.0, 'Mouse': 3.0,
... 'HowlerMon': 3.0}, 'z': {'NineBande': 6.0, 'edge.1': 6.0,
... 'DogFaced': 6.0, 'Human': 6.0, 'edge.0': 6.0, 'Mouse': 6.0,
... 'HowlerMon': 6.0},
... 'edge.name': ['Human', 'HowlerMon', 'Mouse', 'NineBande', 'DogFaced',
... 'edge.0', 'edge.1']}
>>> row_order = d2D['edge.name']
>>> d2D['edge.name'] = dict(zip(row_order, row_order))
>>> t3 = Table(['edge.name', 'edge.parent', 'length', 'x', 'y', 'z'], d2D,
...            row_order=row_order, missing_data='*', space=8,
...            max_width=50, row_ids=True, title='My title',
...            legend='legend: this is a nonsense example.')
>>> print(t3)
My title
=========================================
edge.name        edge.parent        length
-----------------------------------------
    Human             edge.0        4.0000
HowlerMon             edge.0        4.0000
    Mouse             edge.1        4.0000
NineBande               root        4.0000
 DogFaced               root        4.0000
   edge.0             edge.1        4.0000
   edge.1               root        4.0000
-----------------------------------------

continued: My title
===================================
edge.name             x             y
-----------------------------------
    Human        1.0000        3.0000
HowlerMon        1.0000        3.0000
    Mouse        1.0000        3.0000
NineBande        1.0000        3.0000
 DogFaced        1.0000        3.0000
   edge.0        1.0000        3.0000
   edge.1        1.0000        3.0000
-----------------------------------

continued: My title
=======================
edge.name             z
-----------------------
    Human        6.0000
HowlerMon        6.0000
    Mouse        6.0000
NineBande        6.0000
 DogFaced        6.0000
   edge.0        6.0000
```

```
   edge.1        6.0000
----------------------

legend: this is a nonsense example.
```

In the above we specify a maximum width of the table, and also specify row identifiers (using `row_ids`, the integer corresponding to the column at which data begin, preceding columns are taken as the identifiers). This has the effect of forcing the table to wrap when the simple text format is used, but wrapping does not occur for any other format. The `row_ids` are keys for slicing the table by row, and as identifiers are presented in each wrapped sub-table.

Wrapping generate neat looking tables whether or not you index the table rows. We demonstrate here

```
>>> from cogent3 import LoadTable
>>> h = ['A/C', 'A/G', 'A/T', 'C/A']
>>> rows = [[0.0425, 0.1424, 0.0226, 0.0391]]
>>> wrap_table = LoadTable(header=h, rows=rows, max_width=30)
>>> print(wrap_table)
==============================
       A/C       A/G       A/T
------------------------------
    0.0425    0.1424    0.0226
------------------------------

continued:
==========
       C/A
----------
    0.0391
----------

>>> wrap_table = LoadTable(header=h, rows=rows, max_width=30,
...  row_ids=True)
>>> print(wrap_table)
==========================
   A/C       A/G       A/T
--------------------------
0.0425    0.1424    0.0226
--------------------------

continued:
================
   A/C       C/A
----------------
0.0425    0.0391
----------------
```

We can also customise the formatting of individual columns.

```
>>> rows = (('NP_003077_hs_mm_rn_dna', 'Con', 2.5386013224378985),
... ('NP_004893_hs_mm_rn_dna', 'Con', 0.12135142635634111e+06),
... ('NP_005079_hs_mm_rn_dna', 'Con', 0.95165949788861326e+07),
... ('NP_005500_hs_mm_rn_dna', 'Con', 0.73827030202664901e-07),
... ('NP_055852_hs_mm_rn_dna', 'Con', 1.0933217708952725e+07))
```

We first create a table and show the default formatting behaviour for `Table`.

```
>>> t46 = Table(['Gene', 'Type', 'LR'], rows)
>>> print(t46)
```

```
=============================================
                  Gene    Type            LR
---------------------------------------------
NP_003077_hs_mm_rn_dna     Con        2.5386
NP_004893_hs_mm_rn_dna     Con     121351.4264
NP_005079_hs_mm_rn_dna     Con    9516594.9789
NP_005500_hs_mm_rn_dna     Con          0.0000
NP_055852_hs_mm_rn_dna     Con   10933217.7090
---------------------------------------------
```

We then format the `LR` column to use a scientific number format.

```
>>> t46 = Table(['Gene', 'Type', 'LR'], rows)
>>> t46.format_column('LR', "%.4e")
>>> print(t46)
=========================================
                  Gene    Type         LR
-----------------------------------------
NP_003077_hs_mm_rn_dna     Con  2.5386e+00
NP_004893_hs_mm_rn_dna     Con  1.2135e+05
NP_005079_hs_mm_rn_dna     Con  9.5166e+06
NP_005500_hs_mm_rn_dna     Con  7.3827e-08
NP_055852_hs_mm_rn_dna     Con  1.0933e+07
-----------------------------------------
```

It is safe to directly modify certain attributes, such as the title, legend and white space separating columns, which we do for the `t46`.

```
>>> t46.title = "A new title"
>>> t46.legend = "A new legend"
>>> t46.space = '  '
>>> print(t46)
A new title
=======================================
                  Gene  Type         LR
---------------------------------------
NP_003077_hs_mm_rn_dna   Con  2.5386e+00
NP_004893_hs_mm_rn_dna   Con  1.2135e+05
NP_005079_hs_mm_rn_dna   Con  9.5166e+06
NP_005500_hs_mm_rn_dna   Con  7.3827e-08
NP_055852_hs_mm_rn_dna   Con  1.0933e+07
---------------------------------------
A new legend
```

We can provide settings for multiple columns.

```
>>> t3 = Table(['edge.name', 'edge.parent', 'length', 'x', 'y', 'z'], d2D,
...            row_order=row_order)
>>> t3.format_column('x', "%.1e")
>>> t3.format_column('y', "%.2f")
>>> print(t3)
============================================================
edge.name    edge.parent    length          x       y         z
------------------------------------------------------------
    Human         edge.0    4.0000    1.0e+00    3.00    6.0000
HowlerMon         edge.0    4.0000    1.0e+00    3.00    6.0000
    Mouse         edge.1    4.0000    1.0e+00    3.00    6.0000
NineBande           root    4.0000    1.0e+00    3.00    6.0000
```

```
 DogFaced           root     4.0000    1.0e+00     3.00     6.0000
   edge.0         edge.1     4.0000    1.0e+00     3.00     6.0000
   edge.1           root     4.0000    1.0e+00     3.00     6.0000
-------------------------------------------------------------------
```

In some cases, the contents of a column can be of different types. In this instance, rather than passing a column template we pass a reference to a function that will handle this complexity. To illustrate this we will define a function that formats floating point numbers, but returns everything else as is.

```
>>> def formatcol(value):
...     if isinstance(value, float):
...         val = "%.2f" % value
...     else:
...         val = str(value)
...     return val
```

We apply this to a table with mixed string, integer and floating point data.

```
>>> t6 = Table(['ColHead'], [['a'], [1], [0.3], ['cc']],
...            column_templates=dict(ColHead=formatcol))
>>> print(t6)
=======
ColHead
-------
      a
      1
   0.30
     cc
-------
```

Creating a Table from a pandas DataFrame

Assign the `DataFrame` instance to the `data_frame` argument.

```
>>> from pandas import DataFrame
>>> df = DataFrame(data=[[0, 1], [3,7]], columns=['a', 'b'])
>>> print(df)
   a  b
0  0  1
1  3  7
>>> df_as_table = LoadTable(data_frame=df)
>>> print(df_as_table)
======
a    b
------
0    1
3    7
------
```

Representation of tables

The representation formatting provides a quick overview of a table's dimensions and it's contents. We show this for a table with 3 columns and multiple rows

```
>>> t46
Table(numrows=5, numcols=3, header=['Gene', 'Type', 'LR'], rows=[['NP_003077_hs_mm_rn_
↪dna', 'Con', 2.5386],..])
```

and larger

```
>>> t3
Table(numrows=7, numcols=6, header=['edge.name', 'edge.parent', 'length',..], rows=[[
↪'Human', 'edge.0', 4.0000,..],..])
```

Note: within a script use `print(repr(t3))` to get the same representation.

Table output

Table can output in multiple formats, including restructured text or 'rest' and delimited. These can be obtained using the `tostring` method and `format` argument as follows. Using table `t` from above,

```
>>> print(t.tostring(format='rest'))
+-----------------------------------+
|         Alignment lengths         |
+-------+-----------------+---------+
| chrom |        stableid | length  |
+=======+=================+=========+
|     X | ENSG00000005893 |   1353  |
+-------+-----------------+---------+
|     A | ENSG00000019485 |   1827  |
+-------+-----------------+---------+
|     A | ENSG00000019102 |    999  |
+-------+-----------------+---------+
|     X | ENSG00000012174 |   1599  |
+-------+-----------------+---------+
|     X | ENSG00000010671 |   1977  |
+-------+-----------------+---------+
|     A | ENSG00000019186 |   1554  |
+-------+-----------------+---------+
|     A | ENSG00000019144 |   4185  |
+-------+-----------------+---------+
|     X | ENSG00000008056 |   2307  |
+-------+-----------------+---------+
|     A | ENSG00000018408 |   1383  |
+-------+-----------------+---------+
|     A | ENSG00000019169 |   1698  |
+-------+-----------------+---------+
| Some analysis                     |
+-----------------------------------+
```

or Markdown format

```
>>> print(t.tostring(format='md'))
| chrom |        stableid | length  |
|-------|-----------------|---------|
|     X | ENSG00000005893 |   1353  |
|     A | ENSG00000019485 |   1827  |
|     A | ENSG00000019102 |    999  |
|     X | ENSG00000012174 |   1599  |...
```

which can also take an optional *justify* argument. The latter must be a series with a value for each column. (It only affects the html display of a Markdown table.)

```
>>> print(t.tostring(format='md', justify='lcr'))
| chrom |        stableid | length |
|:------|:---------------:|-------:|
|     X | ENSG00000005893 |   1353 |
|     A | ENSG00000019485 |   1827 |
|     A | ENSG00000019102 |    999 |
|     X | ENSG00000012174 |   1599 |...
```

where the values *lcr* correspond to left, centre and right justification.

In the case of Markdown, the pipe character (`|`) is special and so cells containing it must be escaped.

```
>>> md_table = LoadTable(header=["a", "b"],
...                      rows=[["val1", "val2"],
...                            ["has | symbol", "val4"]])
>>> print(md_table.tostring(format='md'))
|              a |    b |
|----------------|------|
|           val1 | val2 |
| has \| symbol | val4 |
```

Arguments such as `space` have no effect in this case. The table may also be written to file in any of the available formats (latex, simple text, html, pickle) or using a custom separator (such as a comma or tab). This makes it convenient to get data into other applications (such as R or a spreadsheet program).

The display format can be specified for a `Table` using any valid argument to `tostring()`. For instance, we can make a `Table` instance that defaults to Markdown display.

```
>>> md_table = LoadTable(header=["a", "b"],
...                      rows=[["val1", "val2"],
...                            ["has | symbol", "val4"]],
...                      format="md")
>>> print(md_table)
|              a |    b |
|----------------|------|
|           val1 | val2 |
| has \| symbol | val4 |
```

This can be changed by modifying the *format* attribute, for example

```
>>> md_table.format = "rst"
>>> print(md_table)
+--------------+------+
|            a |    b |
+==============+======+
|         val1 | val2 |
+--------------+------+
| has | symbol | val4 |
+--------------+------+
```

Here is the latex format, note how the title and legend are joined into the latex table caption. We also provide optional arguments for the column alignment (fist column left aligned, second column right aligned and remaining columns centred) and a label for table referencing.

```
>>> print(t3.tostring(format='tex', justify="lrcccc", label="table:example"))
\begin{longtable}[htp!]{ l r c c c c }
\hline
\bf{edge.name} & \bf{edge.parent} & \bf{length} & \bf{x} & \bf{y} & \bf{z} \\
\hline
\hline
    Human &      edge.0 & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
HowlerMon &      edge.0 & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
    Mouse &      edge.1 & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
NineBande &        root & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
 DogFaced &        root & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
   edge.0 &      edge.1 & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
   edge.1 &        root & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
\hline
\label{table:example}
\end{longtable}
```

More complex latex table justifying is also possible. Specifying the width of individual columns requires passing in a series (list or tuple) of justification commands. In the following we introduce the command for specific columns widths.

```
>>> print(t3.tostring(format='tex', justify=["l","p{3cm}","c","c","c","c"]))
\begin{longtable}[htp!]{ l p{3cm} c c c c }
\hline
\bf{edge.name} & \bf{edge.parent} & \bf{length} & \bf{x} & \bf{y} & \bf{z} \\
\hline
\hline
    Human &      edge.0 & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
HowlerMon &      edge.0 & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
    Mouse &      edge.1 & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
NineBande &        root & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
 DogFaced &        root & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
   edge.0 &      edge.1 & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
   edge.1 &        root & 4.0000 & 1.0e+00 & 3.00 & 6.0000 \\
\hline
\end{longtable}
>>> print(t3.tostring(sep=','))
edge.name,edge.parent,length,      x,    y,      z
    Human,      edge.0,4.0000,1.0e+00,3.00,6.0000
HowlerMon,      edge.0,4.0000,1.0e+00,3.00,6.0000
    Mouse,      edge.1,4.0000,1.0e+00,3.00,6.0000
NineBande,        root,4.0000,1.0e+00,3.00,6.0000
 DogFaced,        root,4.0000,1.0e+00,3.00,6.0000
   edge.0,      edge.1,4.0000,1.0e+00,3.00,6.0000
   edge.1,        root,4.0000,1.0e+00,3.00,6.0000
```

You can specify any standard text character that will work with your desired target. Useful separators are tabs (`\t`), or pipes (`|`). If `Table` encounters the specified separator character within a cell, it wraps the cell in quotes – a standard approach to facilitate import by other applications. We will illustrate this with `t2`.

```
>>> print(t2.tostring(sep=', '))
           abcd,   data
"[1, 2, 3, 4, 5]",      0
              x, 5.0000
              y,      *
```

Note that I introduced an extra space after the column just to make the result more readable in this example.

Test the writing of phylip distance matrix format.

```
>>> rows = [['a', '', 0.088337278874079342, 0.18848582712597683,
...  0.44084000179091454], ['c', 0.088337278874079342, '',
...  0.088337278874079342, 0.44083999937417828], ['b', 0.18848582712597683,
...  0.088337278874079342, '', 0.44084000179090932], ['e',
...  0.44084000179091454, 0.44083999937417828, 0.44084000179090932, '']]
>>> header = ['seq1/2', 'a', 'c', 'b', 'e']
>>> dist = Table(header=header, rows=rows, row_ids=True)
>>> print(dist.tostring(format='phylip'))
   4
a           0.0000  0.0883  0.1885  0.4408
c           0.0883  0.0000  0.0883  0.4408
b           0.1885  0.0883  0.0000  0.4408
e           0.4408  0.4408  0.4408  0.0000
```

The `tostring` method also provides generic html generation via the restructured text format. The `to_rich_html` method can be used to generate the html table element by itself, with greater control over formatting. Specifically, users can provide custom callback functions to the `row_cell_func` and `header_cell_func` arguments to control in detail the formatting of table elements, or use the simpler dictionary based `element_formatters` approach. We use the above `dist` table to provide a specific callback that will set the background color for diagonal cells. We first write a function that takes the cell value and coordinates, returning the html formmatted text.

```
>>> def format_cell(value, row_num, col_num):
...     bgcolor=['', ' bgcolor="#0055ff"'][value=='']
...     return '<td%s>%s</td>' % (bgcolor, value)
```

We then call the method, without this argument, then with it.

```
>>> straight_html = dist.to_rich_html(compact=True)
>>> print(straight_html)
<table><tr><th>seq1/2</th><th>a...
>>> rich_html = dist.to_rich_html(row_cell_func=format_cell,
...                                compact=False)
>>> print(rich_html)
<table>
<tr>
<th>seq1/2</th>
<th>a</th>
<th>c</th>
<th>b</th>
<th>e</th>
</tr>
<tr>
<td>a</td>
<td bgcolor="#0055ff"></td>
<td>0.0883</td>...
```

Convert Table to pandas DataFrame

If you have `pandas` installed, you can convert a `Table` instance to a `DataFrame`.

```
>>> tbl = Table(header=['a', 'b'], rows=[[0, 1], [3,7]])
>>> df = tbl.to_pandas_df()
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
>>> print(df)
```

```
   a  b
0  0  1
1  3  7
```

Exporting bedGraph format

One export format available is bedGraph. This format can be used for viewing data as annotation track in a genome browser. This format allows for unequal spans and merges adjacent spans with the same value. The format has many possible arguments that modify the appearance in the genome browser. For this example we just create a simple data set.

```
>>> rows = [['1', 100, 101, 1.123], ['1', 101, 102, 1.123],
...         ['1', 102, 103, 1.123], ['1', 103, 104, 1.123],
...         ['1', 104, 105, 1.123], ['1', 105, 106, 1.123],
...         ['1', 106, 107, 1.123], ['1', 107, 108, 1.123],
...         ['1', 108, 109, 1], ['1', 109, 110, 1],
...         ['1', 110, 111, 1], ['1', 111, 112, 1],
...         ['1', 112, 113, 1], ['1', 113, 114, 1],
...         ['1', 114, 115, 1], ['1', 115, 116, 1],
...         ['1', 116, 117, 1], ['1', 117, 118, 1],
...         ['1', 118, 119, 2], ['1', 119, 120, 2],
...         ['1', 120, 121, 2], ['1', 150, 151, 2],
...         ['1', 151, 152, 2], ['1', 152, 153, 2],
...         ['1', 153, 154, 2], ['1', 154, 155, 2],
...         ['1', 155, 156, 2], ['1', 156, 157, 2],
...         ['1', 157, 158, 2], ['1', 158, 159, 2],
...         ['1', 159, 160, 2], ['1', 160, 161, 2]]
...
>>> bgraph = LoadTable(header=['chrom', 'start', 'end', 'value'],
...                    rows=rows)
...
>>> print(bgraph.tostring(format='bedgraph', name='test track',
...     graphType='bar', description='test of bedgraph', color=(255,0,0)))
track type=bedGraph name="test track" description="test of bedgraph" color=255,0,0␣
↪graphType=bar
1   100     108     1.12
1   108     118     1.00
1   118     161     2.00
```

The bedgraph formatter defaults to rounding values to 2 decimal places. You can adjust that precision using the `digits` argument.

```
>>> print(bgraph.tostring(format='bedgraph', name='test track',
...     graphType='bar', description='test of bedgraph', color=(255,0,0),
...     digits=0))
track type=bedGraph name="test track" description="test of bedgraph" color=255,0,0␣
↪graphType=bar
1   100     118     1.00
1   118     161     2.00
```

Note: Writing files in bedgraph format is done using the `write(format='bedgraph', name='test track', description='test of bedgraph', color=(255,0,0))`.

Saving a table for reloading

Saving a table object to file for later reloading can be done using the standard `write` method and `filename` argument to the `Table` constructor, specifying any of the formats supported by `tostring`. The table loading will recreate a table from raw data located at `filename`. To illustrate this, we first write out the table `t3` in `pickle` format, then the table `t2` in a csv (comma separated values format). We then remove it's header and write/reload as a tsv (tab separated values format).

```
>>> t3 = Table(['edge.name', 'edge.parent', 'length', 'x', 'y', 'z'], d2D,
...            row_order=row_order, missing_data='*', space=8,
...            max_width=50, row_ids=True, title='My title',
...            legend='legend: this is a nonsense example.')
>>> t3.write("t3.pickle")
>>> t3_loaded = LoadTable(filename="t3.pickle")
>>> print(t3_loaded)
My title
=========================================
edge.name        edge.parent        length
-----------------------------------------
    Human             edge.0        4.0000
HowlerMon             edge.0        4.0000
    Mouse             edge.1        4.0000
NineBande               root        4.0000
 DogFaced               root        4.0000
   edge.0             edge.1        4.0000
   edge.1               root        4.0000
-----------------------------------------

continued: My title
===================================
edge.name             x             y
-----------------------------------
    Human        1.0000        3.0000
HowlerMon        1.0000        3.0000
    Mouse        1.0000        3.0000
NineBande        1.0000        3.0000
 DogFaced        1.0000        3.0000
   edge.0        1.0000        3.0000
   edge.1        1.0000        3.0000
-----------------------------------

continued: My title
=======================
edge.name             z
-----------------------
    Human        6.0000
HowlerMon        6.0000
    Mouse        6.0000
NineBande        6.0000
 DogFaced        6.0000
   edge.0        6.0000
   edge.1        6.0000
-----------------------

legend: this is a nonsense example.
>>> t2 = Table(['abcd', 'data'], [[str([1, 2, 3, 4, 5]), '0'], ['x', 5.0],
... ['y', None]], missing_data='*', title='A \ntitle')
>>> t2.write('t2.csv')
```

```
>>> t2_loaded = LoadTable(filename='t2.csv', header=True, with_title=True)
>>> print(t2_loaded)
A
title
=========================
           abcd      data
-------------------------
[1, 2, 3, 4, 5]          0
              x     5.0000
              y
-------------------------
>>> t2.title = ""
>>> t2.write("t2.tsv")
>>> t2_loaded = LoadTable(filename='t2.tsv')
>>> print(t2_loaded)
=========================
           abcd      data
-------------------------
[1, 2, 3, 4, 5]          0
              x     5.0000
              y
-------------------------
```

Note the `missing_data` attribute is not saved in the delimited format, but is in the `pickle` format. In the next case, I'm going to override the digits format on reloading of the table.

```
>>> t2 = Table(['abcd', 'data'], [[str([1, 2, 3, 4, 5]), '0'], ['x', 5.0],
...             ['y', None]], missing_data='*', title='A \ntitle',
...             legend="And\na legend too")
>>> t2.write('t2.csv', sep=',')
>>> t2_loaded = LoadTable(filename='t2.csv', header=True, with_title=True,
...                       with_legend=True, sep=',', digits = 2)
>>> print(t2_loaded)
A
title
=======================
           abcd    data
-----------------------
[1, 2, 3, 4, 5]       0
              x    5.00
              y
-----------------------
And
a legend too
```

A few things to note about the delimited file saving: formatting arguments are lost in saving to a delimited format; the `header` argument specifies whether the first line of the file should be treated as the header; the `with_title` and `with_legend` arguments are necessary if the file contains them, otherwise they become the header or part of the table. Importantly, if you wish to preserve numerical precision use the `pickle` format.

`pickle` can load a useful object from the pickled `Table` by itself, without needing to know anything about the `Table` class.

```
>>> import pickle
>>> f = open("t3.pickle", "rb")
>>> pickled = pickle.load(f)
>>> f.close()
>>> sorted(pickled.keys())
```

```
['digits', 'format', 'header', 'legend', 'max_width', 'missing_data',...
>>> pickled['rows'][0]
['Human', 'edge.0', 4.0, 1.0, 3.0, 6.0]
```

We can read in a delimited format using a custom reader. There are two approaches. The first one allows specifying different type conversions for different columns. The second allows specifying a whole line-based parser.

You can also read and write tables in gzip compressed format. This can be done simply by ending a filename with '.gz' or specifying `compress=True`. We write a compressed file the two different ways and read it back in.

```
>>> t2.write('t2.csv.gz', sep=',')
>>> t2_gz = LoadTable('t2.csv.gz', sep=',', with_title=True,
...                   with_legend=True)
>>> t2_gz.shape == t2.shape
True
>>> t2.write('t2.csv', sep=',', compress=True)
>>> t2_gz = LoadTable('t2.csv.gz', sep=',', with_title=True,
...                   with_legend=True)
>>> t2_gz.shape == t2.shape
True
```

Defining a custom reader with type conversion for each column

We convert columns 2-5 to floats by specifying a field convertor. We then create a reader, specifying the data (below a list but can be a file) properties. Note that if no convertor is provided all data are returned as strings. We can also provide this reader to the `Table` constructor for a more direct way of opening such files. In this case, `Table` assumes there is a header row and nothing else.

```
>>> from cogent3.parse.table import ConvertFields, SeparatorFormatParser
>>> t3.title = t3.legend = None
>>> comma_sep = t3.tostring(sep=",").splitlines()
>>> print(comma_sep)
['edge.name,edge.parent,length,     x,     y,     z', '    Human,    ...
>>> converter = ConvertFields([(2,float), (3,float), (4,float), (5, float)])
>>> reader = SeparatorFormatParser(with_header=True,converter=converter,
...       sep=",")
>>> comma_sep = [line for line in reader(comma_sep)]
>>> print(comma_sep)
[['edge.name', 'edge.parent', 'length', 'x', 'y', 'z'], ['Human',...
>>> t3.write("t3.tab", sep="\t")
>>> reader = SeparatorFormatParser(with_header=True,converter=converter,
...       sep="\t")
>>> t3a = LoadTable(filename="t3.tab", reader=reader, title="new title",
...        space=2)
...
>>> print(t3a)
new title
=========================================================
edge.name  edge.parent  length       x       y       z
---------------------------------------------------------
    Human       edge.0  4.0000  1.0000  3.0000  6.0000
HowlerMon       edge.0  4.0000  1.0000  3.0000  6.0000
    Mouse       edge.1  4.0000  1.0000  3.0000  6.0000
NineBande         root  4.0000  1.0000  3.0000  6.0000
 DogFaced         root  4.0000  1.0000  3.0000  6.0000
   edge.0       edge.1  4.0000  1.0000  3.0000  6.0000
```

```
   edge.1          root   4.0000   1.0000   3.0000   6.0000
-------------------------------------------------------------
```

We can use the `SeparatorFormatParser` to ignore reading certain lines by using a callback function. We illustrate this using the above data, skipping any rows with `edge.name` starting with `edge`.

```
>>> def ignore_internal_nodes(line):
...     return line[0].startswith('edge')
...
>>> reader = SeparatorFormatParser(with_header=True,converter=converter,
...      sep="\t", ignore=ignore_internal_nodes)
...
>>> tips = LoadTable(filename="t3.tab", reader=reader, digits=1, space=2)
>>> print(tips)
=============================================
edge.name  edge.parent  length    x    y    z
---------------------------------------------
    Human       edge.0     4.0  1.0  3.0  6.0
HowlerMon       edge.0     4.0  1.0  3.0  6.0
    Mouse       edge.1     4.0  1.0  3.0  6.0
NineBande         root     4.0  1.0  3.0  6.0
 DogFaced         root     4.0  1.0  3.0  6.0
---------------------------------------------
```

We can also limit the amount of data to be read in, very handy for checking large files.

```
>>> t3a = LoadTable("t3.tab", sep='\t', limit=3)
>>> print(t3a)
================================================================
edge.name    edge.parent    length         x         y         z
----------------------------------------------------------------
    Human         edge.0    4.0000    1.0000    3.0000    6.0000
HowlerMon         edge.0    4.0000    1.0000    3.0000    6.0000
    Mouse         edge.1    4.0000    1.0000    3.0000    6.0000
----------------------------------------------------------------
```

Limiting should also work when `static_column_types` is invoked

```
>>> t3a = LoadTable("t3.tab", sep='\t', limit=3, static_column_types=True)
>>> t3a.shape[0] == 3
True
```

or when

In the above example, the data type in a column is static, e.g. all values in `x` are floats. Rather than providing a custom reader, you can get the `Table` to construct such a reader based on the first data row using the `static_column_types` argument.

```
>>> t3a = LoadTable(filename="t3.tab", static_column_types=True, digits=1,
...                 sep='\t')
>>> print(t3a)
=====================================================
edge.name    edge.parent    length      x      y      z
-----------------------------------------------------
    Human         edge.0       4.0    1.0    3.0    6.0
HowlerMon         edge.0       4.0    1.0    3.0    6.0
    Mouse         edge.1       4.0    1.0    3.0    6.0
NineBande           root       4.0    1.0    3.0    6.0
```

```
 DogFaced           root       4.0    1.0    3.0    6.0
   edge.0         edge.1       4.0    1.0    3.0    6.0
   edge.1           root       4.0    1.0    3.0    6.0
-------------------------------------------------------
```

If you invoke the `static_column_types` argument and the column data are not static, you'll get a `ValueError`. We show this by first creating a simple table with mixed data types in a column, write to file and then try to load with `static_column_types=True`.

```
>>> t3b = LoadTable(header=['A', 'B'], rows=[[1,1], ['a', 2]], sep=2)
>>> print(t3b)
======
A    B
------
1    1
a    2
------
>>> t3b.write('test3b.txt', sep='\t')
>>> t3b = LoadTable('test3b.txt', sep='\t', static_column_types=True)
Traceback (most recent call last):
ValueError: invalid literal for int() with base 10: 'a'
```

We also test the reader function for a tab delimited format with missing data at the end.

```
>>> data = ['ab\tcd\t', 'ab\tcd\tef']
>>> tab_reader = SeparatorFormatParser(sep='\t')
>>> for line in tab_reader(data):
...     assert len(line) == 3, line
```

Defining a custom reader that operates on entire lines

It can also be the case that data types differ between lines. The basic mechanism is the same as above, except in defining the converter you must set the argument `by_column=True`.

We illustrate this capability by writing a short function that tries to cast entire lines to `int`, `float` or leaves as a string.

```
>>> def CastLine():
...     floats = lambda x: list(map(float, x))
...     ints = lambda x: list(map(int, x))
...     def call(line):
...         try:
...             line = ints(line)
...         except ValueError:
...             try:
...                 line = floats(line)
...             except ValueError:
...                 pass
...         return line
...     return call
```

We then define a couple of lines, create an instance of `ConvertFields` and call it for each type.

```
>>> line_str_ints = '\t'.join(map(str, range(5)))
>>> line_str_floats = '\t'.join(map(str, map(float, range(5))))
>>> data = [line_str_ints, line_str_floats]
```

```
>>> cv = ConvertFields(CastLine(), by_column=False)
>>> tab_reader = SeparatorFormatParser(with_header=False, converter=cv,
...                                    sep='\t')
>>> for line in tab_reader(data):
...     print(line)
[0, 1, 2, 3, 4]
[0.0, 1.0, 2.0, 3.0, 4.0]
```

Defining a custom writer

We can likewise specify a writer, using a custom field formatter and provide this to the `Table` directly for writing. We first illustrate how the writer works to generate output. We then use it to escape some text fields in quotes. In order to read that back in, we define a custom reader that strips these quotes off.

```
>>> from cogent3.format.table import FormatFields, SeparatorFormatWriter
>>> formatter = FormatFields([(0,'"%s"'), (1,'"%s"')])
>>> writer = SeparatorFormatWriter(formatter=formatter, sep=" | ")
>>> for formatted in writer(comma_sep, has_header=True):
...      print(formatted)
edge.name | edge.parent | length | x | y | z
"Human" | "edge.0" | 4.0 | 1.0 | 3.0 | 6.0
"HowlerMon" | "edge.0" | 4.0 | 1.0 | 3.0 | 6.0
"Mouse" | "edge.1" | 4.0 | 1.0 | 3.0 | 6.0
"NineBande" | "root" | 4.0 | 1.0 | 3.0 | 6.0
"DogFaced" | "root" | 4.0 | 1.0 | 3.0 | 6.0
"edge.0" | "edge.1" | 4.0 | 1.0 | 3.0 | 6.0
"edge.1" | "root" | 4.0 | 1.0 | 3.0 | 6.0
>>> t3.write(filename="t3.tab", writer=writer)
>>> strip = lambda x: x.replace('"', '')
>>> converter = ConvertFields([(0,strip), (1, strip)])
>>> reader = SeparatorFormatParser(with_header=True, converter=converter,
...       sep="|", strip_wspace=True)
>>> t3a = LoadTable(filename="t3.tab", reader=reader, title="new title",
...       space=2)
>>> print(t3a)
new title
================================================
edge.name  edge.parent  length    x    y    z
------------------------------------------------
    Human       edge.0     4.0  1.0  3.0  6.0
HowlerMon       edge.0     4.0  1.0  3.0  6.0
    Mouse       edge.1     4.0  1.0  3.0  6.0
NineBande         root     4.0  1.0  3.0  6.0
 DogFaced         root     4.0  1.0  3.0  6.0
   edge.0       edge.1     4.0  1.0  3.0  6.0
   edge.1         root     4.0  1.0  3.0  6.0
------------------------------------------------
```

Note: There are performance issues for large files. Pickling has proven very slow for saving very large files and introduces significant file size bloat. A simple delimited format is much more efficient both storage wise and, if you use a custom reader (or specify `static_column_types=True`), to generate and read. A custom reader was approximately 6 fold faster than the standard delimited file reader.

Table slicing and iteration

The Table class is capable of slicing by row, range of rows, column or range of columns headings or used to identify a single cell. Slicing using the method `get_columns` can also be used to reorder columns. In the case of columns, either the string headings or their position integers can be used. For rows, if `row_ids` was specified as `True` the 0'th cell in each row can also be used.

```
>>> t4 = Table(['edge.name', 'edge.parent', 'length', 'x', 'y', 'z'], d2D,
...            row_order=row_order, row_ids=True, title='My title')
```

We subset `t4` by column and reorder them.

```
>>> new = t4.get_columns(['z', 'y'])
>>> print(new)
My title
============================
edge.name          z          y
----------------------------
    Human     6.0000     3.0000
HowlerMon     6.0000     3.0000
    Mouse     6.0000     3.0000
NineBande     6.0000     3.0000
 DogFaced     6.0000     3.0000
   edge.0     6.0000     3.0000
   edge.1     6.0000     3.0000
----------------------------
```

We use the column position indexes to do get the same table.

```
>>> new = t4.get_columns([5, 4])
>>> print(new)
My title
============================
edge.name          z          y
----------------------------
    Human     6.0000     3.0000
HowlerMon     6.0000     3.0000
    Mouse     6.0000     3.0000
NineBande     6.0000     3.0000
 DogFaced     6.0000     3.0000
   edge.0     6.0000     3.0000
   edge.1     6.0000     3.0000
----------------------------
```

We can also using more general slicing, by both rows and columns. The following returns all rows from 4 on, and columns up to (but excluding) 'y':

```
>>> k = t4[4:, :'y']
>>> print(k)
My title
=========================================
edge.name    edge.parent    length         x
-----------------------------------------
 DogFaced           root    4.0000    1.0000
   edge.0         edge.1    4.0000    1.0000
   edge.1           root    4.0000    1.0000
-----------------------------------------
```

We can explicitly reference individual cells, in this case using both row and column keys.

```
>>> val = t4['HowlerMon', 'y']
>>> print(val)
3.0
```

We slice a single row,

```
>>> new = t4[3]
>>> print(new)
My title
=================================================================
edge.name    edge.parent    length         x         y         z
-----------------------------------------------------------------
NineBande           root    4.0000    1.0000    3.0000    6.0000
-----------------------------------------------------------------
```

and range of rows.

```
>>> new = t4[3:6]
>>> print(new)
My title
=================================================================
edge.name    edge.parent    length         x         y         z
-----------------------------------------------------------------
NineBande           root    4.0000    1.0000    3.0000    6.0000
 DogFaced           root    4.0000    1.0000    3.0000    6.0000
   edge.0         edge.1    4.0000    1.0000    3.0000    6.0000
-----------------------------------------------------------------
```

You can get disjoint rows.

```
>>> print(t4.get_disjoint_rows(['Human', 'Mouse', 'DogFaced']))
My title
=================================================================
edge.name    edge.parent    length         x         y         z
-----------------------------------------------------------------
    Human         edge.0    4.0000    1.0000    3.0000    6.0000
    Mouse         edge.1    4.0000    1.0000    3.0000    6.0000
 DogFaced           root    4.0000    1.0000    3.0000    6.0000
-----------------------------------------------------------------
```

You can iterate over the table one row at a time and slice the rows. We illustrate this for slicing a single column,

```
>>> for row in t:
...     print(row['stableid'])
ENSG00000005893
ENSG00000019485
ENSG00000019102...
```

and for multiple columns.

```
>>> for row in t:
...     print(row['stableid'], row['length'])
ENSG00000005893 1353
ENSG00000019485 1827
ENSG00000019102 999...
```

The numerical slice equivalent to the first case above would be `row[0]`, to the second case either `row[:]`, `row[:2]`.

Filtering tables - selecting subsets of rows/columns

We want to be able to slice a table, based on some condition(s), to produce a new subset table. For instance, we construct a table with type and probability values.

```
>>> header = ['Gene', 'type', 'LR', 'df', 'Prob']
>>> rows = (('NP_003077_hs_mm_rn_dna', 'Con', 2.5386, 1, 0.1111),
...         ('NP_004893_hs_mm_rn_dna', 'Con', 0.1214, 1, 0.7276),
...         ('NP_005079_hs_mm_rn_dna', 'Con', 0.9517, 1, 0.3293),
...         ('NP_005500_hs_mm_rn_dna', 'Con', 0.7383, 1, 0.3902),
...         ('NP_055852_hs_mm_rn_dna', 'Con', 0.0000, 1, 0.9997),
...         ('NP_057012_hs_mm_rn_dna', 'Unco', 34.3081, 1, 0.0000),
...         ('NP_061130_hs_mm_rn_dna', 'Unco', 3.7986, 1, 0.0513),
...         ('NP_065168_hs_mm_rn_dna', 'Con', 89.9766, 1, 0.0000),
...         ('NP_065396_hs_mm_rn_dna', 'Unco', 11.8912, 1, 0.0006),
...         ('NP_109590_hs_mm_rn_dna', 'Con', 0.2121, 1, 0.6451),
...         ('NP_116116_hs_mm_rn_dna', 'Unco', 9.7474, 1, 0.0018))
>>> t5 = Table(header, rows)
>>> print(t5)
============================================================
                  Gene    type          LR    df      Prob
------------------------------------------------------------
NP_003077_hs_mm_rn_dna     Con      2.5386     1    0.1111
NP_004893_hs_mm_rn_dna     Con      0.1214     1    0.7276
NP_005079_hs_mm_rn_dna     Con      0.9517     1    0.3293
NP_005500_hs_mm_rn_dna     Con      0.7383     1    0.3902
NP_055852_hs_mm_rn_dna     Con      0.0000     1    0.9997
NP_057012_hs_mm_rn_dna    Unco     34.3081     1    0.0000
NP_061130_hs_mm_rn_dna    Unco      3.7986     1    0.0513
NP_065168_hs_mm_rn_dna     Con     89.9766     1    0.0000
NP_065396_hs_mm_rn_dna    Unco     11.8912     1    0.0006
NP_109590_hs_mm_rn_dna     Con      0.2121     1    0.6451
NP_116116_hs_mm_rn_dna    Unco      9.7474     1    0.0018
------------------------------------------------------------
```

We then seek to obtain only those rows that contain probabilities < 0.05. We use valid python code within a string. **Note:** Make sure your column headings could be valid python variable names or the string based approach will fail (you could use an external function instead, see below).

```
>>> sub_table1 = t5.filtered(callback="Prob < 0.05")
>>> print(sub_table1)
============================================================
                  Gene    type          LR    df      Prob
------------------------------------------------------------
NP_057012_hs_mm_rn_dna    Unco     34.3081     1    0.0000
NP_065168_hs_mm_rn_dna     Con     89.9766     1    0.0000
NP_065396_hs_mm_rn_dna    Unco     11.8912     1    0.0006
NP_116116_hs_mm_rn_dna    Unco      9.7474     1    0.0018
------------------------------------------------------------
```

Using the above table we test the function to extract the raw data for a single column,

```
>>> raw = sub_table1.tolist('LR')
>>> raw
[34.3081..., 89.9766..., 11.8912, 9.7474...]
```

and from multiple columns.

```
>>> raw = sub_table1.tolist(columns=['df', 'Prob'])
>>> raw
[[1, 0.0], [1, 0.0],...
```

We can also do filtering using an external function, in this case we use a `lambda` to obtain only those rows of type 'Unco' that contain probabilities < 0.05, modifying our callback function.

```
>>> sub_table2 = t5.filtered(
...                 lambda ty_pr: ty_pr[0] == 'Unco' and ty_pr[1] < 0.05,
...                 columns=('type', 'Prob')
...                 )
>>> print(sub_table2)
=============================================================
                  Gene    type          LR    df      Prob
-------------------------------------------------------------
NP_057012_hs_mm_rn_dna    Unco    34.3081     1    0.0000
NP_065396_hs_mm_rn_dna    Unco    11.8912     1    0.0006
NP_116116_hs_mm_rn_dna    Unco     9.7474     1    0.0018
-------------------------------------------------------------
```

This can also be done using the string approach.

```
>>> sub_table2 = t5.filtered("type == 'Unco' and Prob < 0.05")
>>> print(sub_table2)
=============================================================
                  Gene    type          LR    df      Prob
-------------------------------------------------------------
NP_057012_hs_mm_rn_dna    Unco    34.3081     1    0.0000
NP_065396_hs_mm_rn_dna    Unco    11.8912     1    0.0006
NP_116116_hs_mm_rn_dna    Unco     9.7474     1    0.0018
-------------------------------------------------------------
```

We can also filter table columns using `filtered_by_column`. Say we only want the numerical columns, we can write a callback that returns `False` if some numerical operation fails, `True` otherwise.

```
>>> def is_numeric(values):
...     try:
...         sum(values)
...     except TypeError:
...         return False
...     return True
>>> print(t5.filtered_by_column(callback=is_numeric))
=======================
     LR    df      Prob
-----------------------
 2.5386     1    0.1111
 0.1214     1    0.7276
 0.9517     1    0.3293
 0.7383     1    0.3902
 0.0000     1    0.9997
34.3081     1    0.0000
 3.7986     1    0.0513
89.9766     1    0.0000
11.8912     1    0.0006
 0.2121     1    0.6451
 9.7474     1    0.0018
-----------------------
```

Appending tables

Tables may also be appended to each other, to make larger tables. We'll construct two simple tables to illustrate this.

```
>>> geneA = Table(['edge.name', 'edge.parent', 'z'], [['Human','root',
... 6.0],['Mouse','root', 6.0], ['Rat','root', 6.0]],
... title='Gene A')
>>> geneB = Table(['edge.name', 'edge.parent', 'z'], [['Human','root',
... 7.0],['Mouse','root', 7.0], ['Rat','root', 7.0]],
... title='Gene B')
>>> print(geneB)
Gene B
==================================
edge.name    edge.parent          z
----------------------------------
    Human           root     7.0000
    Mouse           root     7.0000
      Rat           root     7.0000
----------------------------------
```

we now use the `appended` Table method to create a new table, specifying that we want a new column created (by passing the `new_column` argument a heading) in which the table titles will be placed.

```
>>> new = geneA.appended('Gene', geneB, title='Appended tables')
>>> print(new)
Appended tables
==========================================
  Gene    edge.name    edge.parent          z
------------------------------------------
Gene A        Human           root     6.0000
Gene A        Mouse           root     6.0000
Gene A          Rat           root     6.0000
Gene B        Human           root     7.0000
Gene B        Mouse           root     7.0000
Gene B          Rat           root     7.0000
------------------------------------------
```

We repeat this without adding a new column.

```
>>> new = geneA.appended(None, geneB, title="Appended, no new column")
>>> print(new)
Appended, no new column
==================================
edge.name    edge.parent          z
----------------------------------
    Human           root     6.0000
    Mouse           root     6.0000
      Rat           root     6.0000
    Human           root     7.0000
    Mouse           root     7.0000
      Rat           root     7.0000
----------------------------------
```

Miscellaneous

Tables have a `shape` attribute, which specifies *x* (number of columns) and *y* (number of rows). The attribute is a tuple and we illustrate it for the above `sub_table` tables. Combined with the `filtered` method, this attribute can tell

you how many rows satisfy a specific condition.

```
>>> t5.shape
(11, 5)
>>> sub_table1.shape
(4, 5)
>>> sub_table2.shape
(3, 5)
```

For instance, 3 of the 11 rows in `t` were significant and belonged to the `Unco` type.

For completeness, we generate a table with no rows and assess its shape.

```
>>> sub_table3 = t5.filtered(
...                 lambda ty_pr: ty_pr[0] == 'Unco' and ty_pr[1] > 0.1,
...                 columns=('type', 'Prob'))
>>> sub_table3.shape
(0, 5)
```

The distinct values can be obtained for a single column,

```
>>> distinct = new.distinct_values("edge.name")
>>> assert distinct == set(['Rat', 'Mouse', 'Human'])
```

or multiple columns

```
>>> distinct = new.distinct_values(["edge.parent", "z"])
>>> assert distinct == set([('root', 6.0), ('root', 7.0)]), distinct
```

We can compute column sums. Assuming only numerical values in a column.

```
>>> assert new.summed('z') == 39., new.summed('z')
```

We construct an example with mixed numerical and non-numerical data. We now compute the column sum with mixed non-numerical/numerical data.

```
>>> mix = LoadTable(header=['A', 'B'], rows=[[0,''],[1,2],[3,4]])
>>> print(mix)
======
A    B
------
0
1    2
3    4
------
>>> mix.summed('B', strict=False)
6
```

We also compute row sums for the pure numerical and mixed non-numerical/numerical rows. For summing across rows we must specify the actual row index as an `int`.

```
>>> mix.summed(0, col_sum=False, strict=False)
0
>>> mix.summed(1, col_sum=False)
3
```

We can compute the totals for all columns or rows too.

```
>>> mix.summed(strict=False)
[4, 6]
>>> mix.summed(col_sum=False, strict=False)
[0, 3, 7]
```

It is not currently possible to do a subset of columns/rows. We show this for rows here.

```
>>> mix.summed([0, 2], col_sum=False, strict=False)
Traceback (most recent call last):
RuntimeError: unknown indices type: [0, 2]
```

We test these for a strictly numerical table.

```
>>> non_mix = LoadTable(header=['A', 'B'], rows=[[0,1],[1,2],[3,4]])
>>> non_mix.summed()
[4, 7]
>>> non_mix.summed(col_sum=False)
[1, 3, 7]
```

We can normalise a numerical table by row,

```
>>> print(non_mix.normalized(by_row=True))
================
     A         B
----------------
0.0000    1.0000
0.3333    0.6667
0.4286    0.5714
----------------
```

or by column, such that the row/column sums are 1.

```
>>> print(non_mix.normalized(by_row=False))
================
     A         B
----------------
0.0000    0.1429
0.2500    0.2857
0.7500    0.5714
----------------
```

We normalize by an arbitrary function (maximum value) by row,

```
>>> print(non_mix.normalized(by_row=True, denominator_func=max))
================
     A         B
----------------
0.0000    1.0000
0.5000    1.0000
0.7500    1.0000
----------------
```

by column.

```
>>> print(non_mix.normalized(by_row=False, denominator_func=max))
================
     A         B
----------------
```

```
0.0000    0.2500
0.3333    0.5000
1.0000    1.0000
----------------
```

Extending tables

In some cases it is desirable to compute an additional column from existing column values. This is done using the `with_new_column` method. We'll use t4 from above, adding two of the columns to create an additional column.

```
>>> t7 = t4.with_new_column('Sum', callback="z+x", digits=2)
>>> print(t7)
My title
=================================================================
edge.name    edge.parent    length       x       y       z     Sum
-----------------------------------------------------------------
    Human         edge.0      4.00    1.00    3.00    6.00    7.00
HowlerMon         edge.0      4.00    1.00    3.00    6.00    7.00
    Mouse         edge.1      4.00    1.00    3.00    6.00    7.00
NineBande           root      4.00    1.00    3.00    6.00    7.00
 DogFaced           root      4.00    1.00    3.00    6.00    7.00
   edge.0         edge.1      4.00    1.00    3.00    6.00    7.00
   edge.1           root      4.00    1.00    3.00    6.00    7.00
-----------------------------------------------------------------
```

We test this with an externally defined function.

```
>>> func = lambda x_y: x_y[0] * x_y[1]
>>> t7 = t4.with_new_column('Sum', callback=func, columns=("y","z"),
... digits=2)
>>> print(t7)
My title
=================================================================
edge.name    edge.parent    length       x       y       z      Sum
-----------------------------------------------------------------
    Human         edge.0      4.00    1.00    3.00    6.00    18.00
HowlerMon         edge.0      4.00    1.00    3.00    6.00    18.00
    Mouse         edge.1      4.00    1.00    3.00    6.00    18.00
NineBande           root      4.00    1.00    3.00    6.00    18.00
 DogFaced           root      4.00    1.00    3.00    6.00    18.00
   edge.0         edge.1      4.00    1.00    3.00    6.00    18.00
   edge.1           root      4.00    1.00    3.00    6.00    18.00
-----------------------------------------------------------------
>>> func = lambda x: x**3
>>> t7 = t4.with_new_column('Sum', callback=func, columns="y", digits=2)
>>> print(t7)
My title
=================================================================
edge.name    edge.parent    length       x       y       z      Sum
-----------------------------------------------------------------
    Human         edge.0      4.00    1.00    3.00    6.00    27.00
HowlerMon         edge.0      4.00    1.00    3.00    6.00    27.00
    Mouse         edge.1      4.00    1.00    3.00    6.00    27.00
NineBande           root      4.00    1.00    3.00    6.00    27.00
 DogFaced           root      4.00    1.00    3.00    6.00    27.00
   edge.0         edge.1      4.00    1.00    3.00    6.00    27.00
```

```
   edge.1            root       4.00     1.00     3.00     6.00    27.00
-------------------------------------------------------------------------
```

Sorting tables

We want a table sorted according to values in a column.

```
>>> sorted = t5.sorted(columns='LR')
>>> print(sorted)
=========================================================
                 Gene     type          LR     df       Prob
---------------------------------------------------------
NP_055852_hs_mm_rn_dna      Con      0.0000      1     0.9997
NP_004893_hs_mm_rn_dna      Con      0.1214      1     0.7276
NP_109590_hs_mm_rn_dna      Con      0.2121      1     0.6451
NP_005500_hs_mm_rn_dna      Con      0.7383      1     0.3902
NP_005079_hs_mm_rn_dna      Con      0.9517      1     0.3293
NP_003077_hs_mm_rn_dna      Con      2.5386      1     0.1111
NP_061130_hs_mm_rn_dna     Unco      3.7986      1     0.0513
NP_116116_hs_mm_rn_dna     Unco      9.7474      1     0.0018
NP_065396_hs_mm_rn_dna     Unco     11.8912      1     0.0006
NP_057012_hs_mm_rn_dna     Unco     34.3081      1     0.0000
NP_065168_hs_mm_rn_dna      Con     89.9766      1     0.0000
---------------------------------------------------------
```

We want a table sorted according to values in a subset of columns, note the order of columns determines the sort order.

```
>>> sorted = t5.sorted(columns=('LR', 'type'))
>>> print(sorted)
=========================================================
                 Gene     type          LR     df       Prob
---------------------------------------------------------
NP_055852_hs_mm_rn_dna      Con      0.0000      1     0.9997
NP_004893_hs_mm_rn_dna      Con      0.1214      1     0.7276
NP_109590_hs_mm_rn_dna      Con      0.2121      1     0.6451
NP_005500_hs_mm_rn_dna      Con      0.7383      1     0.3902
NP_005079_hs_mm_rn_dna      Con      0.9517      1     0.3293
NP_003077_hs_mm_rn_dna      Con      2.5386      1     0.1111
NP_061130_hs_mm_rn_dna     Unco      3.7986      1     0.0513
NP_116116_hs_mm_rn_dna     Unco      9.7474      1     0.0018
NP_065396_hs_mm_rn_dna     Unco     11.8912      1     0.0006
NP_057012_hs_mm_rn_dna     Unco     34.3081      1     0.0000
NP_065168_hs_mm_rn_dna      Con     89.9766      1     0.0000
---------------------------------------------------------
```

We now do a sort based on 2 columns.

```
>>> sorted = t5.sorted(columns=('type', 'LR'))
>>> print(sorted)
=========================================================
                 Gene     type          LR     df       Prob
---------------------------------------------------------
NP_055852_hs_mm_rn_dna      Con      0.0000      1     0.9997
NP_004893_hs_mm_rn_dna      Con      0.1214      1     0.7276
NP_109590_hs_mm_rn_dna      Con      0.2121      1     0.6451
NP_005500_hs_mm_rn_dna      Con      0.7383      1     0.3902
```

```
NP_005079_hs_mm_rn_dna      Con      0.9517      1     0.3293
NP_003077_hs_mm_rn_dna      Con      2.5386      1     0.1111
NP_065168_hs_mm_rn_dna      Con     89.9766      1     0.0000
NP_061130_hs_mm_rn_dna     Unco      3.7986      1     0.0513
NP_116116_hs_mm_rn_dna     Unco      9.7474      1     0.0018
NP_065396_hs_mm_rn_dna     Unco     11.8912      1     0.0006
NP_057012_hs_mm_rn_dna     Unco     34.3081      1     0.0000
------------------------------------------------------------
```

Reverse sort a single column

```
>>> sorted = t5.sorted('LR', reverse='LR')
>>> print(sorted)
============================================================
                  Gene     type          LR     df       Prob
------------------------------------------------------------
NP_065168_hs_mm_rn_dna      Con     89.9766      1     0.0000
NP_057012_hs_mm_rn_dna     Unco     34.3081      1     0.0000
NP_065396_hs_mm_rn_dna     Unco     11.8912      1     0.0006
NP_116116_hs_mm_rn_dna     Unco      9.7474      1     0.0018
NP_061130_hs_mm_rn_dna     Unco      3.7986      1     0.0513
NP_003077_hs_mm_rn_dna      Con      2.5386      1     0.1111
NP_005079_hs_mm_rn_dna      Con      0.9517      1     0.3293
NP_005500_hs_mm_rn_dna      Con      0.7383      1     0.3902
NP_109590_hs_mm_rn_dna      Con      0.2121      1     0.6451
NP_004893_hs_mm_rn_dna      Con      0.1214      1     0.7276
NP_055852_hs_mm_rn_dna      Con      0.0000      1     0.9997
------------------------------------------------------------
```

Sort by just specifying the `reverse` column

```
>>> sorted = t5.sorted(reverse='LR')
>>> print(sorted)
============================================================
                  Gene     type          LR     df       Prob
------------------------------------------------------------
NP_065168_hs_mm_rn_dna      Con     89.9766      1     0.0000
NP_057012_hs_mm_rn_dna     Unco     34.3081      1     0.0000
NP_065396_hs_mm_rn_dna     Unco     11.8912      1     0.0006
NP_116116_hs_mm_rn_dna     Unco      9.7474      1     0.0018
NP_061130_hs_mm_rn_dna     Unco      3.7986      1     0.0513
NP_003077_hs_mm_rn_dna      Con      2.5386      1     0.1111
NP_005079_hs_mm_rn_dna      Con      0.9517      1     0.3293
NP_005500_hs_mm_rn_dna      Con      0.7383      1     0.3902
NP_109590_hs_mm_rn_dna      Con      0.2121      1     0.6451
NP_004893_hs_mm_rn_dna      Con      0.1214      1     0.7276
NP_055852_hs_mm_rn_dna      Con      0.0000      1     0.9997
------------------------------------------------------------
```

Reverse sort one column but not another

```
>>> sorted = t5.sorted(columns=('type', 'LR'), reverse='LR')
>>> print(sorted)
============================================================
                  Gene     type          LR     df       Prob
------------------------------------------------------------
NP_065168_hs_mm_rn_dna      Con     89.9766      1     0.0000
NP_003077_hs_mm_rn_dna      Con      2.5386      1     0.1111
```

```
NP_005079_hs_mm_rn_dna      Con      0.9517      1     0.3293
NP_005500_hs_mm_rn_dna      Con      0.7383      1     0.3902
NP_109590_hs_mm_rn_dna      Con      0.2121      1     0.6451
NP_004893_hs_mm_rn_dna      Con      0.1214      1     0.7276
NP_055852_hs_mm_rn_dna      Con      0.0000      1     0.9997
NP_057012_hs_mm_rn_dna     Unco     34.3081      1     0.0000
NP_065396_hs_mm_rn_dna     Unco     11.8912      1     0.0006
NP_116116_hs_mm_rn_dna     Unco      9.7474      1     0.0018
NP_061130_hs_mm_rn_dna     Unco      3.7986      1     0.0513
-----------------------------------------------------------
```

Reverse sort both columns.

```
>>> sorted = t5.sorted(columns=('type', 'LR'), reverse=('type', 'LR'))
>>> print(sorted)
===========================================================
                  Gene     type          LR    df      Prob
-----------------------------------------------------------
NP_057012_hs_mm_rn_dna     Unco     34.3081     1     0.0000
NP_065396_hs_mm_rn_dna     Unco     11.8912     1     0.0006
NP_116116_hs_mm_rn_dna     Unco      9.7474     1     0.0018
NP_061130_hs_mm_rn_dna     Unco      3.7986     1     0.0513
NP_065168_hs_mm_rn_dna      Con     89.9766     1     0.0000
NP_003077_hs_mm_rn_dna      Con      2.5386     1     0.1111
NP_005079_hs_mm_rn_dna      Con      0.9517     1     0.3293
NP_005500_hs_mm_rn_dna      Con      0.7383     1     0.3902
NP_109590_hs_mm_rn_dna      Con      0.2121     1     0.6451
NP_004893_hs_mm_rn_dna      Con      0.1214     1     0.7276
NP_055852_hs_mm_rn_dna      Con      0.0000     1     0.9997
-----------------------------------------------------------
```

Joining Tables

The Table object is capable of joins or merging of records in two tables. There are two fundamental types of joins – inner and outer – with there being different sub-types. We demonstrate these first constructing some simple tables.

```
>>> a=Table(header=["index", "col2","col3"],
...         rows=[[1,2,3],[2,3,1],[2,6,5]], title="A")
>>> print(a)
A
=====================
index    col2    col3
---------------------
    1       2       3
    2       3       1
    2       6       5
---------------------
>>> b=Table(header=["index", "col2","col3"],
...         rows=[[1,2,3],[2,2,1],[3,6,3]], title="B")
>>> print(b)
B
=====================
index    col2    col3
---------------------
    1       2       3
    2       2       1
```

```
    3       6       3
--------------------
>>> c=Table(header=["index","col_c2"],rows=[[1,2],[3,2],[3,5]],title="C")
>>> print(c)
C
===============
index    col_c2
---------------
    1         2
    3         2
    3         5
---------------
```

For a natural inner join, only 1 copy of columns with the same name are retained. So we expect the headings to be identical between the table `a`/`b` and the result of `a.joined(b)` or `b.joined(a)`.

```
>>> assert a.joined(b).header == b.header
>>> assert b.joined(a).header == a.header
```

For a standard inner join, the joined table should contain all columns from `a` and `b` excepting the index column(s). Simply providing a column name (or index) selects this behaviour. Note that in this case, column names from the second table are made unique by prefixing them with that tables title. If the provided tables do not have a title, a `RuntimeError` is raised.

```
>>> b.title = None
>>> try:
...     a.joined(b)
... except RuntimeError:
...     pass
>>> b.title = 'B'
>>> assert a.joined(b, "index").header == ["index", "col2", "col3",
...                                        "B_col2", "B_col3"]
...
```

Note that the table title's were used to prefix the column headings from the second table. We further test this using table `c` which has different dimensions.

```
>>> assert a.joined(c,"index").header == ["index","col2","col3",
...                                       "C_col_c2"]
```

It's also possible to specify index columns using numerical values, the results of which should be the same.

```
>>> assert a.joined(b,[0, 2]).tolist() ==\
...                          a.joined(b,["index","col3"]).tolist()
```

Additionally, it's possible to provide two series of indices for the two tables. Here, they have identical values.

```
>>> assert a.joined(b, ["index", "col3"],["index", "col3"]).tolist()\
...         == a.joined(b,["index","col3"]).tolist()
```

The results of a standard join between tables `a` and `b` are

```
>>> print(a.joined(b, ["index"], title='A&B'))
A&B
=======================================
index    col2    col3    B_col2    B_col3
---------------------------------------
```

```
    1       2       3          2          3
    2       3       1          2          1
    2       6       5          2          1
-----------------------------------------
```

We demo the table specific indices.

```
>>> print(a.joined(c, ["col2"], ["index"], title='A&C by "col2/index"'))
A&C by "col2/index"
================================
index    col2    col3    C_col_c2
--------------------------------
    2       3       1           2
    2       3       1           5
--------------------------------
```

Tables `a` and `c` share a single row with the same value in the `index` column, hence a join by that index should return a table with just that row.

```
>>> print(a.joined(c, "index", title='A&C by "index"'))
A&C by "index"
================================
index    col2    col3    C_col_c2
--------------------------------
    1       2       3           2
--------------------------------
```

A natural join of tables `a` and `b` results in a table with only rows that were identical between the two parents.

```
>>> print(a.joined(b, title='A&B Natural Join'))
A&B Natural Join
====================
index    col2    col3
--------------------
    1       2       3
--------------------
```

We test the outer join by defining an additional table with different dimensions, and conducting a join specifying `inner_join=False`.

```
>>> d=Table(header=["index", "col_c2"], rows=[[5,42],[6,23]], title="D")
>>> print(d)
D
===============
index    col_c2
---------------
    5        42
    6        23
---------------
>>> print(c.joined(d,inner_join=False, title='C&D Outer join'))
C&D Outer join
====================================
index    col_c2    D_index    D_col_c2
------------------------------------
    1         2          5          42
    1         2          6          23
    3         2          5          42
    3         2          6          23
```

```
    3          5          5          42
    3          5          6          23
---------------------------------------
```

We establish the `joined` method works for mixtures of character and numerical data, setting some indices and some cell values to be strings.

```
>>> a=Table(header=["index", "col2","col3"],
...         rows=[[1,2,"3"],["2",3,1],[2,6,5]], title="A")
>>> b=Table(header=["index", "col2","col3"],
...         rows=[[1,2,"3"],["2",2,1],[3,6,3]], title="B")
>>> assert a.joined(b, ["index", "col3"],["index", "col3"]).tolist()\
...         == a.joined(b,["index","col3"]).tolist()
```

We test that the `joined` method works when the column index orders differ.

```
>>> t1_header = ['a', 'b']
>>> t1_rows = [(1,2),(3,4)]
>>> t2_header = ['b', 'c']
>>> t2_rows = [(3,6),(4,8)]
>>> t1 = Table(t1_header, rows=t1_rows, title='t1')
>>> t2 = Table(t2_header, rows=t2_rows, title='t2')
>>> t3 = t1.joined(t2, columns_self=["b"], columns_other=["b"])
>>> print(t3)
==============
a    b    t2_c
--------------
3    4       8
--------------
```

We then establish that a join with no values does not cause a failure, just returns an empty `Table`.

```
>>> t4_header = ['b', 'c']
>>> t4_rows = [(5,6),(7,8)]
>>> t4 = LoadTable(header=t4_header, rows=t4_rows)
>>> t4.title = 't4'
>>> t5 = t1.joined(t4, columns_self=["b"], columns_other=["b"])
>>> print(t5)
==============
a    b    t4_c
--------------
--------------
```

Whose representation looks like

```
>>> t5
Table(numrows=0, numcols=3, header=['a', 'b', 't4_c'], rows=[])
```

Transposing a table

Tables can be transposed.

```
>>> from cogent3 import LoadTable
>>> title='#Full OTU Counts'
>>> header = ['#OTU ID', '14SK041', '14SK802']
>>> rows = [[-2920, '332', 294],
```

```
...          [-1606, '302', 229],
...          [-393, 141, 125],
...          [-2109, 138, 120],
...          [-5439, 104, 117],
...          [-1834, 70, 75],
...          [-18588, 65, 47],
...          [-1350, 60, 113],
...          [-2160, 57, 52],
...          [-11632, 47, 36]]
>>> table = LoadTable(header=header,rows=rows,title=title)
>>> print(table)
#Full OTU Counts
============================
#OTU ID    14SK041    14SK802
----------------------------
  -2920        332        294
  -1606        302        229
   -393        141        125
  -2109        138        120
  -5439        104        117
  -1834         70         75
 -18588         65         47
  -1350         60        113
  -2160         57         52
 -11632         47         36
----------------------------
```

We now transpose this. We require a new column heading for header data and an identifier for which existing column will become the header (default is index 0).

```
>>> tp = table.transposed(new_column_name='sample',
...              select_as_header='#OTU ID', space=2)
...
>>> print(tp)
=================================================================================
 sample  -2920  -1606  -393  -2109  -5439  -1834  -18588  -1350  -2160  -11632
---------------------------------------------------------------------------------
14SK041    332    302   141    138    104     70      65     60     57      47
14SK802    294    229   125    120    117     75      47    113     52      36
---------------------------------------------------------------------------------
```

We test transposition with default value is the same.

```
>>> tp = table.transposed(new_column_name='sample', space=2)
...
>>> print(tp)
=================================================================================
 sample  -2920  -1606  -393  -2109  -5439  -1834  -18588  -1350  -2160  -11632
---------------------------------------------------------------------------------
14SK041    332    302   141    138    104     70      65     60     57      47
14SK802    294    229   125    120    117     75      47    113     52      36
---------------------------------------------------------------------------------
```

We test transposition selecting a different column to become the header.

```
>>> tp = table.transposed(new_column_name='sample',
...              select_as_header='14SK802', space=2)
...
```

```
>>> print(tp)
=============================================================================
 sample    294    229   125    120    117     75      47    113     52      36
-----------------------------------------------------------------------------
#OTU ID  -2920  -1606  -393  -2109  -5439  -1834  -18588  -1350  -2160  -11632
14SK041    332    302   141    138    104     70      65     60     57      47
-----------------------------------------------------------------------------
```

Counting rows

We can count the number of rows for which a condition holds. This method uses the same arguments as `filtered` but returns an integer result only.

```
>>> print(c.count("col_c2 == 2"))
2
>>> print(c.joined(d,inner_join=False).count("index==3 and D_index==5"))
2
```

Testing a sub-component

Before using `Table`, we exercise some formatting code:

```
>>> from cogent3.format.table import formatted_cells, phylip_matrix, latex
```

We check we can format an arbitrary 2D list, without a header, using the `formatted_cells` function directly.

```
>>> data = [[230, 'acdef', 1.3], [6, 'cc', 1.9876]]
>>> head = ['one', 'two', 'three']
>>> header, formatted = formatted_cells(data, header=head)
>>> print(formatted)
[['230', 'acdef', '1.3000'], ['  6', '   cc', '1.9876']]
>>> print(header)
['one', '  two', ' three']
```

We directly test the latex formatting.

```
>>> print(latex(formatted, header, justify='lrl', caption='A legend',
...             label="table:test"))
\begin{longtable}[htp!]{ l r l }
\hline
\bf{one} & \bf{two} & \bf{three} \\
\hline
\hline
230 & acdef & 1.3000 \\
  6 &    cc & 1.9876 \\
\hline
\caption{A legend}
\label{table:test}
\end{longtable}
```

Manipulation of Tree Node Objects

Section author: Tony Walters

Examples of how to initialize and manipulate various tree node objects.

```
>>> from cogent3.core.tree import PhyloNode
>>> from cogent3 import LoadTree
>>> from cogent3.parse.tree import DndParser
```

The general method to initialize a tree is `LoadTree`, however, for exceptionally large trees or if one needs to specify the node objects (`TreeNode`, `PhyloNode`, or `RangeNode`), `DndParser` should be used. `LoadTree` uses `PhyloNode` objects by default.

The basic properties of the tree node objects are:

- `TreeNode` objects are general purpose in nature, and lack phylogenetic distance values.
- `PhyloNode` objects inherit the methods of the `TreeNode` class and in addition contain phylogenetic distances.
- `RangeNode` objects contain evolution simulation methods in addition to the standard features of a `PhyloNode`.

The following demonstrates the two methods for initializing a phylogenetic tree object.

```
>>> simple_tree_string="(B:0.2,(C:0.3,D:0.4)E:0.5)F;"
>>> complex_tree_string="(((363564 AB294167.1 Alkalibacterium putridalgicola:0.
↪0028006,55874 AB083411.1 Marinilactibacillus psychrotolerans:0.0022089):0.40998,
↪(15050 Y10772.1 Facklamia hominis:0.32304,(132509 AY707780.1 Aerococcus viridans:0.
↪58815,((143063 AY879307.1 Abiotrophia defectiva:0.5807,83619 AB042060.1 Bacillus␣
↪schlegelii:0.23569):0.03586,169722 AB275483.1 Fibrobacter succinogenes:0.38272):0.
↪06516):0.03492):0.14265):0.63594,(3589 M62687.1 Fibrobacter intestinalis:0.65866,
↪314063 CP001146.1 Dictyoglomus thermophilum:0.38791):0.32147,276579 EU652053.1␣
↪Thermus scotoductus:0.57336);"
>>> simple_tree=LoadTree(treestring=simple_tree_string)
>>> complex_tree=DndParser(complex_tree_string, PhyloNode)
```

Now to displaying, creating, deleting, and inserting a node in simple_tree. Note that simple_tree has three tips, one internal node 'E', and the root 'F.' For this example, we will create a node named 'A', with a distance of 0.1, delete the node 'C' through its parent, the internal node 'E', and finally we will insert 'A' where 'C' once was.

Display the original tree.

```
>>> print(simple_tree.ascii_art())
          /-B
-F--------|
          |          /-C
          \E--------|
                     \-D
```

Create a new node object.

```
>>> A_node=PhyloNode(name='A',Length=0.1)
```

Display the children of the root node, one of which is the parent of the tip we wish to alter. To add or remove a node, we need to use the parent of the target node, which in this case is the internal node 'E.'

```
>>> print(simple_tree.children)
[Tree("B;"), Tree("(C,D)E;")]
```

Remove the 'C' tip. **Note:** `remove()` and `removeNode()` return 'True' if a node is removed, 'False' if they cannot remove a node.

```
>>> simple_tree.children[1].remove('C')
True
```

Insert the new 'A' tip where 'C' was previously.

```
>>> simple_tree.children[1].insert(0,A_node)
```

Finally, display the modified tree.

```
>>> print(simple_tree.ascii_art())
          /-B
-F-------|
          |          /-A
          \E-------|
                     \-D
```

When deleting tree nodes, it is often desirable to clean up any unbranched internal nodes that may have resulted from removal of tips. For example, if we wanted to delete the node 'A' that was previously added, the resulting tree would have an unbranched internal node 'E.'

```
>>> simple_tree.children[1].remove('A')
True
>>> print(simple_tree.ascii_art())
          /-B
-F-------|
          \E------- /-D
```

With the `prune()` method, internal nodes with only a single branch are removed.

```
>>> simple_tree.prune()
>>> print(simple_tree.ascii_art())
          /-B
-F-------|
          \-D
```

An Example of Conditional Tree Node Modifications

Now to look at the more complex and realistic tree. In complex_tree, there are no internal nodes or a defined root. In order to display this tree in a more succinct manner, we can rename these tips to only contain the genus and species names. To step through the tips only, we can use the `iter_tips()` iterator, and rename each node. The `ascii_art()` function, by default, will attempt to display internal nodes; this can be suppressed by the parameter `show_internal=False`.

First, let's split the ungainly name string for each tip and only preserve the genus and species component, separated by a space.

```
>>> for n in complex_tree.iter_tips():
...     n.name=n.name.split()[2]+" "+n.name.split()[3]
```

Now we display the tree with `ascii_art()`.

```
>>> print(complex_tree.ascii_art(show_internal=False))
                              /-Alkalibacterium putridalgicola
                    /--------|
                   |          \-Marinilactibacillus psychrotolerans
          /--------|
```

```
          |          |          /-Facklamia hominis
          |          |         |
          |          \--------|          /-Aerococcus viridans
          |                   |         |
          |                    \--------|                    /-Abiotrophia defectiva
          |                             |          /--------|
---------|                               \--------|          \-Bacillus schlegelii
          |                                       |
          |                                        \-Fibrobacter succinogenes
          |
          |          /-Fibrobacter intestinalis
          |---------|
          |          \-Dictyoglomus thermophilum
          |
           \-Thermus scotoductus
```

For another example of manipulating a phylogenetic tree, let us suppose that we want to remove any species in the tree that are not closely related to *Aerococcus viridans*. To do this, we will delete any nodes that have a greater phylogenetic distance than 1.8 from *Aerococcus viridans*. The best method to remove a large number of nodes from a tree is to first create a list of nodes to delete, followed by the actual removal process. It is important that the `prune()` function be called after deletion of each node to ensure that internal nodes whose tips are deleted are removed instead of becoming tips. Alternatively, one could test for internal nodes whose children are deleted in the procedure and flag these nodes to be deleted as well.

First, generate a list of tip nodes.

```
>>> tips=complex_tree.tips()
```

Next, iterate through this list, compare the distances to *Aerococcus*, and append to the deletion list if greater than 1.8.

```
>>> tips_to_delete=[]
>>> AEROCOCCUS_INDEX=3
>>> for n in tips:
...     if tips[AEROCOCCUS_INDEX].distance(n)>1.8:
...         tips_to_delete.append(n)
```

Now for the actual deletion process. We can simply use the parent of each node in the deletion list to remove itself. Pruning is necessary to prevent internal nodes from being left as tips. **Note:** `remove()` and `remove_node()` return 'True' if a node is successfully removed, 'False' otherwise.

```
>>> for n in tips_to_delete:
...     n.parent.remove(n)
...     complex_tree.prune()
True
True
True
```

Finally, print the modified `complex_tree`.

```
>>> print(complex_tree.ascii_art(show_internal=False))
                           /-Alkalibacterium putridalgicola
                  /--------|
                 |         \-Marinilactibacillus psychrotolerans
--------- /--------|
                 |          /-Facklamia hominis
                 |         |
                  \--------|          /-Aerococcus viridans
```

```
                                |          |
                                 \---------|                     /-Abiotrophia defectiva
                                           |           /---------|
                                            \---------|           \-Bacillus schlegelii
                                                      |
                                                       \-Fibrobacter succinogenes
```

General data analysis

Estimating periodic signals

Section author: Gavin Huttley, Julien Epps, Hua Ying

We consider two different scenarios:

- estimating the periods in a signal
- estimating the power for a given period
- measuring statistical significance for the latter case

Estimating the periods in a signal

For numerical (continuous) data

We first make some sample data. A periodic signal and some noise.

```
>>> import numpy
>>> t = numpy.arange(0, 10, 0.1)
>>> n = numpy.random.randn(len(t))
>>> nse = numpy.convolve(n, numpy.exp(-t/0.05))*0.1
>>> nse = nse[:len(t)]
>>> sig = numpy.sin(2*numpy.pi*t) + nse
```

Discrete Fourier transform

We now use the discrete Fourier transform to estimate periodicity in this signal. Given we set the period to equal 10, we expect the maximum power for that index.

```
>>> from cogent3.maths.period import dft
>>> pwr, period = dft(sig)
>>> print(period)
[   2.            2.04081633    2.08333333    2.12765957    2.17391304
    2.22222222    2.27272727    2.3255814     2.38095238    2.43902439
    2.5           2.56410256    2.63157895    2.7027027     2.77777778
    2.85714286    2.94117647    3.03030303    3.125         3.22580645...
>>> print(pwr)
[ 1.06015801 +0.00000000e+00j  0.74686707 -1.93971914e-02j
  0.36784793 -2.66370366e-02j  0.04384413 +2.86970840e-02j
  1.54473269 -2.43777386e-02j  0.28522968 -2.33602932e-01j...
```

The power (`pwr`) is returned as an array of complex numbers, so we convert into real numbers using `abs`. We then zip the power and corresponding periods and sort to identify the period with maximum signal.

```
>>> pwr = abs(pwr)
>>> max_pwr, max_period = sorted(zip(pwr,period))[-1]
>>> print(max_pwr, max_period)
50.7685934719 10.0
```

Auto-correlation

We now use auto-correlation.

```
>>> from cogent3.maths.period import auto_corr
>>> pwr, period = auto_corr(sig)
>>> print(period)
[ 2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24...
>>> print(pwr)
[  1.63366075e+01  -1.47309007e+01  -3.99310414e+01  -4.94779387e+01...
```

We then zip the power and corresponding periods and sort to identify the period with maximum signal.

```
>>> max_pwr, max_period = sorted(zip(pwr,period))[-1]
>>> print(max_pwr, max_period)
46.7917300733 10
```

For symbolic data

We create a sequence as just a string

```
>>> s = 'ATCGTTGGGACCGGTTCAAGTTTTGGAACTCGCAAGGGGTGAATGGTCTTCGTCTAACGCTGG'\
...     'GGAACCCTGAATCGTTGTAACGCTGGGGTCTTTAACCGTTCTAATTTAACGCTGGGGGGTTCT'\
...     'AATTTTTAACCGCGGAATTGCGTC'
```

We then specify the motifs whose occurrences will be converted into 1, with all other motifs converted into 0. As we might want to do this in batches for many sequences we use a factory function.

```
>>> from cogent3.maths.stats.period import SeqToSymbols
>>> seq_to_symbols = SeqToSymbols(['AA', 'TT', 'AT'])
>>> symbols = seq_to_symbols(s)
>>> len(symbols) == len(s)
True
>>> symbols
array([1, 0, 0, 0, 1, 0, 0, 0, 0, 0...
```

We then estimate the integer discrete Fourier transform for the full data. To do this, we need to pass in the symbols from full conversion of the sequence. The returned values are the powers and periods.

```
>>> from cogent3.maths.period import ipdft
>>> powers, periods = ipdft(symbols)
>>> powers
array([  3.22082108e-14,   4.00000000e+00,   9.48683298e+00,
         6.74585634e+00,   3.46410162e+00,   3.20674669e+00,...
>>> periods
array([  2,   3,   4...
```

We can also compute the auto-correlation statistic, and the hybrid (which combines IPDFT and auto-correlation).

```
>>> from cogent3.maths.period import auto_corr, hybrid
>>> powers, periods = auto_corr(symbols)
>>> powers
array([ 11.,   9.,  11.,   9.,   6...
>>> periods
array([  2,   3,   4...
>>> powers, periods = hybrid(symbols)
>>> powers
array([  3.54290319e-13,   3.60000000e+01,   1.04355163e+02,
         6.07127071e+01,   2.07846097e+01,   2.88607202e+01,...
>>> periods
array([  2,   3,   4...
```

Estimating power for specified period

For numerical (continuous) data

We just use `sig` created above. The Goertzel algorithm gives the same result as the `dft`.

```
>>> from cogent3.maths.period import goertzel
>>> pwr = goertzel(sig, 10)
>>> print(pwr)
50.7685934719
```

For symbolic data

We use the symbols from the above example. For the `ipdft`, `auto_corr` and `hybrid` functions we just need to identify the array index containing the period of interest and slice the corresponding value from the returned powers. The reported periods start at `llim`, which defaults to 2, but indexes start at 0, the index for a period-5 is simply 5-`llim`.

```
>>> powers, periods = auto_corr(symbols)
>>> llim = 2
>>> period5 = 5-llim
>>> periods[period5]
5
>>> powers[period5]
9.0
```

For Fourier techniques, we can compute the power for a specific period more efficiently using Goertzel algorithm.

```
>>> from cogent3.maths.period import goertzel
>>> period = 4
>>> power = goertzel(symbols, period)
>>> ipdft_powers, periods = ipdft(symbols)
>>> ipdft_power = abs(ipdft_powers[period-llim])
>>> round(power, 6) == round(ipdft_power, 6)
True
>>> power
9.4868...
```

It's also possible to specify a period to the stand-alone functions. As per the `goertzel` function, just the power is returned.

```
>>> power = hybrid(symbols, period=period)
>>> power
104.355...
```

Measuring statistical significance of periodic signals

For numerical (continuous data)

We use the signal provided above. Because significance testing is being done using a resampling approach, we define a calculator which precomputes some values to improve compute performance. For a continuous signal, we'll use the Goertzel algorithm.

```
>>> from cogent3.maths.period import Goertzel
>>> goertzel_calc = Goertzel(len(sig), period=10)
```

Having defined this, we then just pass this calculator to the `blockwise_bootstrap` function. The other critical settings are the `block_size` which specifies the size of segments of contiguous sequence positions to use for sampling and `num_reps` which is the number of permuted replicate sequences to generate.

```
>>> from cogent3.maths.stats.period import blockwise_bootstrap
>>> obs_stat, p = blockwise_bootstrap(sig, calc=goertzel_calc, block_size=10,
...                                   num_reps=1000)
>>> print(obs_stat)
50.7685934719
>>> print(p)
0.0
```

For symbolic data

Permutation testing

The very notion of permutation testing for periods, applied to a genome, requires the compute performance be as quick as possible. This means providing as much information up front as possible. We have made the implementation flexible by not assuming how the user will convert sequences to symbols. It's also the case that numerous windows of exactly the same size are being assessed. Accordingly, we use a class to construct a fixed signal length evaluator. We do this for the hybrid metric first.

```
>>> from cogent3.maths.period import Hybrid
>>> len(s)
150
>>> hybrid_calculator = Hybrid(len(s), period=4)
```

Note: We defined the period length of interest in defining this calculator because we're interested in dinucleotide motifs.

We then construct a seq-to-symbol convertor.

```
>>> from cogent3.maths.stats.period import SeqToSymbols
>>> seq_to_symbols = SeqToSymbols(['AA', 'TT', 'AT'], length=len(s))
```

The rest is as per the analysis using `Goertzel` above.

```
>>> from cogent3.maths.stats.period import blockwise_bootstrap
>>> stat, p = blockwise_bootstrap(s, calc=hybrid_calculator,
...      block_size=10, num_reps=1000, seq_to_symbols=seq_to_symbols)
...
>>> print(stat)
104.35...
>>> p < 0.1
True
```

Data Visualisation

Drawing dendrograms and saving to PDF

Section author: Gavin Huttley

From `cogent3` import all the components we need.

```
>>> from cogent3 import LoadSeqs, LoadTree
>>> from cogent3.evolve.models import MG94HKY
>>> from cogent3.draw import dendrogram
```

Do a model, see the neutral test example for more details of this

```
>>> aln = LoadSeqs("data/long_testseqs.fasta")
>>> t = LoadTree("data/test.tree")
>>> sm = MG94HKY()
>>> nonneutral_lf = sm.make_likelihood_function(t)
>>> nonneutral_lf.set_param_rule("omega", is_independent=True)
>>> nonneutral_lf.set_alignment(aln)
>>> nonneutral_lf.optimise(show_progress=False)
```

We will draw two different dendrograms – one with branch lengths contemporaneous, the other where length is scaled.

Specify the dimensions of the canvas in pixels

```
>>> height, width = 500, 700
```

Dendrogram with branch lengths not proportional

```
>>> np = dendrogram.ContemporaneousDendrogram(nonneutral_lf.tree)
>>> np.write_pdf('tree-unscaled.pdf' , width, height, stroke_width=2.0,
...              show_params=['r'], label_template="%(r).2g",
... shade_param='r', max_value=1.0, show_internal_labels=False,
... font_size=10, scale_bar=None, use_lengths=False)
```

Dendrogram with branch lengths proportional

```
>>> p = dendrogram.SquareDendrogram(nonneutral_lf.tree)
>>> p.write_pdf('tree-scaled.pdf', width, height, stroke_width=2.0,
...             shade_param='r', max_value=1.0,
...             show_internal_labels=False, font_size=10)
```

Separating the analysis and visualisation steps

It's typically better to not have the analysis and drawing code in the same script, since drawing involves frequent iterations. This requires saving a tree for later reuse. This can be done using an annotated tree, which looks just like a tree, but has the maximum-likelihood parameter estimates attached to each tree edge. The tree must be saved in xml format to preserve the parameter estimates. The annotated tree is obtained from the likelihood function and saved to file specifying the format with the .xml suffix. This file can then be loaded using the standard `LoadTree` method in a separate script and used for drawing.

```
>>> at = nonneutral_lf.get_annotated_tree()
>>> at.write('annotated_tree.xml')
```

Drawing a dotplot

Section author: Gavin Huttley

```
>>> from cogent3 import LoadSeqs, DNA
>>> from cogent3.core import annotation
>>> from cogent3.draw import dotplot
```

Load the alignment for illustrative purposes, I'll make one sequence a different length than the other and introduce a custom sequence annotation for a miscellaneous feature. Normally, those annotations would be on the unaligned sequences.

```
>>> aln = LoadSeqs("data/test.paml", moltype=DNA, array_align=False)
>>> feature = aln.add_annotation(annotation.Feature, "misc_feature",
...                              "pprobs", [(38, 55)])
>>> seq1 = aln.get_seq('NineBande')[10:-3]
>>> seq2 = aln.get_seq('DogFaced')
```

Write out the dotplot as a pdf file in the current directory note that seq1 will be the x-axis, and seq2 the y-axis.

```
>>> dp = dotplot.Display2D(seq1,seq2)
>>> filename = 'dotplot_example.pdf'
>>> dp.write_pdf(filename)
```

Modelling Evolution

The simplest script

Section author: Gavin Huttley

This is just about the simplest possible `cogent3` script for evolutionary modelling. We use a canned nucleotide substitution model (the `HKY85` model) on just three primate species. As there is only one unrooted tree possible, the sequence names are all that's required to make the tree.

```
>>> from cogent3.evolve.models import HKY85
>>> from cogent3 import LoadSeqs, LoadTree
>>> model = HKY85()
>>> aln = LoadSeqs("data/primate_cdx2_promoter.fasta")
>>> tree = LoadTree(tip_names=aln.names)
>>> lf = model.make_likelihood_function(tree)
>>> lf.set_alignment(aln)
```

```
>>> lf.optimise(show_progress=False)
>>> print(lf)
Likelihood function statistics
log-likelihood = -2494.9537
number of free parameters = 4
======
 kappa
------
5.9589
------
==========================
   edge    parent    length
--------------------------
  human      root    0.0040
macaque      root    0.0384
  chimp      root    0.0061
--------------------------
===============
motif    mprobs
---------------
    T    0.2552
    C    0.2581
    A    0.2439
    G    0.2428
---------------
```

Performing a relative rate test

Section author: Gavin Huttley

From `cogent3` import all the components we need

```
>>> from cogent3 import LoadSeqs, LoadTree
>>> from cogent3.evolve.models import HKY85
>>> from cogent3.maths import stats
```

Get your alignment and tree.

```
>>> aln = LoadSeqs(filename="data/long_testseqs.fasta")
>>> t = LoadTree(filename="data/test.tree")
```

Create a HKY85 model.

```
>>> sm = HKY85()
```

Make the controller object and limit the display precision (to decrease the chance that small differences in estimates cause tests of the documentation to fail).

```
>>> lf = sm.make_likelihood_function(t, digits=2, space=3)
```

Set the local clock for humans & Howler Monkey. This method is just a special interface to the more general `set_param_rules` method.

```
>>> lf.set_local_clock("Human", "HowlerMon")
```

Get the likelihood function object this object performs the actual likelihood calculation.

```
>>> lf.set_alignment(aln)
```

Optimise the function capturing the return optimised lnL, and parameter value vector.

```
>>> lf.optimise(show_progress=False)
```

View the resulting maximum-likelihood parameter values.

```
>>> lf.set_name("clock")
>>> print(lf)
clock
=====
kappa
-----
 4.10
-----
============================
     edge   parent   length
----------------------------
    Human   edge.0     0.04
HowlerMon   edge.0     0.04
   edge.0   edge.1     0.04
    Mouse   edge.1     0.28
   edge.1     root     0.02
NineBande     root     0.09
 DogFaced     root     0.11
----------------------------
==============
motif   mprobs
--------------
    T     0.23
    C     0.19
    A     0.37
    G     0.21
--------------
```

We extract the log-likelihood and number of free parameters for later use.

```
>>> null_lnL = lf.get_log_likelihood()
>>> null_nfp = lf.get_num_free_params()
```

Clear the local clock constraint, freeing up the branch lengths.

```
>>> lf.set_param_rule('length', is_independent=True)
```

Run the optimiser capturing the return optimised lnL, and parameter value vector.

```
>>> lf.optimise(show_progress=False)
```

View the resulting maximum-likelihood parameter values.

```
>>> lf.set_name("non clock")
>>> print(lf)
non clock
=====
kappa
-----
 4.10
```

```
-----
============================
     edge   parent   length
----------------------------
    Human   edge.0     0.03
HowlerMon   edge.0     0.04
   edge.0   edge.1     0.04
    Mouse   edge.1     0.28
   edge.1     root     0.02
NineBande     root     0.09
 DogFaced     root     0.11
----------------------------
==============
motif   mprobs
--------------
    T     0.23
    C     0.19
    A     0.37
    G     0.21
--------------
```

These two lnL's are now used to calculate the likelihood ratio statistic it's degrees-of-freedom and the probability of observing the LR.

```
>>> LR = 2 * (lf.get_log_likelihood() - null_lnL)
>>> df = lf.get_num_free_params() - null_nfp
>>> P = stats.chisqprob(LR, df)
```

Print this and look up a χ^2 with number of edges - 1 degrees of freedom.

```
>>> print("Likelihood ratio statistic = ", LR)
Likelihood ratio statistic =  2.7...
>>> print("degrees-of-freedom = ", df)
degrees-of-freedom =  1
>>> print("probability = ", P)
probability =  0.09...
```

A test of the neutral theory

Section author: Gavin Huttley

This file contains an example for performing a likelihood ratio test of neutrality. The test compares a model where the codon model parameter omega is constrained to be the same for all edges against one where each edge has its' own omega. From `cogent3` import all the components we need.

```
>>> from cogent3 import LoadSeqs, LoadTree
>>> from cogent3.evolve.models import MG94GTR
>>> from cogent3.maths import stats
```

Get your alignment and tree.

```
>>> al = LoadSeqs("data/long_testseqs.fasta")
>>> t = LoadTree("data/test.tree")
```

We use a Goldman Yang 1994 model.

```
>>> sm = MG94GTR()
```

Make the controller object

```
>>> lf = sm.make_likelihood_function(t, digits=2, space=2)
```

Get the likelihood function object this object performs the actual likelihood calculation.

```
>>> lf.set_alignment(al)
```

By default, parameters other than branch lengths are treated as global in scope, so we don't need to do anything special here. We can influence how rigorous the optimisation will be, and switch between the global and local optimisers provided in the toolkit using arguments to the optimise method. The `global_tolerance=1.0` argument specifies conditions for an early break from simulated annealing which will be automatically followed by the Powell local optimiser. .. note:: the 'results' are of course nonsense.

```
>>> lf.optimise(global_tolerance=1.0, show_progress=False)
```

View the resulting maximum-likelihood parameter values

```
>>> print(lf)
Likelihood function statistics
log-likelihood = -8636.1801
number of free parameters = 13
==================================
 A/C   A/G   A/T   C/G   C/T  omega
----------------------------------
1.02  3.36  0.73  0.95  3.71   0.90
----------------------------------
=========================
     edge  parent  length
-------------------------
    Human  edge.0    0.09
HowlerMon  edge.0    0.12
   edge.0  edge.1    0.12
    Mouse  edge.1    0.84
   edge.1    root    0.06
NineBande    root    0.28
 DogFaced    root    0.34
-------------------------
=============
motif  mprobs
-------------
    T    0.23
    C    0.19
    A    0.37
    G    0.21
-------------
```

We'll get the lnL and number of free parameters for later use.

```
>>> null_lnL = lf.get_log_likelihood()
>>> null_nfp = lf.get_num_free_params()
```

Specify each edge has it's own omega by just modifying the existing `lf`. This means the new function will start with the above values.

```
>>> lf.set_param_rule("omega", is_independent=True)
```

Optimise the likelihood function, this time just using the local optimiser.

```
>>> lf.optimise(local=True, show_progress=False)
```

View the resulting maximum-likelihood parameter values.

```
>>> print(lf)
Likelihood function statistics
log-likelihood = -8632.1355
number of free parameters = 19
============================
 A/C   A/G   A/T   C/G   C/T
----------------------------
1.03  3.38  0.73  0.95  3.72
----------------------------
================================
     edge  parent  length  omega
--------------------------------
    Human  edge.0    0.09   0.59
HowlerMon  edge.0    0.12   0.96
   edge.0  edge.1    0.11   1.13
    Mouse  edge.1    0.83   0.92
   edge.1    root    0.06   0.39
NineBande    root    0.28   1.28
 DogFaced    root    0.34   0.84
--------------------------------
=============
motif  mprobs
-------------
    T    0.23
    C    0.19
    A    0.37
    G    0.21
-------------
```

Get out an annotated tree, it looks just like a tree, but has the maximum-likelihood parameter estimates attached to each tree edge. This object can be used for plotting, or to provide starting estimates to a related model.

```
>>> at = lf.get_annotated_tree()
```

The lnL's from the two models are now used to calculate the likelihood ratio statistic (`LR`) it's degrees-of-freedom (`df`) and the probability (`P`) of observing the LR.

```
>>> LR = 2 * (lf.get_log_likelihood() - null_lnL)
>>> df = lf.get_num_free_params() - null_nfp
>>> P = stats.chisqprob(LR, df)
```

Print this and look up a chi-sq with number of edges - 1 degrees of freedom.

```
>>> print("Likelihood ratio statistic = ", LR)
Likelihood ratio statistic =  8...
>>> print("degrees-of-freedom = ", df)
degrees-of-freedom =  6
>>> print("probability = ", P)
probability =  0.2...
```

Allowing substitution model parameters to differ between branches

Section author: Gavin Huttley

A common task concerns assessing how substitution model exchangeability parameters differ between evolutionary lineages. This is most commonly of interest for the case of testing for natural selection. Here I'll demonstrate the different ways of scoping parameters across trees for the codon model case and how these can be used for evolutionary modelling.

We start with the standard imports, plus using a canned codon substitution model and then load the sample data set.

```
>>> from cogent3 import LoadSeqs, LoadTree
>>> from cogent3.evolve.models import MG94HKY
>>> aln = LoadSeqs("data/long_testseqs.fasta")
>>> tree = LoadTree("data/test.tree")
```

We construct the substitution model and likelihood function and set the alignment.

```
>>> sm = MG94HKY()
>>> lf = sm.make_likelihood_function(tree, digits=2, space=3)
>>> lf.set_alignment(aln)
```

At this point we have a likelihood function with two exchangeability parameters from the substitution model (`kappa` the transition/transversion ratio; `omega` the nonsynonymous/synonymous ratio) plus branch lengths for all tree edges. To facilitate subsequent discussion I now display the tree

```
>>> print(tree.ascii_art())
                                /-Human
                     /edge.0--|
          /edge.1--|          \-HowlerMon
         |         |
         |          \-Mouse
-root----|
         |--NineBande
         |
          \-DogFaced
```

In order to scope a parameter on a tree (meaning specifying a subset of edges for which the parameter is to be treated differently to the remainder of the tree) requires uniquely identifying the edges. We do this using the following arguments to the likelihood function `set_param_rule` method:

- `tip_names`: the name of two tips
- `outgroup_name`: the name of a tip that is not part of the clade of interest
- `is_clade`: if `True`, all lineages descended from the tree node identified by the `tip_names` and `outgroup_name` argument are affected by the other arguments. If `False`, then the `is_stem` argument must apply.
- `is_stem`: Whether the edge leading to the node is included.

The next concepts include exactly what can be scoped and how. In the case of testing for distinctive periods of natural selection it is common to specify distinct values for `omega` for an edge. I'll first illustrate some possible uses for the arguments above by setting `omega` to be distinctive for specific edges. I will set a value for omega so that printing the likelihood function illustrates what edges have been effected, but I won't actually do any model fitting.

Specifying a clade

I'm going to cause `omega` to attain a different value for all branches aside from the primate clade and stem (`HowlerMon`, `Human`, `edge.0`).

```
>>> lf.set_param_rule('omega', tip_names=['DogFaced', 'Mouse'],
...                outgroup_name='Human', init=2.0, is_clade=True)
>>> print(lf)
Likelihood function statistics
log-likelihood = -9489.9506
number of free parameters = 10
=====
kappa
-----
 1.00
-----
==================================
     edge   parent   length   omega
----------------------------------
    Human   edge.0     0.03    1.00
HowlerMon   edge.0     0.04    1.00
   edge.0   edge.1     0.04    1.00
    Mouse   edge.1     0.28    2.00
   edge.1     root     0.02    2.00
NineBande     root     0.09    2.00
 DogFaced     root     0.11    2.00
----------------------------------
==============
motif   mprobs
--------------
    T     0.23
    C     0.19
    A     0.37
    G     0.21
--------------
```

As you can see `omega` for the primate edges I listed above have the default parameter value (1.0), while the others have what I've assigned. In fact, you could omit the `is_clade` argument as this is the default, but I think for readability of scripts it's best to be explicit.

Specifying a stem

This time I'll specify the stem leading to the primates as the edge of interest.

Note: I need to reset the `lf` so all edges have the default value again. I'll show this only for this example, but rest assured I'm doing it for all others too.

```
>>> lf.set_param_rule('omega', init=1.0)
>>> lf.set_param_rule('omega', tip_names=['Human', 'HowlerMon'],
...       outgroup_name='Mouse', init=2.0, is_stem=True, is_clade=False)
>>> print(lf)
Likelihood function statistics
log-likelihood = -9424.8896
number of free parameters = 10
=====
```

```
kappa
-----
 1.00
-----
==================================
     edge   parent   length   omega
----------------------------------
    Human   edge.0     0.03    1.00
HowlerMon   edge.0     0.04    1.00
   edge.0   edge.1     0.04    2.00
    Mouse   edge.1     0.28    1.00
   edge.1     root     0.02    1.00
NineBande     root     0.09    1.00
 DogFaced     root     0.11    1.00
----------------------------------...
```

Specifying clade and stem

I'll specify that both the primates and their stem are to be considered.

```
>>> lf.set_param_rule('omega', tip_names=['Human', 'HowlerMon'],
...       outgroup_name='Mouse', init=2.0, is_stem=True, is_clade=True)
>>> print(lf)
Likelihood function statistics
log-likelihood = -9442.4271
number of free parameters = 10
=====
kappa
-----
 1.00
-----
==================================
     edge   parent   length   omega
----------------------------------
    Human   edge.0     0.03    2.00
HowlerMon   edge.0     0.04    2.00
   edge.0   edge.1     0.04    2.00
    Mouse   edge.1     0.28    1.00
   edge.1     root     0.02    1.00
NineBande     root     0.09    1.00
 DogFaced     root     0.11    1.00
----------------------------------...
```

Alternate arguments for specifying edges

The likelihood function `set_param_rule` method also has the arguments of `edge` and `edges`. These allow specific naming of the tree edge(s) to be affected by a rule. In general, however, the `tip_names` + `outgroup_name` combo is more robust.

Applications of scoped parameters

The general use-cases for which a tree scope can be applied are:

1. constraining all edges identified by a rule to have a specific value which is constant and not modifiable

```
>>> lf.set_param_rule('omega', tip_names=['Human', 'HowlerMon'],
...       outgroup_name='Mouse', is_clade=True, is_constant=True)
```

2. all edges identified by a rule have the same but different value to the rest of the tree

```
>>> lf.set_param_rule('omega', tip_names=['Human', 'HowlerMon'],
...       outgroup_name='Mouse', is_clade=True)
```

3. allowing all edges identified by a rule to have different values of the parameter with the remaining tree edges having the same value

```
>>> lf.set_param_rule('omega', tip_names=['Human', 'HowlerMon'],
...       outgroup_name='Mouse', is_clade=True, is_independent=True)
```

4. allowing all edges to have a different value

```
>>> lf.set_param_rule('omega', is_independent=True)
```

I'll demonstrate these cases sequentially as they involve gradually increasing the degrees of freedom in the model. First we'll constrain `omega` to equal 1 on the primate edges. I'll then optimise the model.

Note: here I'm specifying a constant value for the parameter and so I **must** use the argument `value` to set it. This not to be confused with the argument `init` that is used for providing initial (starting) values for fitting.

```
>>> lf.set_param_rule('omega', tip_names=['Human', 'HowlerMon'],
...       outgroup_name='Mouse', is_clade=True, value=1.0, is_constant=True)
>>> lf.optimise(local=True, show_progress=False)
>>> print(lf)
Likelihood function statistics
log-likelihood = -8640.9290
number of free parameters = 9
=====
kappa
-----
 3.87
-----
==================================
     edge   parent   length   omega
----------------------------------
    Human   edge.0     0.09    1.00
HowlerMon   edge.0     0.12    1.00
   edge.0   edge.1     0.12    0.92
    Mouse   edge.1     0.84    0.92
   edge.1     root     0.06    0.92
NineBande     root     0.28    0.92
 DogFaced     root     0.34    0.92
----------------------------------
==============
motif   mprobs
--------------
    T     0.23
    C     0.19
    A     0.37
    G     0.21
--------------
>>> print(lf.get_log_likelihood())
```

```
-8640.9...
>>> print(lf.get_num_free_params())
9
```

I'll now free up `omega` on the primate clade, but making it a single value shared by all primate lineages.

```
>>> lf.set_param_rule('omega', tip_names=['Human', 'HowlerMon'],
...       outgroup_name='Mouse', is_clade=True, is_constant=False)
>>> lf.optimise(local=True, show_progress=False)
>>> print(lf)
Likelihood function statistics
log-likelihood = -8639.7171
number of free parameters = 10
=====
kappa
-----
 3.85
-----
==================================
     edge   parent   length   omega
----------------------------------
    Human   edge.0     0.09    0.77
HowlerMon   edge.0     0.12    0.77
   edge.0   edge.1     0.12    0.92
    Mouse   edge.1     0.84    0.92
   edge.1     root     0.06    0.92
NineBande     root     0.28    0.92
 DogFaced     root     0.34    0.92
----------------------------------
==============
motif   mprobs
--------------
    T     0.23
    C     0.19
    A     0.37
    G     0.21
--------------
>>> print(lf.get_log_likelihood())
-8639.7...
>>> print(lf.get_num_free_params())
10
```

Finally I'll allow all primate edges to have different values of `omega`.

```
>>> lf.set_param_rule('omega', tip_names=['Human', 'HowlerMon'],
...       outgroup_name='Mouse', is_clade=True, is_independent=True)
>>> lf.optimise(local=True, show_progress=False)
>>> print(lf)
Likelihood function statistics
log-likelihood = -8638.9572
number of free parameters = 11
=====
kappa
-----
 3.85
-----
==================================
     edge   parent   length   omega
```

```
-----------------------------------
    Human   edge.0     0.09    0.59
HowlerMon   edge.0     0.12    0.95
   edge.0   edge.1     0.12    0.92
    Mouse   edge.1     0.84    0.92
   edge.1     root     0.06    0.92
NineBande     root     0.28    0.92
 DogFaced     root     0.34    0.92
-----------------------------------
==============
motif   mprobs
--------------
    T     0.23
    C     0.19
    A     0.37
    G     0.21
--------------
>>> print(lf.get_log_likelihood())
-8638.9...
>>> print(lf.get_num_free_params())
11
```

We now allow `omega` to be different on all edges.

```
>>> lf.set_param_rule('omega', is_independent=True)
>>> lf.optimise(local=True, show_progress=False)
>>> print(lf)
Likelihood function statistics
log-likelihood = -8636.1383
number of free parameters = 15
=====
kappa
-----
 3.85
-----
===================================
     edge   parent   length   omega
-----------------------------------
    Human   edge.0     0.09    0.59
HowlerMon   edge.0     0.12    0.95
   edge.0   edge.1     0.12    1.13
    Mouse   edge.1     0.84    0.92
   edge.1     root     0.06    0.38
NineBande     root     0.28    1.27
 DogFaced     root     0.34    0.84
-----------------------------------
==============
motif   mprobs
--------------
    T     0.23
    C     0.19
    A     0.37
    G     0.21
--------------
>>> print(lf.get_log_likelihood())
-8636.1...
>>> print(lf.get_num_free_params())
15
```

Using codon models

Section author: Gavin Huttley

The basic paradigm for evolutionary modelling is:

1. construct the codon substitution model
2. constructing likelihood function
3. modify likelihood function (setting rules)
4. providing the alignment(s)
5. optimisation
6. get results out

Note: In the following, a result followed by '...' just means the output has been truncated for the sake of a succinct presentation.

Constructing the codon substitution model

PyCogent3 implements 4 basic rate matrices, described in a recently accepted manuscript: i) NF models, these are nucleotide frequency weighted rate matrices and were initially described by Muse and Gaut (1994); ii) a variant of (i) where position specific nucleotide frequencies are used; iii) TF models, these are tuple (codon in this case) frequency weighted rate matrices and were initially described by Goldman and Yang (1994); iv) CNF, these use the conditional nucleotide frequency and have developed by Yap, Lindsay, Easteal and Huttley. These different models can be created using provided convenience functions which will be the case here, or specified by directly calling the `Codon` substitution model class and setting the argument `mprob_model` equal to:

- NF, `mprob_model='monomer'`
- NF with position specific nucleotide frequencies, `mprob_model='monomers'`
- TF, `mprob_model=None`
- CNF, `mprob_model='conditional'`

Warning: The TF form is the currently the default, but this will be changed in the near future.

In the following I will construct GTR variants of i and iv and a HKY variant of iii.

We import these explicitly from the `cogent3.evolve.models` module.

```
>>> from cogent3.evolve.models import CNFGTR, MG94GTR, GY94
```

These are functions and calling them returns the indicated substitution model with default behaviour of recoding gap characters into N's.

```
>>> tf = GY94()
>>> nf = MG94GTR()
>>> cnf = CNFGTR()
```

In the following demonstration I will use only the CNF form (`cnf`).

For our example we load a sample alignment and tree as per usual. To reduce the computational overhead for this example we will limit the number of sampled taxa.

```
>>> from cogent3 import LoadSeqs, LoadTree, DNA
>>> aln = LoadSeqs('data/primate_brca1.fasta', moltype=DNA)
>>> tree = LoadTree('data/primate_brca1.tree')
```

Standard test of neutrality

We construct a likelihood function and constrain omega parameter (the ratio of nonsynonymous to synonymous substitutions) to equal 1. We also set some display formatting parameters.

```
>>> lf = cnf.make_likelihood_function(tree, digits=2, space=3)
>>> lf.set_param_rule('omega', is_constant=True, value=1.0)
```

We then provide an alignment and optimise the model. In the current case we just use the local optimiser (hiding progress to keep this document succinct). We then print(the results.)

Note: I'm going to specify a set of conditions that will be used for all optimiser steps. For those new to python, one can construct a dictionary with the following form `{'argument_name': argument_value}`, or alternatively `dict(argument_name=argument_value)`. I'm doing the latter. This dictionary is then passed to functions/methods by prefacing it with `**`.

```
>>> optimiser_args = dict(local=True, max_restarts=5, tolerance=1e-8,
...                       show_progress=False)
>>> lf.set_alignment(aln)
>>> lf.optimise(**optimiser_args)
>>> print(lf)
Likelihood function statistics
log-likelihood = -6767.0980
number of free parameters = 16
=======================================
 A/C    A/G    A/T    C/G    C/T   omega
---------------------------------------
1.10   4.07   0.84   1.95   4.58    1.00
---------------------------------------
============================
      edge   parent   length
----------------------------
    Galago     root     0.53
 HowlerMon     root     0.14
    Rhesus   edge.3     0.07
 Orangutan   edge.2     0.02
   Gorilla   edge.1     0.01
     Human   edge.0     0.02
Chimpanzee   edge.0     0.01
    edge.0   edge.1     0.00
    edge.1   edge.2     0.01
    edge.2   edge.3     0.04
    edge.3     root     0.02
----------------------------
===============
motif    mprobs
---------------
  AAA      0.06
  AAC      0.02
  AAG      0.03
```

```
  AAT      0.06
  ACA      0.02
  ...       ...
  TGT      0.02
  TTA      0.02
  TTC      0.01
  TTG      0.01
  TTT      0.02
---------------
```

In the above output, the first table shows the maximum likelihood estimates (MLEs) for the substitution model parameters that are 'global' in scope. For instance, the `C/T=4.58` MLE indicates that the relative rate of substitutions between C and T is nearly 5 times the background substitution rate.

The above function has been fit using the default counting procedure for estimating the motif frequencies, i.e. codon frequencies are estimated as the average of the observed codon frequencies. If you wanted to numerically optimise the motif probabilities, then modify the likelihood function creation line to

```
lf = cnf.make_likelihood_function(tree, optimise_motif_probs=True)
```

We can then free up the omega parameter, but before we do that we'll store the log-likelihood and number of free parameters for the current model form for reuse later.

```
>>> neutral_lnL = lf.get_log_likelihood()
>>> neutral_nfp = lf.get_num_free_params()
>>> lf.set_param_rule('omega', is_constant=False)
>>> lf.optimise(**optimiser_args)
>>> print(lf)
Likelihood function statistics
log-likelihood = -6762.5761
number of free parameters = 17
=======================================
 A/C    A/G    A/T    C/G    C/T   omega
---------------------------------------
1.08   3.86   0.78   1.96   4.08    0.75
---------------------------------------
============================
      edge   parent   length
----------------------------
    Galago     root     0.53
 HowlerMon     root     0.14
    Rhesus   edge.3     0.07
 Orangutan   edge.2     0.02
   Gorilla   edge.1     0.01
     Human   edge.0     0.02
Chimpanzee   edge.0     0.01
    edge.0   edge.1     0.00
    edge.1   edge.2     0.01
    edge.2   edge.3     0.03
    edge.3     root     0.02
----------------------------
===============
motif    mprobs
---------------
  AAA      0.06
  AAC      0.02
  AAG      0.03
  AAT      0.06
```

```
  ACA       0.02
  ...        ...
  TGT       0.02
  TTA       0.02
  TTC       0.01
  TTG       0.01
  TTT       0.02
---------------
>>> non_neutral_lnL = lf.get_log_likelihood()
>>> non_neutral_nfp = lf.get_num_free_params()
```

We then conduct a likelihood ratio test whether the MLE of omega significantly improves the fit over the constraint it equals 1. We import the convenience function from the `cogent3` stats module.

```
>>> from cogent3.maths.stats import chisqprob
>>> LR = 2 * (non_neutral_lnL - neutral_lnL)
>>> df = non_neutral_nfp - neutral_nfp
>>> print(chisqprob(LR, df))
0.0026...
```

Not surprisingly, this is significant. We then ask whether the Human and Chimpanzee edges have a value of omega that is significantly different from the rest of the tree.

```
>>> lf.set_param_rule('omega', tip_names=['Chimpanzee', 'Human'],
...                           outgroup_name='Galago', is_clade=True)
>>> lf.optimise(**optimiser_args)
>>> print(lf)
Likelihood function statistics
log-likelihood = -6760.1751
number of free parameters = 18
================================
 A/C    A/G    A/T    C/G    C/T
--------------------------------
1.08   3.86   0.78   1.96   4.07
--------------------------------
====================================
      edge   parent   length   omega
------------------------------------
    Galago     root     0.53    0.73
 HowlerMon     root     0.14    0.73
    Rhesus   edge.3     0.07    0.73
 Orangutan   edge.2     0.02    0.73
   Gorilla   edge.1     0.01    0.73
     Human   edge.0     0.02    2.39
Chimpanzee   edge.0     0.01    2.39
    edge.0   edge.1     0.00    0.73
    edge.1   edge.2     0.01    0.73
    edge.2   edge.3     0.03    0.73
    edge.3     root     0.02    0.73
------------------------------------
===============
motif    mprobs
---------------
  AAA       0.06
  AAC       0.02
  AAG       0.03
  AAT       0.06
  ACA       0.02
```

```
  ...       ...
  TGT      0.02
  TTA      0.02
  TTC      0.01
  TTG      0.01
  TTT      0.02
---------------
>>> chimp_human_clade_lnL = lf.get_log_likelihood()
>>> chimp_human_clade_nfp = lf.get_num_free_params()
>>> LR = 2 * (chimp_human_clade_lnL - non_neutral_lnL)
>>> df = chimp_human_clade_nfp - non_neutral_nfp
>>> print(chisqprob(LR, df))
0.028...
```

This is basically a replication of the original Huttley et al (2000) result for *BRCA1*.

Rate-heterogeneity model variants

It is also possible to specify rate-heterogeneity variants of these models. In the first instance we'll create a likelihood function where these rate-classes are global across the entire tree. Because fitting these models can be time consuming I'm going to recreate the non-neutral likelihood function from above first, fit it, and then construct the rate-heterogeneity likelihood function. By doing this I can ensure that the richer model starts with parameter values that produce a log-likelihood the same as the null model, ensuring the subsequent optimisation step improves the likelihood over the null.

```
>>> lf = cnf.make_likelihood_function(tree, digits=2, space=3)
>>> lf.set_alignment(aln)
>>> lf.optimise(**optimiser_args)
>>> non_neutral_lnL = lf.get_log_likelihood()
>>> non_neutral_nfp = lf.get_num_free_params()
```

Now, we have a null model which we know (from having fit it above) has a MLE < 1. We will construct a rate-heterogeneity model with just 2 rate-classes (neutral and adaptive) that are separated by the boundary of omega=1. These rate-classes are specified as discrete bins in PyCogent3 and the model configuration steps for a bin or bins are done using the `set_param_rule` method. To ensure the alternate model starts with a likelihood at least as good as the previous we need to make the probability of the neutral site-class bin ~= 1 (these are referenced by the `bprobs` parameter type) and assign the null model omega MLE to this class.

To get all the parameter MLEs (branch lengths, GTR terms, etc ..) into the alternate model we get an annotated tree from the null model which will have these values associated with it.

```
>>> annot_tree = lf.get_annotated_tree()
>>> omega_mle = lf.get_param_value('omega')
```

We can then construct a new likelihood function, specifying the rate-class properties.

```
>>> rate_lf = cnf.make_likelihood_function(annot_tree,
...                     bins=['neutral', 'adaptive'], digits=2, space=3)
```

We define a very small value (`epsilon`) that is used to specify the starting values.

```
>>> epsilon=1e-6
```

We now provide starting parameter values for `omega` for the two bins, setting the boundary

```
>>> rate_lf.set_param_rule('omega', bin='neutral', upper=1, init=omega_mle)
>>> rate_lf.set_param_rule('omega', bin='adaptive', lower=1+epsilon,
...         upper=100, init=1+2*epsilon)
```

and provide the starting values for the bin probabilities (`bprobs`).

```
>>> rate_lf.set_param_rule('bprobs', init=[1-epsilon, epsilon])
```

The above statement essentially assigns a probability of nearly 1 to the 'neutral' bin. We now set the alignment and fit the model.

```
>>> rate_lf.set_alignment(aln)
>>> rate_lf.optimise(**optimiser_args)
>>> rate_lnL = rate_lf.get_log_likelihood()
>>> rate_nfp = rate_lf.get_num_free_params()
>>> LR = 2 * (rate_lnL - non_neutral_lnL)
>>> df = rate_nfp - non_neutral_nfp
>>> print(rate_lf)
Likelihood function statistics
log-likelihood = -6755.4520
number of free parameters = 19
=================================
 A/C    A/G    A/T    C/G    C/T
---------------------------------
1.07   3.96   0.78   1.96   4.20
---------------------------------
=========================
     bin   bprobs   omega
-------------------------
 neutral     0.14    0.01
adaptive     0.86    1.17
-------------------------
============================
      edge   parent   length
----------------------------
    Galago     root     0.56
 HowlerMon     root     0.14
    Rhesus   edge.3     0.07
 Orangutan   edge.2     0.02
   Gorilla   edge.1     0.01
     Human   edge.0     0.02
Chimpanzee   edge.0     0.01
    edge.0   edge.1     0.00
    edge.1   edge.2     0.01
    edge.2   edge.3     0.03
    edge.3     root     0.02
----------------------------
===============
motif    mprobs
---------------
  AAA      0.06
  AAC      0.02
  AAG      0.03
  AAT      0.06
  ACA      0.02
  ...       ...
  TGT      0.02
  TTA      0.02
```

```
  TTC       0.01
  TTG       0.01
  TTT       0.02
---------------
>>> print(chisqprob(LR, df))
0.000...
```

We can get the posterior probabilities of site-classifications out of this model as

```
>>> pp = rate_lf.get_bin_probs()
```

This is a `DictArray` class which stores the probabilities as a `numpy.array`.

Mixing branch and site-heterogeneity

The following implements a modification of the approach of Zhang, Nielsen and Yang (Mol Biol Evol, 22:2472–9, 2005). For this model class, there are groups of branches for which all positions are evolving neutrally but some proportion of those neutrally evolving sites change to adaptively evolving on so-called foreground edges. For the current example, we'll define the Chimpanzee and Human branches as foreground and everything else as background. The following table defines the parameter scopes.

Site class	Proportion	Background edges	Foreground edges
0	p_0	0 < omega_0 < 1	0 < omega_0 < 1
1	p_1	omega_1=1	omega_1=1
2a	p_2	0 < omega_0 < 1	omega_2 > 1
2b	p_3	omega_1=1	omega_2 > 1

Note: Our implementation is not as parametrically succinct as that of Zhang et al, we have 1 additional bin probability.

After Zhang et al, we first define a null model that has 2 rate classes '0' and '1'. We also get all the MLEs out using `get_statistics`, just printing out the bin parameters table in the current case.

```
>>> rate_lf = cnf.make_likelihood_function(tree, bins=['0', '1'],
...                                digits=2, space=3)
>>> rate_lf.set_param_rule('omega', bin='0', upper=1.0-epsilon,
...                        init=1-epsilon)
>>> rate_lf.set_param_rule('omega', bins='1', is_constant=True, value=1.0)
>>> rate_lf.set_alignment(aln)
>>> rate_lf.optimise(**optimiser_args)
>>> tables = rate_lf.get_statistics(with_titles=True)
>>> for table in tables:
...     if 'bin' in table.title:
...         print(table)
bin params
====================
bin   bprobs   omega
--------------------
  0     0.11    0.00
  1     0.89    1.00
--------------------
```

We're also going to use the MLEs from the `rate_lf` model, since that nests within the more complex branch by rate-class model. This is unfortunately quite ugly compared with just using the annotated tree approach described above. It is currently necessary, however, due to a bug in constructing annotated trees for models with binned parameters.

```
>>> globals = [t for t in tables if 'global' in t.title][0]
>>> globals = dict(zip(globals.header, globals.tolist()[0]))
>>> bin_params = [t for t in tables if 'bin' in t.title][0]
>>> rate_class_omegas = dict(bin_params.tolist(['bin', 'omega']))
>>> rate_class_probs = dict(bin_params.tolist(['bin', 'bprobs']))
>>> lengths = [t for t in tables if 'edge' in t.title][0]
>>> lengths = dict(lengths.tolist(['edge', 'length']))
```

We now create the more complex model,

```
>>> rate_branch_lf = cnf.make_likelihood_function(tree,
...             bins=['0', '1', '2a', '2b'], digits=2, space=3)
```

and set from the nested null model the branch lengths,

```
>>> for branch, length in lengths.items():
...     rate_branch_lf.set_param_rule('length', edge=branch, init=length)
```

GTR term MLES,

```
>>> for param, mle in globals.items():
...     rate_branch_lf.set_param_rule(param, init=mle)
```

binned parameter values,

```
>>> rate_branch_lf.set_param_rule('omega', bins=['0', '2a'], upper=1.0,
...                 init=rate_class_omegas['0'])
>>> rate_branch_lf.set_param_rule('omega', bins=['1', '2b'], is_constant=True,
...                 value=1.0)
>>> rate_branch_lf.set_param_rule('omega', bins=['2a', '2b'],
...                    edges=['Chimpanzee', 'Human'], init=99,
...                    lower=1.0, upper=100.0, is_constant=False)
```

and the bin probabilities.

```
>>> rate_branch_lf.set_param_rule('bprobs',
...         init=[rate_class_probs['0']-epsilon,
...               rate_class_probs['1']-epsilon, epsilon, epsilon])
```

The result of these steps is to create a rate/branch model with initial parameter values that result in likelihood the same as the null.

```
>>> rate_branch_lf.set_alignment(aln)
```

This function can then be optimised as before. The results of one such optimisation are shown below. As you can see, the `omega` value for the '2a' and '2b' bins is at the upper bounds, indicating the model is not maximised in this case.

```
rate_branch_lf.optimise(**optimiser_args)
print(rate_branch_lf)
Likelihood function statistics
log-likelihood = -6753.4561
number of free parameters = 21
=========================
      edge   bin    omega
-------------------------
    Galago     0     0.00
    Galago     1     1.00
```

```
    Galago    2a     0.00
    Galago    2b     1.00
 HowlerMon     0     0.00
 HowlerMon     1     1.00
 HowlerMon    2a     0.00
 HowlerMon    2b     1.00
    Rhesus     0     0.00
    Rhesus     1     1.00
    Rhesus    2a     0.00
    Rhesus    2b     1.00
 Orangutan     0     0.00
 Orangutan     1     1.00
 Orangutan    2a     0.00
 Orangutan    2b     1.00
   Gorilla     0     0.00
   Gorilla     1     1.00
   Gorilla    2a     0.00
   Gorilla    2b     1.00
     Human     0     0.00
     Human     1     1.00
     Human    2a   100.00
     Human    2b   100.00
Chimpanzee     0     0.00
Chimpanzee     1     1.00
Chimpanzee    2a   100.00
Chimpanzee    2b   100.00...
```

Use an empirical protein substitution model

Section author: Gavin Huttley

This file contains an example of importing an empirically determined protein substitution matrix such as Dayhoff et al 1978 and using it to create a substitution model. The globin alignment is from the PAML distribution.

```
>>> from cogent3 import LoadSeqs, LoadTree, PROTEIN
>>> from cogent3.evolve.substitution_model import EmpiricalProteinMatrix
>>> from cogent3.parse.paml_matrix import PamlMatrixParser
```

Make a tree object. In this case from a string.

```
>>> treestring="(((rabbit,rat),human),goat-cow,marsupial);"
>>> t = LoadTree(treestring=treestring)
```

Import the alignment, explicitly setting the `moltype` to be protein

```
>>> al = LoadSeqs('data/abglobin_aa.phylip',
...                interleaved=True,
...                moltype=PROTEIN,
...                )
```

Open the file that contains the empirical matrix and parse the matrix and frequencies.

```
>>> matrix_file = open('data/dayhoff.dat')
```

The `PamlMatrixParser` will import the matrix and frequency from files designed for Yang's PAML package. This format is the lower half of the matrix in three letter amino acid name order white space delineated followed by motif frequencies in the same order.

```
>>> empirical_matrix, empirical_frequencies = PamlMatrixParser(matrix_file)
```

Create an Empirical Protein Matrix Substitution model object. This will take the unscaled empirical matrix and use it and the motif frequencies to create a scaled Q matrix.

```
>>> sm = EmpiricalProteinMatrix(empirical_matrix, empirical_frequencies)
```

Make a parameter controller, likelihood function object and optimise.

```
>>> lf = sm.make_likelihood_function(t)
>>> lf.set_alignment(al)
>>> lf.optimise(show_progress=False)
>>> print(lf.get_log_likelihood())
-1706...
>>> print(lf)
Likelihood function statistics
log-likelihood = -1706.2489
number of free parameters = 7
=============================
     edge    parent    length
-----------------------------
   rabbit    edge.0    0.0785
      rat    edge.0    0.1750
   edge.0    edge.1    0.0324
    human    edge.1    0.0545
   edge.1      root    0.0269
 goat-cow      root    0.0972
marsupial      root    0.2424
-----------------------------
===============
motif    mprobs
---------------
    A    0.0871
    C    0.0335
    D    0.0469
    E    0.0495
    F    0.0398
    G    0.0886
    H    0.0336
    I    0.0369
    K    0.0805
    L    0.0854
    M    0.0148
    N    0.0404
    P    0.0507
    Q    0.0383
    R    0.0409
    S    0.0696
    T    0.0585
    V    0.0647
    W    0.0105
    Y    0.0299
---------------
```

Likelihood analysis of multiple loci

Section author: Gavin Huttley

We want to know whether an exchangeability parameter is different between alignments. We will specify a null model, under which each alignment get's it's own motif probabilities and all alignments share branch lengths and the exchangeability parameter kappa (the transition / transversion ratio). We'll split the example alignment into two-pieces.

```
>>> from cogent3 import LoadSeqs, LoadTree, LoadTable
>>> from cogent3.evolve.models import HKY85
>>> from cogent3.recalculation.scope import EACH, ALL
>>> from cogent3.maths.stats import chisqprob
>>> aln = LoadSeqs("data/long_testseqs.fasta")
>>> half = len(aln)//2
>>> aln1 = aln[:half]
>>> aln2 = aln[half:]
```

We provide names for those alignments, then construct the tree, model instances.

```
>>> loci_names = ["1st-half", "2nd-half"]
>>> loci = [aln1, aln2]
>>> tree = LoadTree(tip_names=aln.names)
>>> mod = HKY85()
```

To make a likelihood function with multiple alignments we provide the list of loci names. We can then specify a parameter (other than length) to be the same across the loci (using the imported `ALL`) or different for each locus (using `EACH`). We conduct a LR test as before.

```
>>> lf = mod.make_likelihood_function(tree,loci=loci_names,digits=2,space=3)
>>> lf.set_param_rule("length", is_independent=False)
>>> lf.set_param_rule("kappa", loci=ALL)
>>> lf.set_alignment(loci)
>>> lf.optimise(local=True)
>>> print(lf)
Likelihood function statistics
log-likelihood = -9168.3331
number of free parameters = 2
==============
kappa   length
--------------
 3.98     0.13
--------------
=========================
   locus    motif   mprobs
-------------------------
1st-half       T     0.22
1st-half       C     0.18
1st-half       A     0.38
1st-half       G     0.21
2nd-half       T     0.24
2nd-half       C     0.19
2nd-half       A     0.35
2nd-half       G     0.22
-------------------------
>>> all_lnL = lf.get_log_likelihood()
>>> all_nfp = lf.get_num_free_params()
>>> lf.set_param_rule('kappa', loci=EACH)
```

```
>>> lf.optimise(local=True, show_progress=False)
>>> print(lf)
Likelihood function statistics
log-likelihood = -9167.5373
number of free parameters = 3
======
length
------
  0.13
------
================
   locus   kappa
----------------
1st-half    4.33
2nd-half    3.74
----------------
=========================
   locus   motif   mprobs
-------------------------
1st-half       T     0.22
1st-half       C     0.18
1st-half       A     0.38
1st-half       G     0.21
2nd-half       T     0.24
2nd-half       C     0.19
2nd-half       A     0.35
2nd-half       G     0.22
-------------------------
>>> each_lnL = lf.get_log_likelihood()
>>> each_nfp = lf.get_num_free_params()
>>> LR = 2 * (each_lnL - all_lnL)
>>> df = each_nfp - all_nfp
```

Just to pretty up the result display, I'll print(a table consisting of the test statistics created on the fly.)

```
>>> print(LoadTable(header=['LR', 'df', 'p'],
...              rows=[[LR, df, chisqprob(LR, df)]], digits=2, space=3))
================
  LR   df      p
----------------
1.59    1   0.21
----------------
```

Reusing results to speed up optimisation

Section author: Gavin Huttley

An example of how to use the maximum-likelihood parameter estimates from one model as starting values for another model. In this file we do something silly, by saving a result and then reloading it. This is silly because the analyses are run consecutively. A better approach when running consecutively is to simply use the annotated tree directly.

```
>>> from cogent3 import LoadSeqs, LoadTree
>>> from cogent3.evolve.models import MG94HKY
```

We'll create a simple model, optimise it and save it for later reuse

```
>>> aln = LoadSeqs("data/long_testseqs.fasta")
>>> t = LoadTree("data/test.tree")
>>> sm = MG94HKY()
>>> lf = sm.make_likelihood_function(t, digits=2, space=2)
>>> lf.set_alignment(aln)
>>> lf.optimise(local=True, show_progress=False)
>>> print(lf)
Likelihood function statistics
log-likelihood = -8640.2127
number of free parameters = 9
============
kappa  omega
------------
 3.85   0.90
------------

=========================
     edge  parent  length
-------------------------
    Human  edge.0    0.09
HowlerMon  edge.0    0.12
   edge.0  edge.1    0.12
    Mouse  edge.1    0.84
   edge.1    root    0.06
NineBande    root    0.28
 DogFaced    root    0.34
-------------------------

=============
motif  mprobs
-------------
    T    0.23
    C    0.19
    A    0.37
    G    0.21
-------------
```

The essential object for reuse is an annotated tree these capture the parameter estimates from the above optimisation we can either use this directly in the same run, or we can save the tree to file in `xml` format and reload the tree at a later time for use. In this example I'll illustrate the latter scenario.

```
>>> at=lf.get_annotated_tree()
>>> at.write('tree.xml')
```

We load the tree as per usual

```
>>> nt = LoadTree('tree.xml')
```

Now create a more parameter rich model, in this case by allowing the `Human` edge to have a different value of `omega`. By providing the annotated tree, the parameter estimates from the above run will be used as starting values for the new model.

```
>>> new_lf = sm.make_likelihood_function(nt, digits=2, space=2)
>>> new_lf.set_param_rule('omega', edge='Human',
...                       is_independent=True)
>>> new_lf.set_alignment(aln)
>>> new_lf.optimise(local=True, show_progress=False)
>>> print(new_lf)
Likelihood function statistics
log-likelihood = -8638.9625
```

```
number of free parameters = 10
=====
kappa
-----
 3.85
-----
================================
     edge  parent  length  omega
--------------------------------
    Human  edge.0    0.09   0.59
HowlerMon  edge.0    0.12   0.92
   edge.0  edge.1    0.12   0.92
    Mouse  edge.1    0.84   0.92
   edge.1    root    0.06   0.92
NineBande    root    0.28   0.92
 DogFaced    root    0.34   0.92
--------------------------------
=============
motif  mprobs
-------------
    T    0.23
    C    0.19
    A    0.37
    G    0.21
-------------
```

Specifying and using an unrestricted nucleotide substitution model

Section author: Gavin Huttley

Do standard `cogent3` imports.

```
>>> from cogent3 import LoadSeqs, LoadTree, DNA
>>> from cogent3.evolve.predicate import MotifChange
>>> from cogent3.evolve.substitution_model import Nucleotide
```

To specify substitution models we use the `MotifChange` class from predicates. In the case of an unrestricted nucleotide model, we specify 11 such `MotifChanges`, the last possible change being ignored (with the result it is constrained to equal 1, thus calibrating the matrix). Also note that this is a non-reversible model and thus we can't assume the nucleotide frequencies estimated from the alignments are reasonable estimates for the root frequencies. We therefore specify they are to be optimised using `optimise_motif_probs` argument.

```
>>> ACTG = list('ACTG')
>>> preds = [MotifChange(i, j, forward_only=True) for i in ACTG for j in ACTG if i !=␣
↪j]
>>> del(preds[-1])
>>> preds
[A>C, A>T, A>G, C>A, C>T, C>G, T>A, T>C, T>G, G>A, G>C]
>>> sm = Nucleotide(predicates=preds, recode_gaps=True,
...                 optimise_motif_probs=True)
>>> print(sm)

Nucleotide ( name = ''; type = 'None'; params = ['A>C', 'A>T', 'A>G',...
```

We'll illustrate this with a sample alignment and tree in `data/primate_cdx2_promoter.fasta`.

```
>>> al = LoadSeqs("data/primate_cdx2_promoter.fasta", moltype=DNA)
>>> al
3 x 1525 dna alignment: human[AGCGCCCGCGG...], macaque[AGC...
>>> tr = LoadTree(tip_names=al.names)
>>> print(tr)
(human,macaque,chimp)root;
```

We now construct the parameter controller with each predicate constant across the tree, get the likelihood function calculator and optimise the function.

```
>>> lf = sm.make_likelihood_function(tr, digits=2, space=3)
>>> lf.set_alignment(al)
>>> lf.set_name('Unrestricted model')
>>> lf.optimise(local=True, show_progress=False)
```

We just used the Powell optimiser, as this works quite well.

```
>>> print(lf)
Unrestricted model
log-likelihood = -2491.7870
number of free parameters = 17
=============================================================================
 A>C    A>G    A>T    C>A    C>G    C>T    G>A    G>C    T>A    T>C    T>G
-----------------------------------------------------------------------------
0.49   4.88   1.04   2.04   0.99   7.89   9.00   1.55   0.48   5.53   1.57
-----------------------------
=========================
   edge   parent   length
-------------------------
  human     root     0.00
macaque     root     0.04
  chimp     root     0.01
-------------------------
==============
motif   mprobs
--------------
    T     0.26
    C     0.26
    A     0.24
    G     0.24
--------------
```

This data set consists of species that are relatively close for a modest length alignment. As a result, doing something like allowing the parameters to differ between edges is not particularly well supported. If you have lots of data it makes sense to allow parameters to differ between edges, which can be specified by modifying the `lf` as follows.

```
>>> for pred in preds:
...     lf.set_param_rule(pred, is_independent=True)
```

You would then re-optimise the model as above.

Simulate an alignment

Section author: Gavin Huttley

How to simulate an alignment. For this example we just create a simple model using a four taxon tree with very different branch lengths, a Felsenstein model with very different nucleotide frequencies and a long alignment.

See the other examples for how to define other substitution models.

```
>>> import sys
>>> from cogent3 import LoadTree
>>> from cogent3.evolve import substitution_model
```

Specify the 4 taxon tree,

```
>>> t = LoadTree(treestring='(a:0.4,b:0.3,(c:0.15,d:0.2)edge.0:0.1);')
```

Define our Felsenstein 1981 substitution model.

```
>>> sm = substitution_model.Nucleotide(motif_probs = {'A': 0.5, 'C': 0.2,
... 'G': 0.2, 'T': 0.1}, model_gaps=False)
>>> lf = sm.make_likelihood_function(t)
>>> lf.set_constant_lengths()
>>> lf.set_name('F81 model')
>>> print(lf)
F81 model
=========================
  edge    parent    length
-------------------------
     a      root    0.4000
     b      root    0.3000
     c    edge.0    0.1500
     d    edge.0    0.2000
edge.0      root    0.1000
-------------------------
===============
motif    mprobs
---------------
    T    0.1000
    C    0.2000
    A    0.5000
    G    0.2000
---------------
```

We'll now create a simulated alignment of length 1000 nucleotides.

```
>>> simulated = lf.simulate_alignment(sequence_length=1000)
```

The result is a normal `Cogent` alignment object, which can be used in the same way as any other alignment object.

Performing a parametric bootstrap

Section author: Gavin Huttley

This file contains an example for estimating the probability of a Likelihood ratio statistic obtained from a relative rate test. The bootstrap classes can take advantage of parallel architectures.

From `cogent3` import all the components we need.

```
>>> from cogent3 import LoadSeqs, LoadTree
>>> from cogent3.evolve import bootstrap
>>> from cogent3.evolve.models import HKY85
>>> from cogent3.maths import stats
```

Define the null model that takes an alignment object and returns a likelihood function properly assembled for optimising the likelihood under the null hypothesis. The sample distribution is generated using this model.

We will use a HKY model.

```
>>> def create_alt_function():
...     t = LoadTree("data/test.tree")
...     sm = HKY85()
...     return sm.make_likelihood_function(t)
```

Define a function that takes an alignment object and returns an appropriately assembled function for the alternative model. Since the two models are identical bar the constraint on the branch lengths, we'll use the same code to generate the basic likelihood function as for the alt model, and then apply the constraint here

```
>>> def create_null_function():
...     lf = create_alt_function()
...     # set the local clock for humans & howler monkey
...     lf.set_local_clock("Human", "HowlerMon")
...     return lf
```

Get our observed data alignment

```
>>> aln = LoadSeqs(filename="data/long_testseqs.fasta")
```

Create a `EstimateProbability` bootstrap instance

```
>>> estimateP = bootstrap.EstimateProbability(create_null_function(),
...                                           create_alt_function(),
...                                           aln)
```

Specify how many random samples we want it to generate. Here we use a very small number of replicates only for the purpose of testing.

```
>>> estimateP.set_num_replicates(5)
```

Run it.

```
>>> estimateP.run(show_progress=False)
```

Get the estimated probability.

```
>>> p = estimateP.get_estimated_prob()
```

`p` is a floating point value, as you'd expect. Grab the estimated likelihoods (null and alternate) for the observed data.

```
>>> print('%.2f, %.2f' % estimateP.get_observed_lnL())
-8751.94, -8750.59
```

Estimate parameter values using a sampling from a dataset

Section author: Gavin Huttley

This script uses the `sample` method of the alignment class to provide an estimate for a two stage optimisation. This allows rapid optimisation of long alignments and complex models with a good chance of arriving at the global maximum for the model and data. Local optimisation of the full alignment may end up in local maximum and for this reason results from this strategy my be inaccurate.

From `cogent3` import all the components we need.

```
>>> from cogent3 import LoadSeqs, LoadTree
>>> from cogent3.evolve import  substitution_model
```

Load your alignment, note that if your file ends with a suffix that is the same as it's format (assuming it's a supported format) then you can just give the filename. Otherwise you can specify the format using the format argument.

```
>>> aln = LoadSeqs("data/long_testseqs.fasta")
```

Get your tree

```
>>> t = LoadTree("data/test.tree")
```

Get the substitution model (defaults to Felsensteins 1981 model)

```
>>> sm = substitution_model.Nucleotide()
```

Make a likelihood function from a sample of the alignment the `sample` method selects the chosen number of bases at random.

```
>>> lf = sm.make_likelihood_function(t)
>>> lf.set_motif_probs_from_data(aln)
>>> lf.set_alignment(aln.sample(20))
```

Optimise with the local optimiser

```
>>> lf.optimise(local=True, show_progress=False)
```

Next use the whole alignment

```
>>> lf.set_alignment(aln)
```

and the faster Powell optimiser that will only find the best result near the provided starting point

```
>>> lf.optimise(local=True, show_progress=False)
```

```
>>> print(lf)
Likelihood function statistics
log-likelihood = -9022.5387
number of free parameters = 7
=============================
     edge    parent    length
-----------------------------
    Human    edge.0    0.0309
HowlerMon    edge.0    0.0412
   edge.0    edge.1    0.0359
    Mouse    edge.1    0.2666
   edge.1      root    0.0226
NineBande      root    0.0895
 DogFaced      root    0.1095
-----------------------------
===============
motif    mprobs
---------------
    T    0.2317
    C    0.1878
    A    0.3681
```

```
    G    0.2125
---------------
```

Perform a coevolutionary analysis on biological sequence alignments

Section author: Greg Caporaso

This document describes how to perform a coevolutionary analysis on a `ArrayAlignment` object. Coevolutionary analyses identify correlated substitution patterns between `ArrayAlignment` positions (columns). Several coevolution detection methods are currently provided via the PyCogent3 coevolution module. `ArrayAlignment` objects must always be used as input to these functions.

Before using an alignment in a coevolutionary analysis, you should be confident in the alignment. Poorly aligned sequences can yield very misleading results. There can be no ambiguous residue/base codes (e.g., B/Z/X in protein alignments) – while some of the algorithms could tolerate them (e.g. Mutual Information), others which rely on information such as background residue frequencies (e.g. Statistical Coupling Analysis) cannot handle them. Some recoded amino acid alphabets will also not handle ambiguous residues. The best strategy is just to exclude ambiguous codes all together. To test for invalid characters before starting an analysis you can do the following:

```
>>> from cogent3 import LoadSeqs, PROTEIN, DNA, RNA
>>> from cogent3.core.alignment import ArrayAlignment
>>> from cogent3.evolve.coevolution import validate_alignment
>>> aln = LoadSeqs(data={'1':'GAA','2':'CTA', '3':'CTC','4':'-TC'},
...                moltype=PROTEIN, array_align=True)
>>> validate_alignment(aln)
```

To run a coevolutionary analysis, first create a `ArrayAlignment`:

```
>>> from cogent3 import LoadSeqs, PROTEIN, DNA, RNA
>>> from cogent3.core.alignment import ArrayAlignment
>>> aln = LoadSeqs(data={'1':'AAA','2':'CTA', '3':'CTC','4':'-TC'},
...                moltype=PROTEIN, array_align=True)
```

Perform a coevolutionary analysis on a pair of positions in the alignment using mutual information (`mi`):

```
>>> from cogent3.evolve.coevolution import coevolve_pair_functions, coevolve_pair
>>> coevolve_pair(coevolve_pair_functions['mi'],aln,pos1=1,pos2=2)
0.31127...
```

Perform a coevolutionary analysis on a pair of positions in the alignment using statistical coupling analysis (`sca`):

```
>>> from cogent3.evolve.coevolution import coevolve_pair_functions, coevolve_pair
>>> coevolve_pair(coevolve_pair_functions['sca'],aln,pos1=1,pos2=2,cutoff=0.5)
0.98053...
```

Perform a coevolutionary analysis on one position and all other positions in the alignment using mutual information (`mi`):

```
>>> from cogent3.evolve.coevolution import coevolve_position_functions, coevolve_
↪position
>>> coevolve_position(coevolve_position_functions['mi'],aln,position=1)
array([        nan,  0.81127812,  0.31127812])
```

Perform a coevolutionary analysis on all pairs of positions in the alignment using mutual information (`mi`):

```
>>> from cogent3.evolve.coevolution import coevolve_alignment_functions, coevolve_
↪alignment
>>> coevolve_alignment(coevolve_alignment_functions['mi'],aln)
array([[        nan,         nan,         nan],
       [        nan,  0.81127812,  0.31127812],
       [        nan,  0.31127812,  1.        ]])
```

View the available algorithms for computing coevolution values:

```
>>> print(coevolve_pair_functions.keys())
dict_keys(['an', 'rmi', 'sca', 'mi', 'nmi'])
```

Perform an intermolecular coevolutionary analysis using mutual information (`mi`). Note that there are strict requirements on the sequence identifiers for intermolecular analyses, and some important considerations involved in preparing alignments for these analyses. See the coevolve_alignments docstring (i.e., `help(coevolve_alignments)` from the python interpreter) for information. Briefly, sequence identifiers are split on + symbols. The ids before the + must match perfectly between the two alignments as these are used to match the sequences between alignments. In the following example, these are common species names: human, chicken, echidna, and pig. The text after the + can be anything, and should probably be the original database identifiers of the sequences.

```
>>> from cogent3.evolve.coevolution import coevolve_alignment_functions,\
...   coevolve_alignments
>>> aln1 = LoadSeqs(data={'human+protein1':'AAA','pig+protein1':'CTA',
...  'chicken+protein1':'CTC','echidna+weird_db_identifier':'-TC'},
...   moltype=PROTEIN, array_align=True)
>>> aln2 = LoadSeqs(data={'pig+protein2':'AAAY','chicken+protein2':'CTAY',
...  'echidna+protein2':'CTCF','human+protein2':'-TCF'},
...   moltype=PROTEIN, array_align=True)
>>> coevolve_alignments(coevolve_alignment_functions['mi'],aln1,aln2)
array([[        nan,         nan,         nan],
       [        nan,  0.12255625,  0.31127812],
       [        nan,  0.31127812,  0.        ],
       [        nan,  0.31127812,  0.        ]])
```

Analysis of rate heterogeneity

Section author: Gavin Huttley

A simple example for analyses involving rate heterogeneity among sites. In this case we will simulate an alignment with two rate categories and then try to recover the rates from the alignment.

```
>>> from cogent3.evolve.substitution_model import Nucleotide
>>> from cogent3 import LoadTree
```

Make an alignment with equal split between rates 0.6 and 0.2, and then concatenate them to create a new alignment.

```
>>> model = Nucleotide(equal_motif_probs=True)
>>> tree = LoadTree("data/test.tree")
>>> lf = model.make_likelihood_function(tree)
>>> lf.set_param_rule('length', value=0.6, is_constant=True)
>>> aln1 = lf.simulate_alignment(sequence_length=10000)
>>> lf.set_param_rule('length', value=0.2, is_constant=True)
>>> aln2 = lf.simulate_alignment(sequence_length=10000)
>>> aln3 = aln1 + aln2
```

Start from scratch, optimising only rates and the rate probability ratio.

```
>>> model = Nucleotide(equal_motif_probs=True, ordered_param="rate",
...                    distribution="free")
>>> lf = model.make_likelihood_function(tree, bins=2, digits=2, space=3)
>>> lf.set_alignment(aln3)
>>> lf.optimise(local=True, max_restarts=2, show_progress=False)
```

We want to know the bin probabilities and the posterior probabilities.

```
>>> bprobs = [t for t in lf.get_statistics() if 'bin' in t.title][0]
```

Print the `bprobs` sorted by `'rate'` will generate a table like

```
bin params
====================
 bin    bprobs   rate
--------------------
bin0      0.49   0.49
bin1      0.51   1.48
--------------------
```

We'll now use a gamma distribution on the sample alignment, specifying the number of bins as 4. We specify that the bins have equal density using the `lf.set_param_rule('bprobs', is_constant=True)` command.

```
>>> model = Nucleotide(equal_motif_probs=True, ordered_param="rate",
...                    distribution="gamma")
>>> lf = model.make_likelihood_function(tree, bins=4)
>>> lf.set_param_rule('bprobs', is_constant=True)
>>> lf.set_alignment(aln3)
>>> lf.optimise(local=True, max_restarts=2, show_progress=False)
```

Evaluate process heterogeneity using a Hidden Markov Model

Section author: Gavin Huttley

The existence of rate heterogeneity in the evolution of biological sequences is well known. Typically such an evolutionary property is evaluated using so-called site-heterogeneity models. These models postulate the existence of discrete classes of sites, where sites within a class evolve according to a specific rate that is distinct from the rates of the other classes. These models retain the assumption that alignment columns evolve independently. One can naturally ask the question of whether rate classes occur randomly throughout the sequence or whether they are in fact auto-correlated - meaning sites of a class tend to cluster together. Because we do not have, *a priori*, a basis for classifying the sites the models are specified such that each column can belong to any of the designated site classes and the likelihood is computed across all possible classifications. Post numerical optimisation we can calculate the posterior probability a site column belongs to a specific site class. In `cogent3`, site classes are referred to as `bins` and so we refer to bin probabilities etc ...

To illustrate how to evaluate these hypotheses formally we specify 3 nested hypotheses: (i) Ho: no rate heterogeneity; (ii) Ha(1): two classes of sites - fast and slow, but independent sites; (iii) Ha(2): fast and slowly evolving sites are auto-correlated (meaning a sites class is correlated with that of its' immediate neighbours).

It is also possible to apply these models to different types of changes and we illustrate this with a single parameterisation at the end.

First import standard components necessary for all of the following calculations. As the likelihood ratio tests (LRT) involve nested hypotheses we will employ the chi-square approximation for assessing statistical significance.

```
>>> from cogent3.evolve.substitution_model import Nucleotide, predicate
>>> from cogent3 import LoadSeqs, LoadTree
>>> from cogent3.maths.stats import chisqprob
```

Load the alignment and tree.

```
>>> aln = LoadSeqs("data/long_testseqs.fasta")
>>> tree = LoadTree("data/test.tree")
```

Model Ho: no rate heterogeneity

We define a HKY model of nucleotide substitution, which has a transition parameter. This is defined using the `MotifChange` class, by specifying a transition as **not** a transversion (`~MotifChange('R','Y')`).

```
>>> MotifChange = predicate.MotifChange
>>> treat_gap = dict(recode_gaps=True, model_gaps=False)
>>> kappa = (~MotifChange('R', 'Y')).aliased('kappa')
>>> model = Nucleotide(predicates=[kappa], **treat_gap)
```

We specify a null model with no bins, and optimise it.

```
>>> lf_one = model.make_likelihood_function(tree, digits=2, space=3)
>>> lf_one.set_alignment(aln)
>>> lf_one.optimise(show_progress=False)
>>> lnL_one = lf_one.get_log_likelihood()
>>> df_one = lf_one.get_num_free_params()
>>> print(lf_one)
Likelihood function statistics
log-likelihood = -8750.5889
number of free parameters = 8
=====
kappa
-----
 4.10
-----
============================
     edge   parent   length
----------------------------
    Human   edge.0     0.03
HowlerMon   edge.0     0.04
   edge.0   edge.1     0.04
    Mouse   edge.1     0.28
   edge.1     root     0.02...
```

Model Ha(1): two classes of gamma distributed but independent sites

Our next hypothesis is that there are two rate classes, or bins, with rates gamma distributed. We will restrict the bin probabilities to be equal.

```
>>> bin_submod = Nucleotide(predicates=[kappa], ordered_param='rate',
...                         distribution='gamma', **treat_gap)
>>> lf_bins = bin_submod.make_likelihood_function(tree, bins=2,
...                                sites_independent=True, digits=2, space=3)
>>> lf_bins.set_param_rule('bprobs', is_constant=True)
>>> lf_bins.set_alignment(aln)
```

```
>>> lf_bins.optimise(local=True, show_progress=False)
>>> lnL_bins = lf_bins.get_log_likelihood()
>>> df_bins = lf_bins.get_num_free_params()
>>> assert df_bins == 9
>>> print(lf_bins)
Likelihood function statistics
log-likelihood = -8739.0900
number of free parameters = 9
==================
kappa   rate_shape
------------------
 4.38         1.26
------------------
====================
 bin   bprobs   rate
--------------------
bin0     0.50   0.41
bin1     0.50   1.59
--------------------
============================
     edge   parent   length
----------------------------
    Human   edge.0     0.03
HowlerMon   edge.0     0.04
   edge.0   edge.1     0.04
    Mouse   edge.1     0.31...
```

Model Ha(2): fast and slowly evolving sites are auto-correlated

We then specify a model with switches for changing between site-classes, the HMM part. The setup is almost identical to that for above with the sole difference being setting the `sites_independent=False`.

```
>>> lf_patches = bin_submod.make_likelihood_function(tree, bins=2,
...                         sites_independent=False, digits=2, space=3)
>>> lf_patches.set_param_rule('bprobs', is_constant=True)
>>> lf_patches.set_alignment(aln)
>>> lf_patches.optimise(local=True, show_progress=False)
>>> lnL_patches = lf_patches.get_log_likelihood()
>>> df_patches = lf_patches.get_num_free_params()
>>> print(lf_patches)
Likelihood function statistics
log-likelihood = -8728.1367
number of free parameters = 10
================================
bin_switch   kappa   rate_shape
--------------------------------
      0.56    4.42         1.16
--------------------------------
====================
 bin   bprobs   rate
--------------------
bin0     0.50   0.39
bin1     0.50   1.61
--------------------
============================
     edge   parent   length
```

```
--------------------------
    Human    edge.0     0.03
HowlerMon    edge.0     0.04
   edge.0    edge.1     0.04
    Mouse    edge.1     0.31
   edge.1      root     0.02
NineBande      root     0.10
 DogFaced      root     0.12
--------------------------...
```

We use the following short function to compute the LR test statistic.

```
>>> LR = lambda alt, null: 2 * (alt - null)
```

We conduct the test between the sequentially nested models.

```
>>> lr = LR(lnL_bins, lnL_one)
>>> print(lr)
22...
>>> print("%.4f" % chisqprob(lr, df_patches-df_bins))
0.0000
```

The stationary bin probabilities are labelled as `bprobs` and can be obtained as follows.

```
>>> bprobs = lf_patches.get_param_value('bprobs')
>>> print("%.1f : %.1f" % tuple(bprobs))
0.5 : 0.5
```

Of greater interest here (given the model was set up so the bin probabilities were equal, i.e. `is_constant=True`) are the posterior probabilities as those allow classification of sites. The result is a `DictArray` class instance, which behaves like a dictionary.

```
>>> pp = lf_patches.get_bin_probs()
```

If we want to know the posterior probability the 21st position belongs to `bin0`, we can determine it as:

```
>>> print(pp['bin0'][20])
0.8...
```

A model with patches of `kappa`

In this example we model sequence evolution where there are 2 classes of sites distinguished by their `kappa` parameters. We need to know what value of `kappa` to specify the delineation of the bin boundaries. We can determine this from the null model (`lf_one`). For this use case, we also need to use a `numpy.array`, so we'll import that.

```
>>> from numpy import array
>>> single_kappa = lf_one.get_param_value('kappa')
```

We then construct the substitution model in a different way to that when evaluating generic rate heterogeneity (above).

```
>>> kappa_bin_submod = Nucleotide(predicates=[kappa], **treat_gap)
>>> lf_kappa = kappa_bin_submod.make_likelihood_function(tree,
...       bins = ['slow', 'fast'], sites_independent=False, digits=1,
...       space=3)
```

To improve the likelihood fitting it is desirable to set starting values in the model that result in it's initial likelihood being that of the null model (or as close as possible). To do this, we're going to define an arbitrarily small value (`epsilon`) which we use to provide the starting value to the two bins as slightly smaller/greater than `single_kappa` for the slow/fast bins respectively. At the same time we set the upper/lower bin boundaries.

```
>>> epsilon = 1e-6
>>> lf_kappa.set_param_rule(kappa, init=single_kappa-epsilon,
...                         upper=single_kappa, bin='slow')
>>> lf_kappa.set_param_rule(kappa, init=single_kappa+epsilon,
...                         lower=single_kappa, bin='fast')
```

We then illustrate how to adjust the bin probabilities, here doing it so that one of them is nearly 1, the other nearly 0. This ensures the likelihood will be near identical to that of `lf_one` and as a result the optimisation step will actually improve fit over the simpler model.

```
>>> lf_kappa.set_param_rule('bprobs',
...             init=array([1.0-epsilon, 0.0+epsilon]))
>>> lf_kappa.set_alignment(aln)
>>> lf_kappa.optimise(local=True, show_progress=False)
>>> print(lf_kappa)
Likelihood function statistics
log-likelihood = -8749.3117
number of free parameters = 11
==========
bin_switch
----------
       0.6
----------

=====================
 bin   bprobs   kappa
---------------------
slow      0.8     3.0
fast      0.2    23.3
---------------------

===========================
     edge   parent   length
---------------------------
    Human   edge.0      0.0
HowlerMon   edge.0      0.0
   edge.0   edge.1      0.0
    Mouse   edge.1      0.3
   edge.1     root      0.0
NineBande     root      0.1
 DogFaced     root      0.1
---------------------------

==============
motif   mprobs
--------------
    T      0.2
    C      0.2
    A      0.4
    G      0.2
--------------
>>> print(lf_kappa.get_log_likelihood())
-8749.3...
```

Phylogenetic Reconstruction

Calculate pairwise distances between sequences

Section author: Gavin Huttley

An example of how to calculate the pairwise distances for a set of sequences.

```
>>> from cogent3 import LoadSeqs
>>> from cogent3.phylo import distance
```

Import a substitution model (or create your own)

```
>>> from cogent3.evolve.models import HKY85
```

Load my alignment

```
>>> al = LoadSeqs("data/long_testseqs.fasta")
```

Create a pairwise distances object with your alignment and substitution model

```
>>> d = distance.EstimateDistances(al, submodel=HKY85())
```

Printing `d` before execution shows its status.

```
>>> print(d)
=============================================================================
Seq1 \ Seq2        Human     HowlerMon        Mouse     NineBande     DogFaced
-----------------------------------------------------------------------------
      Human            *     Not Done     Not Done      Not Done     Not Done
  HowlerMon     Not Done            *     Not Done      Not Done     Not Done
      Mouse     Not Done     Not Done            *      Not Done     Not Done
  NineBande     Not Done     Not Done     Not Done             *     Not Done
   DogFaced     Not Done     Not Done     Not Done      Not Done            *
-----------------------------------------------------------------------------
```

Which in this case is to simply indicate nothing has been done.

```
>>> d.run(show_progress=False)
>>> print(d)
=====================================================================
Seq1 \ Seq2      Human     HowlerMon      Mouse     NineBande     DogFaced
---------------------------------------------------------------------
      Human          *        0.0730     0.3363        0.1804       0.1972
  HowlerMon     0.0730             *     0.3487        0.1865       0.2078
      Mouse     0.3363        0.3487          *        0.3813       0.4022
  NineBande     0.1804        0.1865     0.3813             *       0.2019
   DogFaced     0.1972        0.2078     0.4022        0.2019            *
---------------------------------------------------------------------
```

Note that pairwise distances can be distributed for computation across multiple CPU's. In this case, when statistics (like distances) are requested only the master CPU returns data.

We'll write a phylip formatted distance matrix.

```
>>> d.write('dists_for_phylo.phylip', format="phylip")
```

We'll also save the distances to file in Python's pickle format.

```
>>> import pickle
>>> f = open('dists_for_phylo.pickle', "wb")
>>> pickle.dump(d.get_pairwise_distances(), f)
>>> f.close()
```

Make a neighbor joining tree

Section author: Gavin Huttley

An example of how to calculate the pairwise distances for a set of sequences.

```
>>> from cogent3 import LoadSeqs
>>> from cogent3.phylo import distance, nj
```

Import a substitution model (or create your own)

```
>>> from cogent3.evolve.models import HKY85
```

Load the alignment.

```
>>> al = LoadSeqs("data/long_testseqs.fasta")
```

Create a pairwise distances object calculator for the alignment, providing a substitution model instance.

```
>>> d = distance.EstimateDistances(al, submodel=HKY85())
>>> d.run(show_progress=False)
```

Now use this matrix to build a neighbour joining tree.

```
>>> mytree = nj.nj(d.get_pairwise_distances())
```

We can visualise this tree by `print mytree.ascii_art()`, which generates the equivalent of:

```
                    /-Human
          /edge.0--|
         |          \-HowlerMon
         |
-root----|          /-NineBande
         |-edge.1--|
         |          \-DogFaced
         |
          \-Mouse
```

We can save this tree to file.

```
>>> mytree.write('test_nj.tree')
```

Make a UPGMA cluster

Section author: Catherine Lozupone

An example of how to calculate the pairwise distances for a set of sequences.

NOTE: UPGMA should not be used for phylogenetic reconstruction.

```
>>> from cogent3 import LoadSeqs
>>> from cogent3.phylo import distance
>>> from cogent3.cluster.UPGMA import upgma
```

Import a substitution model (or create your own)

```
>>> from cogent3.evolve.models import HKY85
```

Load the alignment.

```
>>> al = LoadSeqs("data/test.paml")
```

Create a pairwise distances object calculator for the alignment, providing a substitution model instance.

```
>>> d = distance.EstimateDistances(al, submodel=HKY85())
>>> d.run(show_progress=False)
```

Now use this matrix to build a UPGMA cluster.

```
>>> mycluster = upgma(d.get_pairwise_distances())
>>> print(mycluster.ascii_art())
                            /-NineBande
                  /edge.1--|
                 |          |          /-HowlerMon
          /edge.0--|          \edge.2--|
         |         |                    \-Human
-root----|         |
         |          \-DogFaced
         |
          \-Mouse
```

We demonstrate saving this UPGMA cluster to a file.

```
>>> mycluster.write('test_upgma.tree')
```

Phylogenetic reconstruction by least squares

Section author: Gavin Huttley

We will load some pre-computed pairwise distance data. To see how that data was computed see the *Calculate pairwise distances between sequences* example. That data is saved in a format called `pickle` which is native to python. As per usual, we import the basic components we need.

```
>>> import pickle
>>> from cogent3.phylo import distance, least_squares
```

Now load the distance data.

```
>>> f = open('dists_for_phylo.pickle', 'rb')
>>> dists = pickle.load(f)
>>> f.close()
```

If there are extremely small distances, they can cause an error in the least squares calculation. Since such estimates are between extremely closely related sequences we could simply drop all distances for one of the sequences. We won't do that here, we'll leave that as exercise.

We make the ls calculator.

```
>>> ls = least_squares.WLS(dists)
```

We will search tree space for the collection of best trees using the advanced stepwise addition algorithm (hereafter *asaa*).

Look for the single best tree

In this use case we are after just 1 tree. We specify up to what taxa size all possible trees for the sample will be computed. Here we are specifying `a=5`. This means 5 sequences will be picked randomly and all possible trees relating them will be evaluated. `k=1` means only the best tree will be kept at the end of each such round of evaluation. For every remaining sequence it is grafted onto every possible branch of this tree. The best `k` results are then taken to the next round, when another sequence is randomly selected for addition. This proceeds until all sequences have been added. The result with following arguments is a single wls score and a single `Tree` which can be saved etc ..

```
>>> score, tree = ls.trex(a=5, k=1)
>>> assert score < 1e-4
```

We won't display this tree, because we are doing more below.

A more rigourous tree space search

We change the asaa settings, so we keep more trees and then look at the distribution of the statistics for the last collection of trees. We could also change `a` to be larger, but in the current case we just adjust `k`. We also set the argument `return_all=True`, the effect of which is to return the complete set of saved trees. These, and their support statistic, can then be inspected.

```
>>> trees = ls.trex(a=5, k=5, return_all=True)
```

Remember the sum-of-squares statistic will be smaller for 'good' trees. The order of the trees returned is from good to bad. The number of returned `trees` is the same as the number requested to be retained at each step.

```
>>> print(len(trees))
5
```

Lets inspect the resulting statistics. First, the object `trees` is a list of `(wls, Tree)` tuples. We will therefore loop over the list to generate a separate list of just the wls statistics. The following syntax is called a list comprehension - basically just a very succinct `for` loop.

```
>>> wls_stats = [tree[0] for tree in trees]
```

The `wls_stats` is a list which, if printed, looks like

```
[1.3308768548934439e-05, 0.0015588630350439783, ...
```

From this you'll see that the first 5 results are very similar to each other and would probably reasonably be considered equivalently supported topologies. I'll just print(the first two of the these trees after balancing them (in order to make their representations as equal as possible).)

```
>>> t1 = trees[0][1].balanced()
>>> t2 = trees[1][1].balanced()
>>> print(t1.ascii_art())
                    /-Human
          /edge.0--|
         |          \-HowlerMon
```

```
          |
-root-----|--Mouse
          |
          |           /-NineBande
           \edge.1--|
                      \-DogFaced
>>> print(t2.ascii_art())
           /-DogFaced
          |
          |           /-Human
-root-----|-edge.0--|
          |           \-HowlerMon
          |
          |           /-NineBande
           \edge.1--|
                      \-Mouse
```

You can see the difference involves the Jackrabbit, TreeShrew, Gorilla, Rat clade.

Assessing the fit for a pre-specified tree topology

In some instances we may have a tree from the literature or elsewhere whose fit to the data we seek to evaluate. In this case I'm going load a tree as follows.

```
>>> from cogent3 import LoadTree
>>> query_tree = LoadTree(
... treestring="((Human:.2,DogFaced:.2):.3,(NineBande:.1, Mouse:.5):.2,HowlerMon:.1)")
```

We now just use the `ls` object created above. The following evaluates the query using it's associated branch lengths, returning only the wls statistic.

```
>>> ls.evaluate_tree(query_tree)
2.8...
```

We can also evaluate just the tree's topology, returning both the wls statistic and the tree with best fit branch lengths.

```
>>> wls, t = ls.evaluate_topology(query_tree)
>>> assert "%.4f" % wls == '0.0084'
```

Using maximum likelihood for measuring tree fit

This is a much slower algorithm and the interface largely mirrors that for the above. The difference is you import `maximum_likelihood` instead of `least_squares`, and use the `ML` instead of `WLS` classes. The `ML` class requires a substitution model (like a HKY85 for DNA or JTT92 for protein), and an alignment. It also optionally takes a distance matrix, such as that used here, computed for the same sequences. These distances are then used to obtain estimates of branch lengths by the WLS method for each evaluated tree topology which are then used as starting values for the likelihood optimisation.

Making a phylogenetic tree from a protein sequence alignment

Section author: Gavin Huttley

In this example we pull together the distance calculation and tree building with the additional twist of using an empirical protein substitution matrix. We will therefore be computing the tree from a protein sequence alignment. We will first do the standard `cogent3` import for `LoadSeqs`.

```
>>> from cogent3 import LoadSeqs, PROTEIN
```

We will use an empirical protein substitution matrix.

```
>>> from cogent3.evolve.models import JTT92
```

The next components we need are for computing the matrix of pairwise sequence distances and then for estimating a neighbour joining tree from those distances.

```
>>> from cogent3.phylo import nj, distance
```

Now load our sequence alignment, explicitly setting the alphabet to be protein.

```
>>> aln = LoadSeqs('data/abglobin_aa.phylip', interleaved=True,
...                moltype=PROTEIN)
```

Create an Empirical Protein Matrix Substitution model object. This will take the unscaled empirical matrix and use it and the motif frequencies to create a scaled Q matrix.

```
>>> sm = JTT92()
```

We now use this and the alignment to construct a distance calculator.

```
>>> d = distance.EstimateDistances(aln, submodel=sm)
>>> d.run(show_progress=False)
```

The resulting distances are passed to the nj function.

```
>>> mytree = nj.nj(d.get_pairwise_distances())
```

The shape of the resulting tree can be readily view by printing `mytree.ascii_art()`. The result will be equivalent to.

```
          /-human
         |
         |          /-rabbit
-root----|-edge.1--|
         |          \-rat
         |
         |          /-goat-cow
          \edge.0--|
                    \-marsupial
```

This tree can be saved to file, the `with_distances` argument specifies that branch lengths are to be included in the newick formatted output.

```
>>> mytree.write('test_nj.tree', with_distances=True)
```

CHAPTER 6

PyCogent3 Cookbook

Contents:

Introduction

What we hope is clear from the Cookbook is that PyCogent3 has extensive capabilities that cover for many areas of genomic biology.

If you have particular need of an example falling in any of the sections, please post a ticket. Better yet, if you write one, please consider submitting it as a pull request to the PyCogent3 mercurial repository and we'll include it in PyCogent3!

Manipulating biological data

Sequences

Loading nucleotide, protein sequences

`LoadSeqs` from a file

As an alignment

The function `LoadSeqs()` creates either a sequence collection or an alignment depending on the keyword argument `aligned` (the default is `True`).

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs('data/long_testseqs.fasta', moltype=DNA)
>>> type(aln)
<class 'cogent3.core.alignment.ArrayAlignment'>
```

This example and some of the following use the `long_testseqs.fasta` file.

As a sequence collection (unaligned)

Setting the `LoadSeqs()` function keyword argument `aligned=False` returns a sequence collection.

```
>>> from cogent3 import LoadSeqs, DNA
>>> seqs = LoadSeqs('data/long_testseqs.fasta', moltype=DNA, aligned=False)
>>> print(type(seqs))
<class 'cogent3.core.alignment.SequenceCollection'>
```

Note: An alignment can be sliced, but a `SequenceCollection` can not.

Specifying the file format

`LoadSeqs()` uses the filename suffix to infer the file format. This can be overridden using the `format` argument.

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs('data/long_testseqs.fasta', moltype=DNA,
...                  format='fasta')
...
>>> aln
5 x 2532 dna alignment: Human[TGTGGCACAAA...
```

`LoadSeqs` from a series of strings

```
>>> from cogent3 import LoadSeqs
>>> seqs = ['>seq1','AATCG-A','>seq2','AATCGGA']
>>> seqs_loaded = LoadSeqs(data=seqs)
>>> print(seqs_loaded)
>seq1
AATCG-A
>seq2
AATCGGA
```

`LoadSeqs` from a dict of strings

```
>>> from cogent3 import LoadSeqs
>>> seqs = {'seq1': 'AATCG-A','seq2': 'AATCGGA'}
>>> seqs_loaded = LoadSeqs(data=seqs)
```

Specifying the sequence molecular type

Simple case of loading a `list` of aligned amino acid sequences in FASTA format, with and without molecule type specification. When the `MolType` is not specified it defaults to BYTES.

```
>>> from cogent3 import LoadSeqs
>>> from cogent3 import DNA, PROTEIN
>>> protein_seqs = ['>seq1','DEKQL-RG','>seq2','DDK--SRG']
>>> proteins_loaded = LoadSeqs(data=protein_seqs)
>>> proteins_loaded.moltype
MolType(('\x00', '\x01', '\x02', '\x03'...
>>> print(proteins_loaded)
>seq1
DEKQL-RG
>seq2
DDK--SRG

>>> proteins_loaded = LoadSeqs(data=protein_seqs, moltype=PROTEIN)
>>> print(proteins_loaded)
>seq1
DEKQL-RG
>seq2
DDK--SRG
```

Stripping label characters on loading

Load a list of aligned nucleotide sequences, while specifying the DNA molecule type and stripping the comments from the label. In this example, stripping is accomplished by passing a function that removes everything after the first whitespace to the `label_to_name` parameter.

```
>>> from cogent3 import LoadSeqs, DNA
>>> DNA_seqs = ['>sample1 Mus musculus','AACCTGC--C','>sample2 Gallus gallus','AAC-
↪TGCAAC']
>>> loaded_seqs = LoadSeqs(data=DNA_seqs, moltype=DNA, label_to_name=lambda x: x.
↪split()[0])
>>> print(loaded_seqs)
>sample1
AACCTGC--C
>sample2
AAC-TGCAAC
```

Using alternative constructors for the *Alignment* object

An example of using an alternative constructor is given below. A constructor is passed to the aligned parameter in lieu of `True` or `False`.

```
>>> from cogent3 import LoadSeqs
>>> from cogent3.core.alignment import ArrayAlignment
>>> seqs = ['>seq1','AATCG-A','>seq2','AATCGGA']
>>> seqs_loaded = LoadSeqs(data=seqs,array_align=True)
>>> print(seqs_loaded)
>seq1
AATCG-A
>seq2
AATCGGA
```

Loading sequences using format parsers

`LoadSeqs` is just a convenience interface to format parsers. It can sometimes be more effective to use the parsers directly, say when you don't want to load everything into memory.

Loading FASTA sequences from an open file or list of lines

To load FASTA formatted sequences directly, you can use the `MinimalFastaParser`.

Note: This returns the sequences as strings.

```
>>> from cogent3.parse.fasta import MinimalFastaParser
>>> f=open('data/long_testseqs.fasta')
>>> seqs = [(name, seq) for name, seq in MinimalFastaParser(f)]
>>> print(seqs)
[('Human', 'TGTGGCACAAATAC...
```

Handling overloaded FASTA sequence labels

The FASTA label field is frequently overloaded, with different information fields present in the field and separated by some delimiter. This can be flexibly addressed using the `LabelParser`. By creating a custom label parser, we can decided which part we use as the sequence name. We show how convert a field into something specific.

```
>>> from cogent3.parse.fasta import LabelParser
>>> def latin_to_common(latin):
...     return {'Homo sapiens': 'human',
...             'Pan troglodtyes': 'chimp'}[latin]
>>> label_parser = LabelParser("%(species)s",
...             [[1, "species", latin_to_common]], split_with=':')
>>> for label in ">abcd:Homo sapiens:misc", ">abcd:Pan troglodtyes:misc":
...     label = label_parser(label)
...     print(label, type(label))
human <class 'cogent3.parse.fasta.RichLabel'>
chimp <class 'cogent3.parse.fasta.RichLabel'>
```

The `RichLabel` objects have an `Info` object as an attribute, allowing specific reference to all the specified label fields.

```
>>> from cogent3.parse.fasta import MinimalFastaParser, LabelParser
>>> fasta_data = ['>gi|10047090|ref|NP_055147.1| small muscle protein, X-linked [Homo␣
↪sapiens]',
...  'MNMSKQPVSNVRAIQANINIPMGAFRPGAGQPPRRKECTPEVEEGVPPTSDEEKKPIPGAKKLPGPAVNL',
... 'SEIQNIKSELKYVPKAEQ',
... '>gi|10047092|ref|NP_037391.1| neuronal protein [Homo sapiens]',
... 'MANRGPSYGLSREVQEKIEQKYDADLENKLVDWIILQCAEDIEHPPPGRAHFQKWLMDGTVLCKLINSLY',
... 'PPGQEPIPKISESKMAFKQMEQISQFLKAAETYGVRTTDIFQTVDLWEGKDMAAVQRTLMALGSVAVTKD']
...
>>> label_to_name = LabelParser("%(ref)s",
...                             [[1,"gi", str],
...                              [3, "ref", str],
...                              [4, "description", str]],
...                              split_with="|")
...
```

```
>>> for name, seq in MinimalFastaParser(fasta_data, label_to_name=label_to_name):
...     print(name)
...     print(name.info.gi)
...     print(name.info.description)
NP_055147.1
10047090
 small muscle protein, X-linked [Homo sapiens]
NP_037391.1
10047092
 neuronal protein [Homo sapiens]
```

Using the `MolType` and `Sequence` objects

MolType

`MolType` provides services for resolving ambiguities, or providing the correct ambiguity for recoding. It also maintains the mappings between different kinds of alphabets, sequences and alignments.

One issue with `MolType`'s is that they need to know about `Sequence`, `Alphabet`, and other objects, but, at the same time, those objects need to know about the `MolType`. It is thus essential that the connection between these other types and the `MolType` can be made after the objects are created.

Setting up a `MolType` object with an RNA sequence

```
>>> from cogent3.core.moltype import MolType, IUPAC_RNA_chars,\
...   IUPAC_RNA_ambiguities, RnaStandardPairs, RnaMW,\
...   IUPAC_RNA_ambiguities_complements
>>> from cogent3.core.sequence import NucleicAcidSequence
>>> testrnaseq = 'ACGUACGUACGUACGU'
>>> RnaMolType = MolType(
...     seq_constructor=NucleicAcidSequence(testrnaseq),
...     motifset=IUPAC_RNA_chars,
...     ambiguities=IUPAC_RNA_ambiguities,
...     label="rna_with_lowercase",
...     mw_calculator=RnaMW,
...     complements=IUPAC_RNA_ambiguities_complements,
...     pairs= RnaStandardPairs,
...     add_lower=True,
...     preserve_existing_moltypes=True,
...     make_alphabet_group=True,
...     )
```

Setting up a `MolType` object with a DNA sequence

```
>>> from cogent3.core.moltype import MolType, IUPAC_DNA_chars,\
...   IUPAC_DNA_ambiguities, DnaMW, IUPAC_DNA_ambiguities_complements,\
...   DnaStandardPairs
>>> testdnaseq = 'ACGTACGTACGUACGT'
>>> DnaMolType = MolType(
...     seq_constructor=NucleicAcidSequence(testdnaseq),
...     motifset=IUPAC_DNA_chars,
...     ambiguities=IUPAC_DNA_ambiguities,
```

```
...     label="dna_with_lowercase",
...     mw_calculator=DnaMW,
...     complements=IUPAC_DNA_ambiguities_complements,
...     pairs=DnaStandardPairs,
...     add_lower=True,
...     preserve_existing_moltypes=True,
...     make_alphabet_group=True,
...     )
```

Setting up a DNA `MolType` object allowing `.` as gaps

```
>>> from cogent3.core import moltype as mt
>>> DNAgapped = mt.MolType(seq_constructor=mt.DnaSequence,
...                        motifset=mt.IUPAC_DNA_chars,
...                        ambiguities=mt.IUPAC_DNA_ambiguities,
...                        complements=mt.IUPAC_DNA_ambiguities_complements,
...                        pairs=mt.DnaStandardPairs,
...                        gaps='.')
>>> seq = DNAgapped.make_seq('ACG.')
```

Setting up a `MolType` object with a protein sequence

```
>>> from cogent3.core.moltype import MolType, IUPAC_PROTEIN_chars,\
...   IUPAC_PROTEIN_ambiguities, ProteinMW
>>> from cogent3.core.sequence import ProteinSequence, ArrayProteinSequence
>>> protstr = 'TEST'
>>> ProteinMolType = MolType(
...     seq_constructor=ProteinSequence(protstr),
...     motifset=IUPAC_PROTEIN_chars,
...     ambiguities=IUPAC_PROTEIN_ambiguities,
...     mw_calculator=ProteinMW,
...     make_alphabet_group=True,
...     array_seq_constructor=ArrayProteinSequence,
...     label="protein")
>>> protseq = ProteinMolType.make_seq
```

Verify sequences

```
>>> rnastr = 'ACGUACGUACGUACGU'
>>> dnastr = 'ACGTACGTACGTACGT'
>>> RnaMolType.is_valid(rnastr)
True
>>> RnaMolType.is_valid(dnastr)
False
>>> RnaMolType.is_valid(NucleicAcidSequence(dnastr).to_rna())
True
```

Sequence

The Sequence object contains classes that represent biological sequence data. These provide generic biological sequence manipulation functions, plus functions that are critical for the evolve module calculations.

Warning: Do not import sequence classes directly! It is expected that you will access them through MolType objects. The most common molecular types DNA, RNA, PROTEIN are provided as top level imports in cogent3 (e.g. cogent3.DNA). Sequence classes depend on information from the MolType that is **only** available after MolType has been imported. Sequences are intended to be immutable. This is not enforced by the code for performance reasons, but don't alter the MolType or the sequence data after creation.

More detailed usage of sequence objects can be found in *DNA and RNA sequences*.

Alphabets

Alphabet and MolType

MolType instances have an Alphabet.

```
>>> from cogent3 import DNA, PROTEIN
>>> print(DNA.alphabet)
('T', 'C', 'A', 'G')
>>> print(PROTEIN.alphabet)
('A', 'C', 'D', 'E', ...
```

Alphabet instances have a MolType.

```
>>> PROTEIN.alphabet.moltype == PROTEIN
True
```

Creating tuple alphabets

You can create a tuple alphabet of, for example, dinucleotides or trinucleotides.

```
>>> dinuc_alphabet = DNA.alphabet.get_word_alphabet(2)
>>> print(dinuc_alphabet)
('TT', 'CT', 'AT', 'GT', ...
>>> trinuc_alphabet = DNA.alphabet.get_word_alphabet(3)
>>> print(trinuc_alphabet)
('TTT', 'CTT', 'ATT', ...
```

Convert a sequence into integers

```
>>> seq = 'TAGT'
>>> indices = DNA.alphabet.to_indices(seq)
>>> indices
[0, 2, 3, 0]
```

Convert integers to a sequence

```
>>> seq = DNA.alphabet.from_indices([0,2,3,0])
>>> seq
['T', 'A', 'G', 'T']
```

or

```
>>> seq = DNA.alphabet.from_ordinals_to_seq([0,2,3,0])
>>> seq
DnaSequence(TAGT)
```

DNA and RNA sequences

Creating a DNA sequence from a string

All sequence and alignment objects have a molecular type, or `MolType` which provides key properties for validating sequence characters. Here we use the `DNA` `MolType` to create a DNA sequence.

```
>>> from cogent3 import DNA
>>> my_seq = DNA.make_seq("AGTACACTGGT")
>>> my_seq
DnaSequence(AGTACAC... 11)
>>> print(my_seq)
AGTACACTGGT
>>> str(my_seq)
'AGTACACTGGT'
```

Creating a RNA sequence from a string

```
>>> from cogent3 import RNA
>>> rnaseq = RNA.make_seq('ACGUACGUACGUACGU')
```

Converting to FASTA format

```
>>> from cogent3 import DNA
>>> my_seq = DNA.make_seq('AGTACACTGGT')
>>> print(my_seq.to_fasta())
>0
AGTACACTGGT
```

Convert a RNA sequence to FASTA format

```
>>> from cogent3 import RNA
>>> rnaseq = RNA.make_seq('ACGUACGUACGUACGU')
>>> rnaseq.to_fasta()
'>0\nACGUACGUACGUACGU'
```

Creating a named sequence

```
>>> from cogent3 import DNA
>>> my_seq = DNA.make_seq('AGTACACTGGT','my_gene')
>>> my_seq
DnaSequence(AGTACAC... 11)
>>> type(my_seq)
<class 'cogent3.core.sequence.DnaSequence'>
```

Setting or changing the name of a sequence

```
>>> from cogent3 import DNA
>>> my_seq = DNA.make_seq('AGTACACTGGT')
>>> my_seq.name = 'my_gene'
>>> print(my_seq.to_fasta())
>my_gene
AGTACACTGGT
```

Complementing a DNA sequence

```
>>> from cogent3 import DNA
>>> my_seq = DNA.make_seq("AGTACACTGGT")
>>> print(my_seq.complement())
TCATGTGACCA
```

Reverse complementing a DNA sequence

```
>>> print(my_seq.reversecomplement())
ACCAGTGTACT
```

The `rc` method name is easier to type

```
>>> print(my_seq.rc())
ACCAGTGTACT
```

Translate a `DnaSequence` to protein

```
>>> from cogent3 import DNA
>>> my_seq = DNA.make_seq('GCTTGGGAAAGTCAAATGGAA','protein-X')
>>> pep = my_seq.get_translation()
>>> type(pep)
<class 'cogent3.core.sequence.ProteinSequence'>
>>> print(pep.to_fasta())
>protein-X
AWESQME
```

Converting a DNA sequence to RNA

```
>>> from cogent3 import DNA
>>> my_seq = DNA.make_seq('ACGTACGTACGTACGT')
>>> print(my_seq.to_rna())
ACGUACGUACGUACGU
```

Convert an RNA sequence to DNA

```
 >>> from cogent3 import RNA
>>> rnaseq = RNA.make_seq('ACGUACGUACGUACGU')
>>> print(rnaseq.to_dna())
ACGTACGTACGTACGT
```

Testing complementarity

```
>>> from cogent3 import DNA
>>> a = DNA.make_seq("AGTACACTGGT")
>>> a.can_pair(a.complement())
False
>>> a.can_pair(a.reversecomplement())
True
```

Joining two DNA sequences

```
>>> from cogent3 import DNA
>>> my_seq = DNA.make_seq("AGTACACTGGT")
>>> extra_seq = DNA.make_seq("CTGAC")
>>> long_seq = my_seq + extra_seq
>>> long_seq
DnaSequence(AGTACAC... 16)
>>> str(long_seq)
'AGTACACTGGTCTGAC'
```

Slicing DNA sequences

```
>>> my_seq[1:6]
DnaSequence(GTACA)
```

Getting 3rd positions from codons

The easiest approach is to work off the `cogent3` `ArrayAlignment` object.

We'll do this by specifying the position indices of interest, creating a sequence `Feature` and using that to extract the positions.

```
>>> from cogent3 import DNA
>>> seq = DNA.make_array_seq('ATGATGATGATG')
>>> pos3 = seq[2::3]
>>> assert str(pos3) == 'GGGG'
```

Getting 1st and 2nd positions from codons

In this instance we can use the annotatable sequence classes.

```
>>> from cogent3 import DNA
>>> seq = DNA.make_seq('ATGATGATGATG')
>>> indices = [(i, i+2) for i in range(len(seq))[::3]]
>>> pos12 = seq.add_feature('pos12', 'pos12', indices)
>>> pos12 = pos12.get_slice()
>>> assert str(pos12) == 'ATATATAT'
```

Return a randomized version of the sequence

```
print rnaseq.shuffle()
ACAACUGGCUCUGAUG
```

Remove gaps from a sequence

```
 >>> from cogent3 import RNA
>>> s = RNA.make_seq('--AUUAUGCUAU-UAu--')
>>> print(s.degap())
AUUAUGCUAUUAU
```

Protein sequences

Creating a ProteinSequence with a name

```
>>> from cogent3 import PROTEIN
>>> p = PROTEIN.make_seq('THISISAPRQTEIN','myProtein')
>>> type(p)
<class 'cogent3.core.sequence.ProteinSequence'>
>>> str(p)
'THISISAPRQTEIN'
```

Converting a DNA sequence string to protein sequence string

```
>>> from cogent3.core.genetic_code import DEFAULT as standard_code
>>> standard_code.translate('TTTGCAAAC')
'FAN'
```

Conversion to a `ProteinSequence` from a `DnaSequence` is shown in *Translate a DnaSequence to protein*.

Loading protein sequences from a Phylip file

```
>>> from cogent3 import LoadSeqs, PROTEIN
>>> seq = LoadSeqs('data/abglobin_aa.phylip', moltype=PROTEIN,
...                aligned=True)
```

Loading other formats, or collections of sequences is shown in *Loading nucleotide, protein sequences*.

Collections and Alignments

For loading collections of unaligned or aligned sequences see *Loading nucleotide, protein sequences*.

Basic Collection objects

Constructing a `SequenceCollection` or `Alignment` object from strings

```
>>> from cogent3 import LoadSeqs, DNA
>>> dna  = {'seq1': 'ATGACC',
...         'seq2': 'ATCGCC'}
>>> seqs = LoadSeqs(data=dna, moltype=DNA)
>>> print(type(seqs))
<class 'cogent3.core.alignment.ArrayAlignment'>
>>> seqs = LoadSeqs(data=dna, moltype=DNA, aligned=False)
>>> print(type(seqs))
<class 'cogent3.core.alignment.SequenceCollection'>
```

Constructing a `ArrayAlignment` using `LoadSeqs`

```
>>> from cogent3 import LoadSeqs, DNA
>>> dna  = {'seq1': 'ATGACC',
...         'seq2': 'ATCGCC'}
>>> seqs = LoadSeqs(data=dna, moltype=DNA, aligned=True, array_align=True)
>>> print(type(seqs))
<class 'cogent3.core.alignment.ArrayAlignment'>
>>> print(seqs)
>seq1
ATGACC
>seq2
ATCGCC
```

Converting a `SequenceCollection` to FASTA format

```
>>> from cogent3 import LoadSeqs
>>> seq = LoadSeqs('data/test.paml', aligned=False)
>>> fasta_data = seq.to_fasta()
>>> print(fasta_data)
>DogFaced
GCAAGGAGCCAGCAGAACAGATGGGTTGAAACTAAGGAAACATGTAATGATAGGCAGACT
>HowlerMon
GCAAGGAGCCAACATAACAGATGGGCTGAAAGTGAGGAAACATGTAATGATAGGCAGACT
```

```
>Human
GCAAGGAGCCAACATAACAGATGGGCTGGAAGTAAGGAAACATGTAATGATAGGCGGACT
>Mouse
GCAGTGAGCCAGCAGAGCAGATGGGCTGCAAGTAAAGGAACATGTAACGACAGGCAGGTT
>NineBande
GCAAGGCGCCAACAGAGCAGATGGGCTGAAAGTAAGGAAACATGTAATGATAGGCAGACT
```

Adding new sequences to an existing collection or alignment

New sequences can be either appended or inserted using the `add_seqs` method. More than one sequence can be added at the same time. Note that `add_seqs` does not modify the existing collection/alignment, it creates a new one.

Appending the sequences

`add_seqs` without additional parameters will append the sequences to the end of the collection/alignment.

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs(data= [('seq1', 'ATGAA------'),
...                       ('seq2', 'ATG-AGTGATG'),
...                       ('seq3', 'AT--AG-GATG')], moltype=DNA)
>>> print(aln)
>seq1
ATGAA------
>seq2
ATG-AGTGATG
>seq3
AT--AG-GATG

>>> new_seqs = LoadSeqs(data=[('seq0', 'ATG-AGT-AGG'),
...                           ('seq4', 'ATGCC------')], moltype=DNA)
>>> new_aln = aln.add_seqs(new_seqs)
>>> print(new_aln)
>seq1
ATGAA------
>seq2
ATG-AGTGATG
>seq3
AT--AG-GATG
>seq0
ATG-AGT-AGG
>seq4
ATGCC------
```

Note: The order is not preserved if you use `to_fasta` method, which sorts sequences by name.

Inserting the sequences

Sequences can be inserted into an alignment at the specified position using either the `before_name` or `after_name` arguments.

```
>>> new_aln = aln.add_seqs(new_seqs, before_name='seq2')
>>> print(new_aln)
>seq1
ATGAA------
>seq0
ATG-AGT-AGG
>seq4
ATGCC------
>seq2
ATG-AGTGATG
>seq3
AT--AG-GATG

>>> new_aln = aln.add_seqs(new_seqs, after_name='seq2')
>>> print(new_aln)
>seq1
ATGAA------
>seq2
ATG-AGTGATG
>seq0
ATG-AGT-AGG
>seq4
ATGCC------
>seq3
AT--AG-GATG
```

Inserting sequence(s) based on their alignment to a reference sequence

Already aligned sequences can be added to an existing `Alignment` object and aligned at the same time using the `add_from_ref_aln` method. The alignment is performed based on their alignment to a reference sequence (which must be present in both alignments). The method assumes the first sequence in `ref_aln.names[0]` is the reference.

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs(data=[('seq1', 'ATGAA------'),
...                      ('seq2', 'ATG-AGTGATG'),
...                      ('seq3', 'AT--AG-GATG')], moltype=DNA)
>>> ref_aln = LoadSeqs(data=[('seq3', 'ATAGGATG'),
...                          ('seq0', 'ATG-AGCG'),
...                          ('seq4', 'ATGCTGGG')], moltype=DNA)
>>> new_aln = aln.add_from_ref_aln(ref_aln)
>>> print(new_aln)
>seq1
ATGAA------
>seq2
ATG-AGTGATG
>seq3
AT--AG-GATG
>seq0
AT--G--AGCG
>seq4
AT--GC-TGGG
```

`add_from_ref_aln` has the same arguments as `add_seqs` so `before_name` and `after_name` can be used to insert the new sequences at the desired position.

Note: This method does not work with the `ArrayAlignment` class.

Removing all columns with gaps in a named sequence

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs(data=[('seq1', 'ATGAA---TG-'),
...                      ('seq2', 'ATG-AGTGATG'),
...                      ('seq3', 'AT--AG-GATG')], moltype=DNA)
>>> new_aln = aln.get_degapped_relative_to('seq1')
>>> print(new_aln)
>seq1
ATGAATG
>seq2
ATG-AAT
>seq3
AT--AAT
```

The elements of a collection or alignment

Accessing individual sequences from a collection or alignment by name

Using the `get_seq` method allows for extracting an unaligned sequence from a collection or alignment by name.

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs(data=[('seq1', 'ATGAA------'),
...                      ('seq2', 'ATG-AGTGATG'),
...                      ('seq3', 'AT--AG-GATG')],
...                 moltype=DNA, array_align=False)
>>> seq = aln.get_seq('seq1')
>>> seq.name
'seq1'
>>> type(seq)
<class 'cogent3.core.sequence.DnaSequence'>
>>> seq.is_gapped()
False
```

Alternatively, if you want to extract the aligned (i.e., gapped) sequence from an alignment, you can use `get_gapped_seq`.

```
>>> seq = aln.get_gapped_seq('seq1')
>>> seq.is_gapped()
True
>>> print(seq)
ATGAA------
```

To see the names of the sequences in a sequence collection, you can use either the `Names` attribute or `get_seq_names` method.

```
>>> aln.names
['seq1', 'seq2', 'seq3']
>>> aln.get_seq_names()
['seq1', 'seq2', 'seq3']
```

Slice the sequences from an alignment like a list

The usual approach is to access a `SequenceCollection` or `Alignment` object as a dictionary, obtaining the individual sequences using the titles as "keys" (above). However, one can also iterate through the collection like a list.

```
>>> from cogent3 import LoadSeqs, DNA
>>> fn = 'data/long_testseqs.fasta'
>>> seqs = LoadSeqs(fn, moltype=DNA, aligned=False)
>>> my_seq = seqs.seqs[0]
>>> my_seq[:24]
DnaSequence(TGTGGCA... 24)
>>> str(my_seq[:24])
'TGTGGCACAAATACTCATGCCAGC'
>>> type(my_seq)
<class 'cogent3.core.sequence.DnaSequence'>
>>> aln = LoadSeqs(fn, moltype=DNA, aligned=True)
>>> aln.seqs[0][:24]
DnaSequence(TGTGGCA... 24)
>>> print(aln.seqs[0][:24])
TGTGGCACAAATACTCATGCCAGC
```

Getting a subset of sequences from the alignment

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs('data/test.paml', moltype=DNA)
>>> aln.names
['NineBande', 'Mouse', 'Human', 'HowlerMon', 'DogFaced']
>>> new = aln.take_seqs(['Human', 'HowlerMon'])
>>> new.names
['Human', 'HowlerMon']
```

Note, if you set `array_align=False`, then the subset contain references to the original sequences, not copies.

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs('data/test.paml', array_align=False, moltype=DNA)
>>> seq = aln.get_seq('Human')
>>> new = aln.take_seqs(['Human', 'HowlerMon'])
>>> id(new.get_seq('Human')) == id(aln.get_seq('Human'))
True
```

Alignments

Creating an `Alignment` object from a `SequenceCollection`

```
>>> from cogent3.core.alignment import Alignment
>>> seq = LoadSeqs('data/test.paml', aligned=False)
>>> aln = Alignment(seq)
>>> fasta_1 = seq.to_fasta()
>>> fasta_2 = aln.to_fasta()
>>> assert fasta_1 == fasta_2
```

Convert alignment to DNA, RNA or PROTEIN moltypes

This is useful if you've loaded a sequence alignment without specifying the moltype and later need to convert it.

```
>>> from cogent3 import LoadSeqs
>>> data = [('a', 'ACG---'), ('b', 'CCTGGG')]
>>> aln = LoadSeqs(data=data)
>>> dna = aln.to_dna()
>>> dna
2 x 6 dna alignment: a[ACG---], b[CCTGGG]
```

To RNA

```
>>> from cogent3 import LoadSeqs
>>> data = [('a', 'ACG---'), ('b', 'CCUGGG')]
>>> aln = LoadSeqs(data=data)
>>> rna = aln.to_rna()
>>> rna
2 x 6 rna alignment: a[ACG---], b[CCUGGG]
```

To PROTEIN

```
>>> from cogent3 import LoadSeqs
>>> data = [('x', 'TYV'), ('y', 'TE-')]
>>> aln = LoadSeqs(data=data)
>>> prot = aln.to_protein()
>>> prot
2 x 3 protein alignment: x[TYV], y[TE-]
```

Handling gaps

Remove all gaps from an alignment in FASTA format

This necessarily returns a `SequenceCollection`.

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs("data/primate_cdx2_promoter.fasta")
>>> degapped = aln.degap()
>>> print(type(degapped))
<class 'cogent3.core.alignment.SequenceCollection'>
```

Writing sequences to file

Both collection and alignment objects have a `write` method. The output format is inferred from the filename suffix,

```
>>> from cogent3 import LoadSeqs, DNA
>>> dna  = {'seq1': 'ATGACC',
...         'seq2': 'ATCGCC'}
>>> aln = LoadSeqs(data=dna, moltype=DNA)
>>> aln.write('sample.fasta')
```

or by the `format` argument.

```
>>> aln.write('sample', format='fasta')
```

Converting an alignment to FASTA format

```
>>> from cogent3.core.alignment import Alignment
>>> seq = LoadSeqs('data/long_testseqs.fasta')
>>> aln = Alignment(seq)
>>> fasta_align = aln.to_fasta()
```

Converting an alignment into Phylip format

```
>>> from cogent3.core.alignment import Alignment
>>> seq = LoadSeqs('data/test.paml')
>>> aln = Alignment(seq)
>>> phylip_file, name_dictionary = aln.to_phylip()
```

Converting an alignment to a list of strings

```
>>> from cogent3.core.alignment import Alignment
>>> seq = LoadSeqs('data/test.paml')
>>> aln = Alignment(seq)
>>> string_list = aln.todict().values()
```

Slicing an alignment

By rows (sequences)

An `Alignment` can be sliced

```
>>> from cogent3 import LoadSeqs, DNA
>>> fn = 'data/long_testseqs.fasta'
>>> aln = LoadSeqs(fn, moltype=DNA, aligned=True)
>>> print(aln[:24])
>Human
TGTGGCACAAATACTCATGCCAGC
>HowlerMon
TGTGGCACAAATACTCATGCCAGC
>Mouse
TGTGGCACAGATGCTCATGCCAGC
>NineBande
TGTGGCACAAATACTCATGCCAAC
>DogFaced
TGTGGCACAAATACTCATGCCAAC
```

but a `SequenceCollection` cannot be sliced

```
>>> from cogent3 import LoadSeqs, DNA
>>> fn = 'data/long_testseqs.fasta'
>>> seqs = LoadSeqs(fn, moltype=DNA, aligned=False)
```

```
>>> print(seqs[:24])
Traceback (most recent call last):
TypeError: 'SequenceCollection' object...
```

Getting a single column from an alignment

```
>>> from cogent3.core.alignment import Alignment
>>> seq = LoadSeqs('data/test.paml')
>>> aln = Alignment(seq)
>>> column_four = aln[3]
```

Getting a region of contiguous columns

```
>>> from cogent3.core.alignment import Alignment
>>> aln = LoadSeqs('data/long_testseqs.fasta')
>>> region = aln[50:70]
```

Iterating over alignment positions

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs('data/primate_cdx2_promoter.fasta')
>>> col = aln[113:115].iter_positions()
>>> type(col)
<class 'generator'>
>>> list(col)
[[ByteSequence(A), ByteSequence(A), ByteSequence(A)], [ByteSequence(T)...
```

Getting codon 3rd positions from `Alignment`

We'll do this by specifying the position indices of interest, creating a sequence `Feature` and using that to extract the positions.

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs(data={'seq1': 'ATGATGATG---',
...                      'seq2': 'ATGATGATGATG'}, array_align=False)
>>> list(range(len(aln))[2::3])
[2, 5, 8, 11]
>>> indices = [(i, i+1) for i in range(len(aln))[2::3]]
>>> indices
[(2, 3), (5, 6), (8, 9), (11, 12)]
>>> pos3 = aln.add_feature('pos3', 'pos3', indices)
>>> pos3 = pos3.get_slice()
>>> print(pos3)
>seq2
GGGG
>seq1
GGG-
```

Getting codon 3rd positions from `ArrayAlignment`

We can use more conventional slice notation in this instance. Note, because Python counts from 0, the 3rd position starts at index 2.

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs(data={'seq1': 'ATGATGATG---',
...                      'seq2': 'ATGATGATGATG'}, array_align=True)
>>> pos3 = aln[2::3]
>>> print(pos3)
>seq1
GGG-
>seq2
GGGG
```

Filtering positions

Trim terminal stop codons

For evolutionary analyses that use codon models we need to exclude terminating stop codons. For the case where the sequences are all of length divisible by 3.

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs(data={'seq1': 'ACGTAA---',
...                      'seq2': 'ACGACA---',
...                      'seq3': 'ACGCAATGA'}, moltype=DNA)
...
>>> new = aln.trim_stop_codons()
>>> print(new)
>seq3
ACGCAA
>seq2
ACGACA
>seq1
ACG---
```

If the alignment contains sequences not divisible by 3, use the `allow_partial` argument.

```
>>> aln = LoadSeqs(data={'seq1': 'ACGTAA---',
...                      'seq2': 'ACGAC----', # terminal codon incomplete
...                      'seq3': 'ACGCAATGA'}, moltype=DNA)
...
>>> new = aln.trim_stop_codons(allow_partial=True)
>>> print(new)
>seq3
ACGCAA
>seq2
ACGAC-
>seq1
ACG---
```

Eliminating columns with non-nucleotide characters

We sometimes want to eliminate ambiguous or gap data from our alignments. We show how to exclude alignment columns by the characters they contain. In the first instance we do this just for single nucleotide columns, then for trinucleotides (equivalent for handling codons). Both are done using the `no_degenerates` method.

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs(data= [('seq1', 'ATGAAGGTG---'),
...                       ('seq2', 'ATGAAGGTGATG'),
...                       ('seq3', 'ATGAAGGNGATG')], moltype=DNA)
```

We apply to nucleotides,

```
>>> nucs = aln.no_degenerates()
>>> print(nucs)
>seq1
ATGAAGGG
>seq2
ATGAAGGG
>seq3
ATGAAGGG
```

Applying the same filter to trinucleotides (specified by setting `motif_length=3`).

```
>>> trinucs = aln.no_degenerates(motif_length=3)
>>> print(trinucs)
>seq1
ATGAAG
>seq2
ATGAAG
>seq3
ATGAAG
```

Getting all variable positions from an alignment

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs('data/long_testseqs.fasta')
>>> pos = aln.variable_positions()
>>> just_variable_aln = aln.take_positions(pos)
>>> print(just_variable_aln[:10])
>Human
AAGCAAAACT
>HowlerMon
AAGCAAGACT
>Mouse
GGGCCCAGCT
>NineBande
AAATAAAACT
>DogFaced
AAACAAAATA
```

Getting all constant positions from an alignment

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs('data/long_testseqs.fasta')
>>> pos = aln.variable_positions()
>>> just_constant_aln = aln.take_positions(pos, negate=True)
>>> print(just_constant_aln[:10])
>Human
TGTGGCACAA
>HowlerMon
TGTGGCACAA
>Mouse
TGTGGCACAA
>NineBande
TGTGGCACAA
>DogFaced
TGTGGCACAA
```

Getting all variable codons from an alignment

This is done using the `filtered` method using the `motif_length` argument. We demonstrate this first for the `ArrayAlignment`.

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs('data/long_testseqs.fasta')
>>> variable_codons = aln.filtered(lambda x: len(set(map(tuple, x))) > 1,
...                                motif_length=3)
>>> print(just_variable_aln[:9])
>Human
AAGCAAAAC
>HowlerMon
AAGCAAGAC
>Mouse
GGGCCCAGC
>NineBande
AAATAAAAC
>DogFaced
AAACAAAAT
```

Then for the standard `Alignment` by first converting the `ArrayAlignment`.

```
>>> aln = aln.to_type(array_align=False)
>>> variable_codons = aln.filtered(lambda x: len(set(''.join(x))) > 1,
...                                motif_length=3)
>>> print(just_variable_aln[:9])
>Human
AAGCAAAAC...
```

Filtering sequences

Extracting sequences by sequence identifier into a new alignment object

You can use `take_seqs` to extract some sequences by sequence identifier from an alignment to a new alignment object:

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs('data/long_testseqs.fasta')
>>> aln.take_seqs(['Human','Mouse'])
2 x 2532 bytes alignment: Human[TGTGGCACAAA...], Mouse[TGTGGCACAGA...]
```

Alternatively, you can extract only the sequences which are not specified by passing `negate=True`:

```
>>> aln.take_seqs(['Human','Mouse'],negate=True)
3 x 2532 bytes alignment: NineBande[TGTGGCACAAA...], HowlerMon[TGTGGCACAAA...],␣
↪DogFaced[TGTGGCACAAA...]
```

Extracting sequences using an arbitrary function into a new alignment object

You can use `take_seqs_if` to extract sequences into a new alignment object based on whether an arbitrary function applied to the sequence evaluates to True. For example, to extract sequences which don't contain any N bases you could do the following:

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs(data= [('seq1', 'ATGAAGGTG---'),
...                       ('seq2', 'ATGAAGGTGATG'),
...                       ('seq3', 'ATGAAGGNGATG')], moltype=DNA)
>>> def no_N_chars(s):
...     return 'N' not in s
>>> aln.take_seqs_if(no_N_chars)
2 x 12 dna alignment: seq1[ATGAAGGTG--...], seq2[ATGAAGGTGAT...]
```

You can additionally get the sequences where the provided function evaluates to False:

```
>>> aln.take_seqs_if(no_N_chars,negate=True)
1 x 12 dna alignment: seq3[ATGAAGGNGAT...]
```

Computing alignment statistics

Getting motif counts

We state the motif length we want and whether to allow gap or ambiguous characters. The latter only has meaning for IPUAC character sets (the DNA, RNA or PROTEIN moltypes). We illustrate this for the DNA moltype with motif lengths of 1 and 3.

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs(data= [('seq1', 'ATGAAGGTG---'),
...                       ('seq2', 'ATGAAGGTGATG'),
...                       ('seq3', 'ATGAAGGNGATG')], moltype=DNA)
>>> counts = aln.counts()
>>> print(counts)
Counter({'G': 14, 'A': 11, 'T': 7})
>>> counts = aln.counts(motif_length=3)
>>> print(counts)
Counter({'ATG': 5, 'AAG': 3, 'GTG': 2})
>>> counts = aln.counts(include_ambiguity=True)
>>> print(counts)
Counter({'G': 14, 'A': 11, 'T': 7, 'N': 1})
```

Note: Only the observed motifs are returned, rather than all defined by the alphabet.

Computing motif probabilities from an alignment

The method `get_motif_probs` of `Alignment` objects returns the probabilities for all motifs of a given length. For individual nucleotides:

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs('data/primate_cdx2_promoter.fasta', moltype=DNA)
>>> motif_probs = aln.get_motif_probs()
>>> print(motif_probs)
{'A': 0.24...
```

For dinucleotides or longer, we need to pass in an `Alphabet` with the appropriate word length. Here is an example with trinucleotides:

```
>>> from cogent3 import LoadSeqs, DNA
>>> trinuc_alphabet = DNA.alphabet.get_word_alphabet(3)
>>> aln = LoadSeqs('data/primate_cdx2_promoter.fasta', moltype=DNA)
>>> motif_probs = aln.get_motif_probs(alphabet=trinuc_alphabet)
>>> for m in sorted(motif_probs, key=lambda x: motif_probs[x],
...                 reverse=True):
...     print("%s  %.3f" % (m, motif_probs[m]))
...
CAG  0.037
CCT  0.034
CGC  0.030...
```

The same holds for other arbitrary alphabets, as long as they match the alignment `MolType`.

Some calculations in `cogent3` require all non-zero values in the motif probabilities, in which case we use a pseudo-count. We illustrate that here with a simple example where T is missing. Without the pseudo-count, the frequency of T is 0.0, with the pseudo-count defined as 1e-6 then the frequency of T will be slightly less than 1e-6.

```
>>> aln = LoadSeqs(data=[('a', 'AACAAC'),('b', 'AAGAAG')], moltype=DNA)
>>> motif_probs = aln.get_motif_probs()
>>> assert motif_probs['T'] == 0.0
>>> motif_probs = aln.get_motif_probs(pseudocount=1e-6)
>>> assert 0 < motif_probs['T'] <= 1e-6
```

It is important to notice that motif probabilities are computed by treating sequences as non-overlapping tuples. Below is a very simple pair of identical sequences where there are clearly 2 'AA' dinucleotides per sequence but only the first one is 'in-frame' (frame width = 2).

We then create a dinucleotide `Alphabet` object and use this to get dinucleotide probabilities. These frequencies are determined by breaking each aligned sequence up into non-overlapping dinucleotides and then doing a count. The expected value for the 'AA' dinucleotide in this case will be 2/8 = 0.25.

```
>>> seqs = [('a', 'AACGTAAG'), ('b', 'AACGTAAG')]
>>> aln = LoadSeqs(data=seqs, moltype=DNA)
>>> dinuc_alphabet = DNA.alphabet.get_word_alphabet(2)
>>> motif_probs = aln.get_motif_probs(alphabet=dinuc_alphabet)
>>> assert motif_probs['AA'] == 0.25
```

What about counting the total incidence of dinucleotides including those not in-frame? A naive application of the Python string object's count method will not work as desired either because it "returns the number of non-overlapping occurrences".

```
>>> seqs = [('my_seq', 'AAAGTAAG')]
>>> aln = LoadSeqs(data=seqs, moltype=DNA)
>>> my_seq = aln.get_seq('my_seq')
>>> my_seq.count('AA')
2
>>> 'AAA'.count('AA')
1
>>> 'AAAA'.count('AA')
2
```

To count all occurrences of a given dinucleotide in a DNA sequence, one could use a standard Python approach such as list comprehension:

```
>>> from cogent3 import Sequence, DNA
>>> seq = Sequence(moltype=DNA, seq='AAAGTAAG')
>>> seq
DnaSequence(AAAGTAAG)
>>> di_nucs = [seq[i:i+2] for i in range(len(seq)-1)]
>>> sum([nn == 'AA' for nn in di_nucs])
3
```

Working with alignment gaps

Filtering extracted columns for the gap character

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs('data/primate_cdx2_promoter.fasta')
>>> col = aln[113:115].iter_positions()
>>> c1, c2 = list(col)
>>> c1, c2
([ByteSequence(A), ByteSequence(A), ByteSequence(A)], [ByteSequence(T),...
>>> list(filter(lambda x: x == '-', c1))
[]
>>> list(filter(lambda x: x == '-', c2))
[ByteSequence(-), ByteSequence(-)]
```

Calculating the gap fraction

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs('data/primate_cdx2_promoter.fasta')
>>> for column in aln[113:150].iter_positions():
...     ungapped = list(filter(lambda x: x == '-', column))
...     gap_fraction = len(ungapped) * 1.0 / len(column)
...     print(gap_fraction)
0.0
0.66666...
```

Extracting maps of aligned to unaligned positions (i.e., gap maps)

It's often important to know how an alignment position relates to a position in one or more of the sequences in the alignment. The `gap_maps` method of the individual sequences is useful for this. To get a map of sequence to alignment positions for a specific sequence in your alignment, do the following:

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs(data=[('seq1', 'ATGAAGG-TG--'),
...                      ('seq2', 'ATG-AGGTGATG'),
...                      ('seq3', 'ATGAAG--GATG')], moltype=DNA)
>>> seq_to_aln_map = aln.get_gapped_seq('seq1').gap_maps()[0]
```

It's now possible to look up positions in the `seq1`, and find out what they map to in the alignment:

```
>>> seq_to_aln_map[3]
3
>>> seq_to_aln_map[8]
9
```

This tells us that in position 3 in `seq1` corresponds to position 3 in `aln`, and that position 8 in `seq1` corresponds to position 9 in `aln`.

Notice that we grabbed the first result from the call to `gap_maps`. This is the sequence position to alignment position map. The second value returned is the alignment position to sequence position map, so if you want to find out what sequence positions the alignment positions correspond to (opposed to what alignment positions the sequence positions correspond to) for a given sequence, you would take the following steps:

```
>>> aln_to_seq_map = aln.get_gapped_seq('seq1').gap_maps()[1]
>>> aln_to_seq_map[3]
3
>>> aln_to_seq_map[8]
7
```

If an alignment position is a gap, and therefore has no corresponding sequence position, you'll get a `KeyError`.

```
>>> seq_pos = aln_to_seq_map[7]
Traceback (most recent call last):
KeyError: 7
```

Note: The first position in alignments and sequences is always numbered position 0.

Filtering alignments based on gaps

Note: An alternate, computationally faster, approach to removing gaps is to use the `filtered` method as discussed in *Filtering positions*.

The `omit_gap_runs` method can be applied to remove long stretches of gaps in an alignment. In the following example, we remove sequences that have more than two adjacent gaps anywhere in the aligned sequence.

```
>>> aln = LoadSeqs(data=[('seq1', 'ATGAA---TG-'),
...                      ('seq2', 'ATG-AGTGATG'),
...                      ('seq3', 'AT--AG-GATG')], moltype=DNA)
```

```
>>> print(aln.omit_gap_runs(2).to_fasta())
>seq2
ATG-AGTGATG
>seq3
AT--AG-GATG
```

If instead, we just wanted to remove positions from the alignment which are gaps in more than a certain percentage of the sequences, we could use the `omit_gap_pos` function. For example:

```
>>> aln = LoadSeqs(data=[('seq1', 'ATGAA---TG-'),
...                      ('seq2', 'ATG-AGTGATG'),
...                      ('seq3', 'AT--AG-GATG')], moltype=DNA)
>>> print(aln.omit_gap_pos(0.40).to_fasta())
>seq1
ATGA--TG-
>seq2
ATGAGGATG
>seq3
AT-AGGATG
```

You'll notice that the 4th and 7th columns of the alignment have been removed because they contained 66% gaps – more than the allowed 40%.

If you wanted to remove sequences which contain more than a certain percent gap characters, you could use the `omit_gap_seqs` method. This is commonly applied to filter partial sequences from an alignment.

```
>>> aln = LoadSeqs(data=[('seq1', 'ATGAA------'),
...                      ('seq2', 'ATG-AGTGATG'),
...                      ('seq3', 'AT--AG-GATG')], moltype=DNA)
>>> filtered_aln = aln.omit_gap_seqs(0.50)
>>> print(filtered_aln.to_fasta())
>seq2
ATG-AGTGATG
>seq3
AT--AG-GATG
```

Note that following this call to `omit_gap_seqs`, the 4th column of `filtered_aln` is 100% gaps. This is generally not desirable, so a call to `omit_gap_seqs` is frequently followed with a call to `omit_gap_pos` with no parameters – this defaults to removing positions which are all gaps:

```
>>> print(filtered_aln.omit_gap_pos().to_fasta())
>seq2
ATGAGTGATG
>seq3
AT-AG-GATG
```

Annotations

Annotations with coordinates

For more extensive documentation about annotations see *Advanced sequence handling*.

Automated introduction from reading genbank files

We load a sample genbank file with plenty of features and grab the CDS features.

```
>>> from cogent3.parse.genbank import RichGenbankParser
>>> parser = RichGenbankParser(open('data/ST_genome_part.gb'))
>>> for accession, seq in parser:
...     print(accession)
...
AE006468
>>> cds = seq.get_annotations_matching('CDS')
>>> print(cds)
[CDS "thrL" at [189:255]/10020, CDS "thrA" at ...
```

Customising annotation construction from reading a genbank file

You can write your own code to construct annotation objects. One reason you might do this is some genbank files do not have a `/gene` tag on gene related features, instead only possessing a `/locus_tag`. For illustrating the approach we only create annotations for `CDS` features. We write a custom callback function that uses the `locus_tag` as the `Feature` name.

```
>>> from cogent3.core.annotation import Feature
>>> def add_annotation(seq, feature, spans):
...     type_ = feature['type']
...     if type_ != 'CDS':
...         return
...     name = feature.get('locus_tag', None)
...     if name and not isinstance(name, str):
...         name = ' '.join(name)
...     seq.add_annotation(Feature, type_, name, spans)
...
>>> parser = RichGenbankParser(open('data/ST_genome_part.gb'),
...           add_annotation=add_annotation)
>>> for accession, seq in parser: # just reading one accession,sequence
...     break
...
>>> genes = seq.get_annotations_matching('CDS')
>>> print(genes)
[CDS "STM0001" at [189:255]/10020, CDS "STM0002" at [336:2799]/10020...
```

Creating directly on a sequence

```
>>> from cogent3 import DNA
>>> from cogent3.core.annotation import Feature
>>> s1 = DNA.make_seq("AAGAAGAAGACCCCCAAAAAAAAAA"\
...                       "TTTTTTTTTTAAAAAGGGAACCCT",
...                       name="seq1")
...
>>> print(s1[10:15]) # this will be exon 1
CCCCC
>>> print(s1[30:40]) # this will be exon 2
TTTTTAAAAA
>>> print(s1[45:48]) # this will be exon 3
CCC
>>> s2 = DNA.make_seq("CGAAACGTTT", name="seq2")
>>> s3 = DNA.make_seq("CGAAACGTTT", name="seq3")
```

Via

add_annotation

```
>>> from cogent3 import DNA
>>> from cogent3.core.annotation import Feature
>>> s1 = DNA.make_seq("AAGAAGAAGACCCCCAAAAAAAAAA"\
...                   "TTTTTTTTTTAAAAAGGGAACCCT",
...                   name="seq1")
...
>>> exon1 = s1.add_annotation(Feature, 'exon', 'A', [(10,15)])
>>> exon2 = s1.add_annotation(Feature, 'exon', 'B', [(30,40)])
```

add_feature

```
>>> from cogent3 import DNA
>>> s1 = DNA.make_seq("AAGAAGAAGACCCCCAAAAAAAAAA"\
...                   "TTTTTTTTTTAAAAAGGGAACCCT",
...                   name="seq1")
...
>>> exon3 = s1.add_feature('exon', 'C', [(45, 48)])
```

There are other annotation types.

Adding as a series or item-wise

```
>>> from cogent3 import DNA
>>> s2 = DNA.make_seq("CGAAACGTTT", name="seq2")
>>> cpgs_series = s2.add_feature('cpgsite', 'cpg', [(0,2), (5,7)])
>>> s3 = DNA.make_seq("CGAAACGTTT", name="seq3")
>>> cpg1 = s3.add_feature('cpgsite', 'cpg', [(0,2)])
>>> cpg2 = s3.add_feature('cpgsite', 'cpg', [(5,7)])
```

Taking the union of annotations

Construct a pseudo-feature (`cds`) that's a union of other features (`exon1`, `exon2`, `exon3`).

```
>>> from cogent3 import DNA
>>> s1 = DNA.make_seq("AAGAAGAAGACCCCCAAAAAAAAAA"\
...                   "TTTTTTTTTTAAAAAGGGAACCCT",
...                   name="seq1")
...
>>> exon1 = s1.add_feature('exon', 'A', [(10,15)])
>>> exon2 = s1.add_feature('exon', 'B', [(30,40)])
>>> exon3 = s1.add_feature('exon', 'C', [(45, 48)])
>>> cds = s1.get_region_covering_all([exon1, exon2, exon3])
```

Getting annotation coordinates

These are useful for doing custom things, e.g. you could construct intron features using the below.

```
>>> cds.get_coordinates()
[(10, 15), (30, 40), (45, 48)]
```

Annotations have shadows

A shadow is a span representing everything but the annotation.

```
>>> not_cds = cds.get_shadow()
>>> not_cds
region "not exon" at [0:10, 15:30, 40:45, 48:49]/49
```

Compare to the coordinates of the original.

```
>>> cds
region "exon" at [10:15, 30:40, 45:48]/49
```

Adding to a sequence member of an alignment

The following annotation is directly applied onto the sequence and so is in ungapped sequence coordinates.

```
>>> from cogent3 import LoadSeqs
>>> aln1 = LoadSeqs(data=[['x','-AAACCCCCA'],
...                       ['y','TTTT--TTTT']], array_align=False)
>>> seq_exon = aln1.get_seq('x').add_feature('exon', 'A', [(3,8)])
```

Adding to an alignment

We add an annotation directly onto an alignment. In this example we add a `Variable` that can be displayed as a red line on a drawing. The resulting annotation (`red_data` here) is in **alignment coordinates**!

```
>>> from cogent3.core.annotation import Variable
>>> red_data = aln1.add_annotation(Variable, 'redline', 'align',
...               [((0,15),1),((15,30),2),((30,45),3)])
...
```

Slicing sequences and alignments by annotations

By a feature or coordinates returns same sequence span

```
>>> from cogent3 import DNA
>>> s1 = DNA.make_seq("AAGAAGAAGACCCCCAAAAAAAAAA"\
...                   "TTTTTTTTTTAAAAAGGGAACCCT",
...                   name="seq1")
...
>>> exon1 = s1.add_feature('exon', 'A', [(10,15)])
>>> exon2 = s1.add_feature('exon', 'B', [(30,40)])
>>> s1[exon1]
DnaSequence(CCCCC)
>>> s1[10:15]
DnaSequence(CCCCC)
```

Using the annotation object `get_slice` method returns the same thing.

```
>>> s1[exon2]
DnaSequence(TTTTTAAAAA)
>>> exon2.get_slice()
DnaSequence(TTTTTAAAAA)
```

Slicing by pseudo-feature or feature series

```
>>> from cogent3 import DNA
>>> s1 = DNA.make_seq("AAGAAGAAGACCCCCAAAAAAAAAA"\
...                   "TTTTTTTTTTAAAAAGGGAACCCT",
...                   name="seq1")
...
>>> exon1 = s1.add_feature('exon', 'A', [(10,15)])
>>> exon2 = s1.add_feature('exon', 'B', [(30,40)])
>>> exon3 = s1.add_feature('exon', 'C', [(45, 48)])
>>> cds = s1.get_region_covering_all([exon1, exon2, exon3])
>>> print(s1[cds])
CCCCCTTTTTAAAAACCC
>>> print(s1[exon1, exon2, exon3])
CCCCCTTTTTAAAAACCC
```

Warning: Slices are applied in order!

```
>>> print(s1)
AAGAAGAAGACCCCCAAAAAAAAAATTTTTTTTTTAAAAAGGGAACCCT
>>> print(s1[exon1, exon2, exon3])
CCCCCTTTTTAAAAACCC
>>> print(s1[exon2])
TTTTTAAAAA
>>> print(s1[exon3])
CCC
>>> print(s1[exon1, exon3, exon2])
CCCCCCCCTTTTTAAAAA
```

Slice series must not be overlapping

```
>>> s1[1:10, 9:15]
Traceback (most recent call last):
ValueError: Uninvertable. Overlap: 9 < 10
>>> s1[exon1, exon1]
Traceback (most recent call last):
ValueError: Uninvertable. Overlap: 10 < 15
```

But `get_region_covering_all` resolves this, ensuring no overlaps.

```
>>> print(s1.get_region_covering_all([exon3, exon3]).get_slice())
CCC
```

You can slice an annotation itself

```
>>> print(s1[exon2])
TTTTTAAAAA
>>> ex2_start = exon2[0:3]
>>> print(s1[ex2_start])
TTT
>>> ex2_end = exon2[-3:]
>>> print(s1[ex2_end])
AAA
```

Sequence vs Alignment slicing

You can't slice an alignment using an annotation from a sequence.

```
>>> aln1[seq_exon]
Traceback (most recent call last):
ValueError: Can't map exon "A" at [3:8]/9 onto 2 x 10 text alignment: x[-AAACCCCCA],␣
↪y[TTTT--TTTT] via []
```

Copying annotations

You can copy annotations onto sequences with the same name, even if the length differs

```
>>> aln2 = LoadSeqs(data=[['x', '-AAAAAAAAA'], ['y', 'TTTT--TTTT']],
...                 array_align=False)
>>> seq = DNA.make_seq('CCCCCCCCCCCCCCCCCCCC', 'x')
>>> match_exon = seq.add_feature('exon', 'A', [(3,8)])
>>> aln2.get_seq('x').copy_annotations(seq)
>>> copied = list(aln2.get_annotations_from_seq('x', 'exon'))
>>> copied
[exon "A" at [4:9]/10]
```

but if the feature lies outside the sequence being copied to, you get a lost span

```
>>> aln2 = LoadSeqs(data=[['x', '-AAAA'], ['y', 'TTTTT']], array_align=False)
>>> seq = DNA.make_seq('CCCCCCCCCCCCCCCCCCCC', 'x')
>>> match_exon = seq.add_feature('exon', 'A', [(5,8)])
>>> aln2.get_seq('x').copy_annotations(seq)
>>> copied = list(aln2.get_annotations_from_seq('x', 'exon'))
>>> copied
[exon "A" at [5:5, -4-]/5]
>>> copied[0].get_slice()
2 x 4 text alignment: x[----], y[----]
```

You can copy to a sequence with a different name, in a different alignment if the feature lies within the length

```
>>> # new test
>>> aln2 = LoadSeqs(data=[['x', '-AAAAAAAAA'], ['y', 'TTTT--TTTT']],
...                 array_align=False)
>>> seq = DNA.make_seq('CCCCCCCCCCCCCCCCCCCC', 'x')
>>> match_exon = seq.add_feature('exon', 'A', [(5,8)])
>>> aln2.get_seq('y').copy_annotations(seq)
>>> copied = list(aln2.get_annotations_from_seq('y', 'exon'))
```

```
>>> copied
[exon "A" at [7:10]/10]
```

If the sequence is shorter, again you get a lost span.

```
>>> aln2 = LoadSeqs(data=[['x', '-AAAAAAAAA'], ['y', 'TTTT--TTTT']],
...                 array_align=False)
>>> diff_len_seq = DNA.make_seq('CCCCCCCCCCCCCCCCCCCCCCCCCC', 'x')
>>> nonmatch = diff_len_seq.add_feature('repeat', 'A', [(12,14)])
>>> aln2.get_seq('y').copy_annotations(diff_len_seq)
>>> copied = list(aln2.get_annotations_from_seq('y', 'repeat'))
>>> copied
[repeat "A" at [10:10, -6-]/10]
```

Querying

You need to get a corresponding annotation projected into alignment coordinates via a query.

```
>>> aln_exon = aln1.get_annotations_from_any_seq('exon')
>>> print(aln1[aln_exon])
>x
CCCCC
>y
--TTT
```

Querying produces objects only valid for their source

```
>>> cpgsite2 = s2.get_annotations_matching('cpgsite')
>>> print(s2[cpgsite2])
CGCG
>>> cpgsite3 = s3.get_annotations_matching('cpgsite')
>>> s2[cpgsite3]
Traceback (most recent call last):
ValueError: Can't map cpgsite "cpg" at [0:2]/10 onto DnaSequence(CGAAACGTTT) via []
```

Querying for absent annotation

You get back an empty list, and slicing with this returns an empty sequence.

```
>>> # this test is new
>>> dont_exist = s2.get_annotations_matching('dont_exist')
>>> dont_exist
[]
>>> s2[dont_exist]
DnaSequence()
```

Querying features that span gaps in alignments

If you query for a feature from a sequence, it's alignment coordinates may be discontinuous.

```
>>> aln3 = LoadSeqs(data=[['x', 'C-CCCAAAAA'], ['y', '-T----TTTT']],
...                 array_align=False)
>>> exon = aln3.get_seq('x').add_feature('exon', 'ex1', [(0,4)])
>>> print(exon.get_slice())
CCCC
>>> aln_exons = list(aln3.get_annotations_from_seq('x', 'exon'))
>>> print(aln_exons)
[exon "ex1" at [0:1, 2:5]/10]
>>> print(aln3[aln_exons])
>x
CCCC
>y
----
```

Note: The `T` opposite the gap is missing since this approach only returns positions directly corresponding to the feature.

`as_one_span` unifies features with discontinuous alignment coordinates

To get positions spanned by a feature, including gaps, use `as_one_span`.

```
>>> unified = aln_exons[0].as_one_span()
>>> print(aln3[unified])
>x
C-CCC
>y
-T---
```

Behaviour of annotations on nucleic acid sequences

Reverse complementing a sequence **does not** reverse annotations, that is they retain the reference to the frame for which they were defined.

```
>>> plus = DNA.make_seq("CCCCCAAAAAAAAAATTTTTTTTTTAAAGG")
>>> plus_rpt = plus.add_feature('blah', 'a', [(5,15), (25, 28)])
>>> print(plus[plus_rpt])
AAAAAAAAAAAAA
>>> minus = plus.rc()
>>> print(minus)
CCTTTAAAAAAAAAATTTTTTTTTGGGGG
>>> minus_rpt = minus.get_annotations_matching('blah')
>>> print(minus[minus_rpt])
AAAAAAAAAAAAA
```

Masking annotated regions

We mask the CDS regions.

```
>>> from cogent3.parse.genbank import RichGenbankParser
>>> parser = RichGenbankParser(open('data/ST_genome_part.gb'))
```

```
>>> seq = [seq for accession, seq in parser][0]
>>> no_cds = seq.with_masked_annotations('CDS')
>>> print(no_cds[150:400])
CAAGACAGACAAATAAAAATGACAGAGTACACAACATCC?????????...
```

The above sequence could then have positions filtered so no position with the ambiguous character '?' was present.

Masking annotated regions on alignments

We mask exon's on an alignment.

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs(data=[['x', 'C-CCCAAAAAGGGAA'],
...                      ['y', '-T----TTTTG-GTT']],
...                moltype=DNA, array_align=False)
>>> exon = aln.get_seq('x').add_feature('exon', 'norwegian', [(0,4)])
>>> print(aln.with_masked_annotations('exon', mask_char='?'))
>x
?-???AAAAAGGGAA
>y
-T----TTTTG-GTT
```

These also persist through reverse complement operations.

```
>>> rc = aln.rc()
>>> print(rc)
>x
TTCCCTTTTTGGG-G
>y
AAC-CAAAA----A-

>>> print(rc.with_masked_annotations('exon', mask_char='?'))
>x
TTCCCTTTTT???-?
>y
AAC-CAAAA----A-
```

You can take mask of the shadow

```
>>> from cogent3 import DNA
>>> s = DNA.make_seq('CCCCAAAAAGGGAA', 'x')
>>> exon = s.add_feature('exon', 'norwegian', [(0,4)])
>>> rpt = s.add_feature('repeat', 'norwegian', [(9, 12)])
>>> rc = s.rc()
>>> print(s.with_masked_annotations('exon', shadow=True))
CCCC??????????
>>> print(rc.with_masked_annotations('exon', shadow=True))
??????????GGGG
>>> print(s.with_masked_annotations(['exon', 'repeat'], shadow=True))
CCCC?????GGG??
>>> print(rc.with_masked_annotations(['exon', 'repeat'], shadow=True))
??CCC?????GGGG
```

What features of a certain type are available?

```
>>> from cogent3 import DNA
>>> s = DNA.make_seq('ATGACCCTGTAAAAAATGTGTTAACCC',
...     name='a')
>>> cds1 = s.add_feature('cds','cds1', [(0,12)])
>>> cds2 = s.add_feature('cds','cds2', [(15,24)])
>>> all_cds = s.get_annotations_matching('cds')
>>> all_cds
[cds "cds1" at [0:12]/27, cds "cds2" at [15:24]/27]
```

Getting all features of a type, or everything but that type

The annotation methods `get_region_covering_all` and `get_shadow` can be used to grab all the coding sequences or non-coding sequences in a `DnaSequence` object.

```
>>> from cogent3.parse.genbank import RichGenbankParser
>>> parser = RichGenbankParser(open('data/ST_genome_part.gb'))
>>> seq = [seq for accession, seq in parser][0]
>>> all_cds = seq.get_annotations_matching('CDS')
>>> coding_seqs = seq.get_region_covering_all(all_cds)
>>> coding_seqs
region "CDS" at [189:255, 336:2799, 2800:3730, 3733...
>>> coding_seqs.get_slice()
DnaSequence(ATGAACC... 9063)
>>> noncoding_seqs = coding_seqs.get_shadow()
>>> noncoding_seqs
region "not CDS" at [0:189, 255:336, 2799:2800, ...
>>> noncoding_seqs.get_slice()
DnaSequence(AGAGATT... 957)
```

Getting sequence features when you have an alignment object

Sequence features can be accessed via a containing `Alignment`.

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs(data=[['x','-AAAAAAAAA'], ['y','TTTT--TTTT']],
...                array_align=False)
>>> print(aln)
>x
-AAAAAAAAA
>y
TTTT--TTTT

>>> exon = aln.get_seq('x').add_feature('exon', '1', [(3,8)])
>>> aln_exons = aln.get_annotations_from_seq('x', 'exon')
>>> aln_exons = aln.get_annotations_from_any_seq('exon')
>>> aln_exons
[exon "1" at [4:9]/10]
```

Annotation display on sequences

We can display annotations on sequences, writing to file.

Note: This requires matplotlib be installed.

We first make a sequence and add some annotations.

```
>>> from cogent3 import DNA
>>> seq = DNA.make_seq('aaaccggttt' * 10)
>>> v = seq.add_feature('exon', 'exon', [(20,35)])
>>> v = seq.add_feature('repeat_unit', 'repeat_unit', [(39,49)])
>>> v = seq.add_feature('repeat_unit', 'rep2', [(49,60)])
```

We then make a `Display` instance and write to file. This will use standard feature policy for colouring and shape of feature types.

```
>>> from cogent3.draw.linear import Display
>>> seq_display = Display(seq, colour_sequences=True)
>>> fig = seq_display.make_figure()
>>> fig.savefig('annotated_1.png')
```

Annotation display on alignments

```
>>> from cogent3 import DNA, LoadSeqs
>>> from cogent3.core.annotation import Variable
>>> from cogent3.draw.linear import Display
>>> aln = LoadSeqs('data/primate_cdx2_promoter.fasta', moltype=DNA,
...                array_align=False)[:150]
>>> annot = aln.add_annotation(Variable, 'redline', 'align',
...                            [((0,15),1),((15,30),2),((30,45),3)])
>>> annot = aln.add_annotation(Variable, 'blueline', 'align',
...                            [((0,15),1.5),((15,30),2.5),((30,45),3.5)])
>>> align_display = Display(aln, colour_sequences=True)
>>> fig = align_display.make_figure(width=25, left=1, right=1)
>>> fig.savefig('annotated_2.png')
```

Annotation display of a custom variable

We just show a series of spans.

```
>>> from cogent3 import DNA
>>> from cogent3.draw.linear import Display
>>> from cogent3.core.annotation import Variable
>>> seq = DNA.make_seq('aaaccggttt' * 10)
>>> annot = seq.add_annotation(Variable, 'redline', 'align',
...     [((0,15),1),((15,30),2),((30,45),3)])
...
>>> seq_display = Display(seq, colour_sequences=True)
>>> fig = seq_display.make_figure()
>>> fig.savefig('annotated_3.png')
```

Generic metadata

To be written.

Info object

To be written.

Genetic code

Getting a genetic code

The standard genetic code.

```
>>> from cogent3.core.genetic_code import GeneticCodes
>>> standard_code = GeneticCodes[1]
```

The vertebrate mt genetic code.

```
>>> from cogent3.core.genetic_code import GeneticCodes
>>> mt_gc = GeneticCodes[2]
>>> print(mt_gc.name)
Vertebrate Mitochondrial
```

To see the key -> genetic code mapping, use a loop.

```
>>> for key, code in GeneticCodes.items():
...     print(key, code.name)
1 Standard Nuclear
2 Vertebrate Mitochondrial
3 Yeast Mitochondrial...
```

Translate DNA sequences

```
>>> from cogent3.core.genetic_code import DEFAULT as standard_code
>>> standard_code.translate('TTTGCAAAC')
'FAN'
```

Conversion to a `ProteinSequence` from a `DnaSequence` is shown in *Translate a DnaSequence to protein*.

Translate all six frames

```
>>> from cogent3 import DNA
>>> from cogent3.core.genetic_code import DEFAULT as standard_code
>>> seq = DNA.make_seq('ATGCTAACATAAA')
>>> translations = standard_code.sixframes(seq)
>>> print(translations)
['MLT*', 'C*HK', 'ANI', 'FMLA', 'LC*H', 'YVS']
```

Find out how many stops in a frame

```
>>> from cogent3 import DNA
>>> from cogent3.core.genetic_code import DEFAULT as standard_code
>>> seq = DNA.make_seq('ATGCTAACATAAA')
>>> stops_frame1 = standard_code.get_stop_indices(seq, start=0)
```

```
>>> stops_frame1
[9]
>>> stop_index = stops_frame1[0]
>>> seq[stop_index:stop_index+3]
DnaSequence(TAA)
```

Translate a codon

```
>>> from cogent3.core.genetic_code import DEFAULT as standard_code
>>> standard_code['TTT']
'F'
```

or get the codons for a single amino acid

```
>>> standard_code['A']
['GCT', 'GCC', 'GCA', 'GCG']
```

Look up the amino acid corresponding to a single codon

```
>>> from cogent3.core.genetic_code import DEFAULT as standard_code
>>> standard_code['TTT']
'F'
```

Or get all the codons for one amino acid

```
>>> standard_code['A']
['GCT', 'GCC', 'GCA', 'GCG']
```

For a group of amino acids

```
>>> targets = ['A','C']
>>> codons = [standard_code[aa] for aa in targets]
>>> codons
[['GCT', 'GCC', 'GCA', 'GCG'], ['TGT', 'TGC']]
>>> flat_list = sum(codons,[])
>>> flat_list
['GCT', 'GCC', 'GCA', 'GCG', 'TGT', 'TGC']
```

Converting the `CodonAlphabet` to codon series

```
>>> from cogent3 import DNA
>>> my_seq = DNA.make_seq("AGTACACTGGTT")
>>> sorted(my_seq.codon_alphabet())
['AAA', 'AAC', 'AAG', 'AAT'...
>>> len(my_seq.codon_alphabet())
61
```

Obtaining the codons from a `DnaSequence` object

Use the method `get_in_motif_size`

```
>>> from cogent3 import DNA
>>> my_seq = DNA.make_seq('ATGCACTGGTAA','my_gene')
>>> codons = my_seq.get_in_motif_size(3)
>>> print(codons)
['ATG', 'CAC', 'TGG', 'TAA']
```

You can't translate a sequence that contains a stop codon.

```
>>> pep = my_seq.get_translation()
Traceback (most recent call last):
AlphabetError: TAA
```

Remove the stop codon first

```
>>> from cogent3 import DNA
>>> my_seq = DNA.make_seq('ATGCACTGGTAA','my_gene')
>>> seq = my_seq.trim_stop_codon()
>>> pep = seq.get_translation()
>>> print(pep.to_fasta())
>my_gene
MHW
>>> print(type(pep))
<class 'cogent3.core.sequence.ProteinSequence'>
```

Or we can just grab the correct slice from the `DnaSequence` object

```
>>> from cogent3 import DNA
>>> my_seq = DNA.make_seq('CAAATGTATTAA','my_gene')
>>> pep = my_seq[:-3].get_translation().to_fasta()
>>> print(pep)
>my_gene
QMY
```

Trees

Trees

Basics

Loading a tree from a file and visualizing it with `ascii_art()`

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> print(tr.ascii_art())
                              /-Human
                    /edge.0--|
```

```
          /edge.1--|          \-HowlerMon
         |         |
         |          \-Mouse
-root----|
         |--NineBande
         |
          \-DogFaced
```

Writing a tree to a file

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> tr.write('data/temp.tree')
```

Getting the individual nodes of a tree by name

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> names = tr.get_node_names()
>>> names[:4]
['root', 'edge.1', 'edge.0', 'Human']
>>> names[4:]
['HowlerMon', 'Mouse', 'NineBande', 'DogFaced']
>>> names_nodes = tr.get_nodes_dict()
>>> names_nodes['Human']
Tree("Human;")
>>> tr.get_node_matching_name('Mouse')
Tree("Mouse;")
```

Getting the name of a node (or a tree)

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> hu = tr.get_node_matching_name('Human')
>>> tr.name
'root'
>>> hu.name
'Human'
```

The object type of a tree and its nodes is the same

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> nodes = tr.get_nodes_dict()
>>> hu = nodes['Human']
>>> type(hu)
<class 'cogent3.core.tree.PhyloNode'>
>>> type(tr)
<class 'cogent3.core.tree.PhyloNode'>
```

Working with the nodes of a tree

Get all the nodes, tips and edges

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> nodes = tr.get_nodes_dict()
>>> for n in nodes.items():
...     print(n)
...
('NineBande', Tree("NineBande;"))
('edge.1', Tree("((Human,HowlerMon),Mouse);"))
('root', Tree("(((Human,HowlerMon),Mouse),NineBande,DogFaced);"))
('DogFaced', Tree("DogFaced;"))
('Human', Tree("Human;"))
('edge.0', Tree("(Human,HowlerMon);"))
('Mouse', Tree("Mouse;"))
('HowlerMon', Tree("HowlerMon;"))
```

only the terminal nodes (tips)

```
>>> for n in tr.iter_tips():
...     print(n)
...
Human:0.0311054096183;
HowlerMon:0.0415847131449;
Mouse:0.277353608988;
NineBande:0.0939768158209;
DogFaced:0.113211053859;
```

for internal nodes (edges) we can use Newick format to simplify the output

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> for n in tr.iter_nontips():
...     print(n.get_newick())
...
((Human,HowlerMon),Mouse);
(Human,HowlerMon);
```

Getting the path between two tips or edges (connecting edges)

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> edges = tr.get_connecting_edges('edge.1','Human')
>>> for edge in edges:
...    print(edge.name)
...
edge.1
edge.0
Human
```

Getting the distance between two nodes

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> nodes = tr.get_nodes_dict()
>>> hu = nodes['Human']
>>> mu = nodes['Mouse']
>>> hu.distance(mu)
0.3467553...
>>> hu.is_tip()
True
```

Getting the last common ancestor (LCA) for two nodes

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> nodes = tr.get_nodes_dict()
>>> hu = nodes['Human']
>>> mu = nodes['Mouse']
>>> lca = hu.last_common_ancestor(mu)
>>> lca
Tree("((Human,HowlerMon),Mouse);")
>>> type(lca)
<class 'cogent3.core.tree.PhyloNode'>
```

Getting all the ancestors for a node

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> hu = tr.get_node_matching_name('Human')
>>> for a in hu.ancestors():
...     print(a.name)
...
edge.0
edge.1
root
```

Getting all the children for a node

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> node = tr.get_node_matching_name('edge.1')
>>> children = list(node.iter_tips()) + list(node.iter_nontips())
>>> for child in children:
...     print(child.name)
...
Human
HowlerMon
Mouse
edge.0
```

Getting all the distances for a tree

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> dists = tr.get_distances()
```

We also show how to select a subset of distances involving just one species.

```
>>> human_dists = [names for names in dists if 'Human' in names]
>>> for dist in human_dists:
...     print(dist, dists[dist])
...
('Human', 'NineBande') 0.183106418165
('DogFaced', 'Human') 0.202340656203
('NineBande', 'Human') 0.183106418165
('Human', 'DogFaced') 0.202340656203
('Mouse', 'Human') 0.346755361094
('HowlerMon', 'Human') 0.0726901227632
('Human', 'Mouse') 0.346755361094
('Human', 'HowlerMon') 0.0726901227632
```

Getting the two nodes that are farthest apart

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> tr.max_tip_tip_distance()
(0.4102925130849, ('Mouse', 'DogFaced'))
```

Get the nodes within a given distance

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> hu = tr.get_node_matching_name('Human')
>>> tips = hu.tips_within_distance(0.2)
>>> for t in tips:
...     print(t)
...
HowlerMon:0.0415847131449;
NineBande:0.0939768158209;
```

Rerooting trees

At a named node

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> print(tr.rooted_at('edge.0').ascii_art())
          /-Human
         |
-root----|--HowlerMon
```

```
          |
          |           /-Mouse
           \edge.0--|
                     |           /-NineBande
                      \edge.1--|
                                 \-DogFaced
```

At the midpoint

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> print(tr.root_at_midpoint().ascii_art())
           /-Mouse
          |
-root----|                     /-Human
          |           /edge.0--|
          |          |          \-HowlerMon
           \edge.0.2|
                     |          /-NineBande
                      \edge.1--|
                                \-DogFaced
>>> print(tr.ascii_art())
                                /-Human
                     /edge.0--|
           /edge.1--|          \-HowlerMon
          |          |
          |           \-------- /-Mouse
-root----|
          |--NineBande
          |
           \-DogFaced
```

Near a given tip

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> print(tr.ascii_art())
                                /-Human
                     /edge.0--|
           /edge.1--|          \-HowlerMon
          |          |
          |           \-Mouse
-root----|
          |--NineBande
          |
           \-DogFaced
>>> print(tr.rooted_with_tip("Mouse").ascii_art())
                     /-Human
           /edge.0--|
          |          \-HowlerMon
          |
-root----|--Mouse
          |
          |           /-NineBande
```

```
          \edge.1--|
                   \-DogFaced
```

Tree representations

Newick format

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> tr.get_newick()
'(((Human,HowlerMon),Mouse),NineBande,DogFaced);'
>>> tr.get_newick(with_distances=True)
'(((Human:0.0311054096183,HowlerMon:0.0415847131449)...
```

XML format

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> xml = tr.get_xml()
>>> for line in xml.splitlines():
...     print(line)
...
<?xml version="1.0"?>
<clade>
  <clade>
     <param><name>length</name><value>0.0197278502379</value></param>
    <clade>
       <param><name>length</name><value>0.0382963424874</value></param>
      <clade>
         <name>Human</name>...
```

Write to PDF

Note: This requires `matplotlib`. It will bring up a `matplotlib` window if run from the command line. But in any case, it will write the pdf file to the data directory.

```
>>> from cogent3 import LoadTree
>>> from cogent3.draw import dendrogram
>>> tr = LoadTree('data/test.tree')
>>> h, w = 500, 500
>>> np = dendrogram.ContemporaneousDendrogram(tr)
>>> np.write_pdf('temp.pdf', w, h, font_size=14)
```

Tree traversal

Here is the example tree for reference:

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> print(tr.ascii_art())
                              /-Human
                    /edge.0--|
          /edge.1--|          \-HowlerMon
         |         |
         |          \-Mouse
-root----|
         |--NineBande
         |
          \-DogFaced
```

Preorder

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> for t in tr.preorder():
...     print(t.get_newick())
...
(((Human,HowlerMon),Mouse),NineBande,DogFaced);
((Human,HowlerMon),Mouse);
(Human,HowlerMon);
Human;
HowlerMon;
Mouse;
NineBande;
DogFaced;
```

Postorder

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> for t in tr.postorder():
...     print(t.get_newick())
...
Human;
HowlerMon;
(Human,HowlerMon);
Mouse;
((Human,HowlerMon),Mouse);
NineBande;
DogFaced;
(((Human,HowlerMon),Mouse),NineBande,DogFaced);
```

Selecting subtrees

One way to do it

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
```

```
>>> for tip in tr.iter_nontips():
...     tip_names = tip.get_tip_names()
...     print(tip_names)
...     sub_tree = tr.get_sub_tree(tip_names)
...     print(sub_tree.ascii_art())
...     print
...
['Human', 'HowlerMon', 'Mouse']
          /-Human
         |
-root----|--HowlerMon
         |
          \-Mouse

['Human', 'HowlerMon']
          /-Human
-root----|
          \-HowlerMon
```

Tree manipulation methods

Pruning the tree

Remove internal nodes with only one child. Create new connections and branch lengths (if tree is a PhyloNode) to reflect the change.

```
>>> from cogent3 import LoadTree
>>> simple_tree_string="(B:0.2,(D:0.4)E:0.5)F;"
>>> simple_tree=LoadTree(treestring=simple_tree_string)
>>> print(simple_tree.ascii_art())
          /-B
-F-------|
          \E------- /-D
>>> simple_tree.prune()
>>> print(simple_tree.ascii_art())
          /-B
-F-------|
          \-D
>>> print(simple_tree)
(B:0.2,D:0.9)F;
```

Create a full unrooted copy of the tree

```
>>> from cogent3 import LoadTree
>>> tr1 = LoadTree('data/test.tree')
>>> print(tr1.get_newick())
(((Human,HowlerMon),Mouse),NineBande,DogFaced);
>>> tr2 = tr1.unrooted_deepcopy()
>>> print(tr2.get_newick())
(((Human,HowlerMon),Mouse),NineBande,DogFaced);
```

Transform tree into a bifurcating tree

Add internal nodes so that every node has 2 or fewer children.

```
>>> from cogent3 import LoadTree
>>> tree_string="(B:0.2,H:0.2,(C:0.3,D:0.4,E:0.1)F:0.5)G;"
>>> tr = LoadTree(treestring=tree_string)
>>> print(tr.ascii_art())
          /-B
         |
         |--H
-G-------|
         |          /-C
         |         |
          \F-------|--D
                   |
                    \-E
>>> print(tr.bifurcating().ascii_art())
          /-B
-G-------|
         |          /-H
          \--------|
                   |          /-C
                    \F-------|
                             |          /-D
                              \--------|
                                        \-E
```

Transform tree into a balanced tree

Using a balanced tree can substantially improve performance of likelihood calculations. Note that the resulting tree has a different orientation with the effect that specifying clades or stems for model parameterization should be done using the "outgroup_name" argument.

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> print(tr.ascii_art())
                             /-Human
                   /edge.0--|
          /edge.1--|          \-HowlerMon
         |         |
         |          \-Mouse
-root----|
         |--NineBande
         |
          \-DogFaced
>>> print(tr.balanced().ascii_art())
                   /-Human
          /edge.0--|
         |         \-HowlerMon
         |
-root----|--Mouse
         |
         |          /-NineBande
          \edge.1--|
                    \-DogFaced
```

Test two trees for same topology

Branch lengths don't matter.

```
>>> from cogent3 import LoadTree
>>> tr1 = LoadTree(treestring="(B:0.2,(C:0.2,D:0.2)F:0.2)G;")
>>> tr2 = LoadTree(treestring="((C:0.1,D:0.1)F:0.1,B:0.1)G;")
>>> tr1.same_topology(tr2)
True
```

Calculate each node's maximum distance to a tip

Sets each node's "TipDistance" attribute to be the distance from that node to its most distant tip.

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree(treestring="(B:0.2,(C:0.3,D:0.4)F:0.5)G;")
>>> print(tr.ascii_art())
          /-B
-G-------|
         |          /-C
          \F-------|
                    \-D
>>> tr.set_tip_distances()
>>> for t in tr.preorder():
...     print(t.name, t.TipDistance)
...
G 0.9
B 0
F 0.4
C 0
D 0
```

Scale branch lengths in place to integers for ascii output

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree(treestring="(B:0.2,(C:0.3,D:0.4)F:0.5)G;")
>>> print(tr)
(B:0.2,(C:0.3,D:0.4)F:0.5)G;
>>> tr.scale_branch_lengths()
>>> print(tr)
(B:22,(C:33,D:44)F:56)G;
```

Get tip-to-tip distances

Get a distance matrix between all pairs of tips and a list of the tip nodes.

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree(treestring="(B:3,(C:2,D:4)F:5)G;")
>>> d,tips = tr.tip_to_tip_distances()
>>> for i,t in enumerate(tips):
...     print(t.name,d[i])
...
```

```
B [  0.  10.  12.]
C [ 10.   0.   6.]
D [ 12.   6.   0.]
```

Compare two trees using tip-to-tip distance matrices

Score ranges from 0 (minimum distance) to 1 (maximum distance). The default is to use Pearson's correlation, in which case a score of 0 means that the Pearson's correlation was perfectly good (1), and a score of 1 means that the Pearson's correlation was perfectly bad (-1).

Note: automatically strips out the names that don't match.

```
>>> from cogent3 import LoadTree
>>> tr1 = LoadTree(treestring="(B:2,(C:3,D:4)F:5)G;")
>>> tr2 = LoadTree(treestring="(C:2,(B:3,D:4)F:5)G;")
>>> tr1.compare_by_tip_distances(tr2)
0.0835...
```

Phylonodes Methods

Basics

Loading a tree from a file and visualizing it with `ascii_art()`

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> print(tr.ascii_art())
                            /-Human
                  /edge.0--|
         /edge.1--|         \-HowlerMon
        |         |
        |          \-Mouse
-root---|
        |--NineBande
        |
         \-DogFaced
```

Writing a tree to a file

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> tr.write('data/temp.tree')
```

Getting the individual nodes of a tree by name

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> names = tr.get_node_names()
>>> names[:4]
```

```
['root', 'edge.1', 'edge.0', 'Human']
>>> names[4:]
['HowlerMon', 'Mouse', 'NineBande', 'DogFaced']
>>> names_nodes = tr.get_nodes_dict()
>>> names_nodes['Human']
Tree("Human;")
>>> tr.get_node_matching_name('Mouse')
Tree("Mouse;")
```

Getting the name of a node (or a tree)

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> hu = tr.get_node_matching_name('Human')
>>> tr.name
'root'
>>> hu.name
'Human'
```

The object type of a tree and its nodes is the same

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> nodes = tr.get_nodes_dict()
>>> hu = nodes['Human']
>>> type(hu)
<class 'cogent3.core.tree.PhyloNode'>
>>> type(tr)
<class 'cogent3.core.tree.PhyloNode'>
```

Working with the nodes of a tree

Get all the nodes, tips and edges

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> nodes = tr.get_nodes_dict()
>>> for n in nodes.items():
...     print(n)
...
('NineBande', Tree("NineBande;"))
('edge.1', Tree("((Human,HowlerMon),Mouse);"))
('root', Tree("(((Human,HowlerMon),Mouse),NineBande,DogFaced);"))
('DogFaced', Tree("DogFaced;"))
('Human', Tree("Human;"))
('edge.0', Tree("(Human,HowlerMon);"))
('Mouse', Tree("Mouse;"))
('HowlerMon', Tree("HowlerMon;"))
```

only the terminal nodes (tips)

```
>>> for n in tr.iter_tips():
...     print(n)
...
Human:0.0311054096183;
HowlerMon:0.0415847131449;
Mouse:0.277353608988;
NineBande:0.0939768158209;
DogFaced:0.113211053859;
```

for internal nodes (edges) we can use Newick format to simplify the output

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> for n in tr.iter_nontips():
...     print(n.get_newick())
...
((Human,HowlerMon),Mouse);
(Human,HowlerMon);
```

Getting the path between two tips or edges (connecting edges)

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> edges = tr.get_connecting_edges('edge.1','Human')
>>> for edge in edges:
...     print(edge.name)
...
edge.1
edge.0
Human
```

Getting the distance between two nodes

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> nodes = tr.get_nodes_dict()
>>> hu = nodes['Human']
>>> mu = nodes['Mouse']
>>> hu.distance(mu)
0.3467553...
>>> hu.is_tip()
True
```

Getting the last common ancestor (LCA) for two nodes

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> nodes = tr.get_nodes_dict()
>>> hu = nodes['Human']
>>> mu = nodes['Mouse']
>>> lca = hu.last_common_ancestor(mu)
```

```
>>> lca
Tree("((Human,HowlerMon),Mouse);")
>>> type(lca)
<class 'cogent3.core.tree.PhyloNode'>
```

Getting all the ancestors for a node

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> hu = tr.get_node_matching_name('Human')
>>> for a in hu.ancestors():
...     print(a.name)
...
edge.0
edge.1
root
```

Getting all the children for a node

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> node = tr.get_node_matching_name('edge.1')
>>> children = list(node.iter_tips()) + list(node.iter_nontips())
>>> for child in children:
...     print(child.name)
...
Human
HowlerMon
Mouse
edge.0
```

Getting all the distances for a tree

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> dists = tr.get_distances()
```

We also show how to select a subset of distances involving just one species.

```
>>> human_dists = [names for names in dists if 'Human' in names]
>>> for dist in human_dists:
...     print(dist, dists[dist])
...
('Human', 'NineBande') 0.183106418165
('DogFaced', 'Human') 0.202340656203
('NineBande', 'Human') 0.183106418165
('Human', 'DogFaced') 0.202340656203
('Mouse', 'Human') 0.346755361094
('HowlerMon', 'Human') 0.0726901227632
('Human', 'Mouse') 0.346755361094
('Human', 'HowlerMon') 0.0726901227632
```

Getting the two nodes that are farthest apart

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> tr.max_tip_tip_distance()
(0.4102925130849, ('Mouse', 'DogFaced'))
```

Get the nodes within a given distance

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> hu = tr.get_node_matching_name('Human')
>>> tips = hu.tips_within_distance(0.2)
>>> for t in tips:
...     print(t)
...
HowlerMon:0.0415847131449;
NineBande:0.0939768158209;
```

Rerooting trees

At a named node

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> print(tr.rooted_at('edge.0').ascii_art())
          /-Human
         |
-root----|--HowlerMon
         |
         |          /-Mouse
          \edge.0--|
                   |          /-NineBande
                    \edge.1--|
                              \-DogFaced
```

At the midpoint

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> print(tr.root_at_midpoint().ascii_art())
          /-Mouse
         |
-root----|                    /-Human
         |          /edge.0--|
         |         |          \-HowlerMon
          \edge.0.2|
                   |          /-NineBande
                    \edge.1--|
                              \-DogFaced
>>> print(tr.ascii_art())
```

```
                              /-Human
                    /edge.0--|
          /edge.1--|          \-HowlerMon
         |         |
         |          \-------- /-Mouse
-root----|
         |--NineBande
         |
          \-DogFaced
```

Tree representations

Newick format

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> tr.get_newick()
'(((Human,HowlerMon),Mouse),NineBande,DogFaced);'
>>> tr.get_newick(with_distances=True)
'(((Human:0.0311054096183,HowlerMon:0.0415847131449)...
```

XML format

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> xml = tr.get_xml()
>>> for line in xml.splitlines():
...     print(line)
...
<?xml version="1.0"?>
<clade>
  <clade>
     <param><name>length</name><value>0.0197278502379</value></param>
    <clade>
       <param><name>length</name><value>0.0382963424874</value></param>
      <clade>
         <name>Human</name>...
```

Write to PDF

Note: This requires `matplotlib`. It will bring up a `matplotlib` window if run from the command line. But in any case, it will write the pdf file to the data directory.

```
>>> from cogent3 import LoadTree
>>> from cogent3.draw import dendrogram
>>> tr = LoadTree('data/test.tree')
>>> h, w = 500, 500
>>> np = dendrogram.ContemporaneousDendrogram(tr)
>>> np.write_pdf('temp.pdf', w, h, font_size=14)
```

Tree traversal

Here is the example tree for reference:

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> print(tr.ascii_art())
                              /-Human
                    /edge.0--|
          /edge.1--|          \-HowlerMon
         |         |
         |          \-Mouse
-root----|
         |--NineBande
         |
          \-DogFaced
```

Preorder

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> for t in tr.preorder():
...     print(t.get_newick())
...
(((Human,HowlerMon),Mouse),NineBande,DogFaced);
((Human,HowlerMon),Mouse);
(Human,HowlerMon);
Human;
HowlerMon;
Mouse;
NineBande;
DogFaced;
```

Postorder

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> for t in tr.postorder():
...     print(t.get_newick())
...
Human;
HowlerMon;
(Human,HowlerMon);
Mouse;
((Human,HowlerMon),Mouse);
NineBande;
DogFaced;
(((Human,HowlerMon),Mouse),NineBande,DogFaced);
```

Selecting subtrees

One way to do it

```
>>> from cogent3 import LoadTree
>>> tr = LoadTree('data/test.tree')
>>> for tip in tr.iter_nontips():
...     tip_names = tip.get_tip_names()
...     print(tip_names)
...     sub_tree = tr.get_sub_tree(tip_names)
...     print(sub_tree.ascii_art())
...     print()
...
['Human', 'HowlerMon', 'Mouse']
          /-Human
         |
-root----|--HowlerMon
         |
          \-Mouse

['Human', 'HowlerMon']
          /-Human
-root----|
          \-HowlerMon
```

Tables

Tabular data

Loading delimited formats

We load a comma separated data file using the generic `LoadTable` function.

```
>>> from cogent3 import LoadTable
>>> table = LoadTable('stats.txt', sep=',')
>>> print(table)
===================================
    Locus    Region            Ratio
-----------------------------------
NP_003077       Con           2.5386
NP_004893       Con      121351.4264
NP_005079       Con     9516594.9789
NP_005500    NonCon           0.0000
NP_055852    NonCon    10933217.7090
-----------------------------------
```

Reading large files

For really large files the automated conversion used by the standard read mechanism can be quite slow. If the data within a column is consistently of one type, set the `LoadTable` argument `static_column_types=True`. This causes the `Table` object to create a custom reader.

```
>>> table = LoadTable('stats.txt', static_column_types=True)
>>> print(table)
===================================
```

```
    Locus     Region             Ratio
-------------------------------------
NP_003077        Con            2.5386
NP_004893        Con       121351.4264
NP_005079        Con      9516594.9789
NP_005500     NonCon            0.0000
NP_055852     NonCon     10933217.7090
-------------------------------------
```

Formatting

Changing displayed numerical precision

We change the `Ratio` column to using scientific notation.

```
>>> table.format_column('Ratio', '%.1e')
>>> print(table)
==============================
    Locus     Region      Ratio
------------------------------
NP_003077        Con    2.5e+00
NP_004893        Con    1.2e+05
NP_005079        Con    9.5e+06
NP_005500     NonCon    7.4e-08
NP_055852     NonCon    1.1e+07
------------------------------
```

Change digits or column spacing

This can be done on table loading,

```
>>> table = LoadTable('stats.txt', sep=',', digits=1, space=2)
>>> print(table)
============================
    Locus  Region       Ratio
----------------------------
NP_003077     Con          2.5
NP_004893     Con     121351.4
NP_005079     Con    9516595.0
NP_005500  NonCon          0.0
NP_055852  NonCon   10933217.7
----------------------------
```

or, for spacing at least, by modifying the attributes

```
>>> table.space = '    '
>>> print(table)
================================
    Locus     Region          Ratio
--------------------------------
NP_003077        Con            2.5
NP_004893        Con       121351.4
NP_005079        Con      9516595.0
NP_005500     NonCon            0.0
```

```
NP_055852    NonCon    10933217.7
---------------------------------
```

Changing column headings

The table `header` is immutable. Changing column headings is done as follows.

```
>>> table = LoadTable('stats.txt', sep=',')
>>> print(table.header)
['Locus', 'Region', 'Ratio']
>>> table = table.with_new_header('Ratio', 'Stat')
>>> print(table.header)
['Locus', 'Region', 'Stat']
```

Creating new columns from existing ones

This can be used to take a single, or multiple columns and generate a new column of values. Here we'll take 2 columns and return True/False based on a condition.

```
>>> table = LoadTable('stats.txt', sep=',')
>>> table = table.with_new_column('LargeCon',
...                     lambda r_v: r_v[0] == 'Con' and r_v[1]>10.0,
...                     columns=['Region', 'Ratio'])
>>> print(table)
================================================
    Locus    Region            Ratio    LargeCon
------------------------------------------------
NP_003077       Con           2.5386       False
NP_004893       Con      121351.4264        True
NP_005079       Con     9516594.9789        True
NP_005500    NonCon           0.0000       False
NP_055852    NonCon    10933217.7090       False
------------------------------------------------
```

Appending tables

Can be done without specifying a new column. Here we simply use the same table data.

```
>>> table1 = LoadTable('stats.txt', sep=',')
>>> table2 = LoadTable('stats.txt', sep=',')
>>> table = table1.appended(None, table2)
>>> print(table)
====================================
    Locus    Region            Ratio
------------------------------------
NP_003077       Con           2.5386
NP_004893       Con      121351.4264
NP_005079       Con     9516594.9789
NP_005500    NonCon           0.0000
NP_055852    NonCon    10933217.7090
NP_003077       Con           2.5386
NP_004893       Con      121351.4264
```

```
NP_005079       Con     9516594.9789
NP_005500    NonCon           0.0000
NP_055852    NonCon    10933217.7090
--------------------------------------
```

or with a new column

```
>>> table1.title = 'Data1'
>>> table2.title = 'Data2'
>>> table = table1.appended('Data#', table2, title='')
>>> print(table)
============================================
Data#        Locus    Region            Ratio
--------------------------------------------
Data1    NP_003077       Con           2.5386
Data1    NP_004893       Con      121351.4264
Data1    NP_005079       Con     9516594.9789
Data1    NP_005500    NonCon           0.0000
Data1    NP_055852    NonCon    10933217.7090
Data2    NP_003077       Con           2.5386
Data2    NP_004893       Con      121351.4264
Data2    NP_005079       Con     9516594.9789
Data2    NP_005500    NonCon           0.0000
Data2    NP_055852    NonCon    10933217.7090
--------------------------------------------
```

Note: We assigned an empty string to `title`, otherwise the resulting table has the same `title` attribute as that of `table1`.

Summing a single column

```
>>> table = LoadTable('stats.txt', sep=',')
>>> table.summed('Ratio')
20571166.652847398
```

Summing multiple columns or rows - strictly numerical data

We define a strictly numerical table,

```
>>> all_numeric = LoadTable(header=['A', 'B', 'C'], rows=[range(3),
...                                 range(3,6), range(6,9), range(9,12)])
>>> print(all_numeric)
=============
A     B     C
-------------
0     1     2
3     4     5
6     7     8
9    10    11
-------------
```

and sum all columns (default condition)

```
>>> all_numeric.summed()
[18, 22, 26]
```

and all rows

```
>>> all_numeric.summed(col_sum=False)
[3, 12, 21, 30]
```

Summing multiple columns or rows with mixed non-numeric/numeric data

We define a table with mixed data, like a distance matrix.

```
>>> mixed = LoadTable(header=['A', 'B', 'C'], rows=[['*',1,2], [3,'*', 5],
...                                                 [6,7,'*']])
>>> print(mixed)
===========
A    B    C
-----------
*    1    2
3    *    5
6    7    *
-----------
```

and sum all columns (default condition), ignoring non-numerical data

```
>>> mixed.summed(strict=False)
[9, 8, 7]
```

and all rows

```
>>> mixed.summed(col_sum=False, strict=False)
[3, 8, 13]
```

Filtering table rows

We can do this by providing a reference to an external function

```
>>> table = LoadTable('stats.txt', sep=',')
>>> sub_table = table.filtered(lambda x: x < 10.0, columns='Ratio')
>>> print(sub_table)
============================
    Locus    Region     Ratio
----------------------------
NP_003077       Con    2.5386
NP_005500    NonCon    0.0000
----------------------------
```

or using valid python syntax within a string, which is executed

```
>>> sub_table = table.filtered("Ratio < 10.0")
>>> print(sub_table)
============================
    Locus    Region     Ratio
----------------------------
```

```
NP_003077       Con    2.5386
NP_005500    NonCon    0.0000
------------------------------
```

You can also filter for values in multiple columns

```
>>> sub_table = table.filtered("Ratio < 10.0 and Region == 'NonCon'")
>>> print(sub_table)
==============================
    Locus    Region     Ratio
------------------------------
NP_005500    NonCon    0.0000
------------------------------
```

Filtering table columns

We select only columns that have a sum > 20 from the `all_numeric` table constructed above.

```
>>> big_numeric = all_numeric.filtered_by_column(lambda x: sum(x)>20)
>>> print(big_numeric)
========
 B      C
--------
 1      2
 4      5
 7      8
10     11
--------
```

Sorting

Standard sorting

```
>>> table = LoadTable('stats.txt', sep=',')
>>> print(table.sorted(columns='Ratio'))
====================================
    Locus    Region            Ratio
------------------------------------
NP_005500    NonCon           0.0000
NP_003077       Con           2.5386
NP_004893       Con      121351.4264
NP_005079       Con     9516594.9789
NP_055852    NonCon    10933217.7090
------------------------------------
```

Reverse sorting

```
>>> print(table.sorted(columns='Ratio', reverse='Ratio'))
====================================
    Locus    Region            Ratio
------------------------------------
```

```
NP_055852    NonCon    10933217.7090
NP_005079       Con     9516594.9789
NP_004893       Con      121351.4264
NP_003077       Con           2.5386
NP_005500    NonCon           0.0000
-----------------------------------
```

Sorting involving multiple columns, one reversed

```
>>> print(table.sorted(columns=['Region', 'Ratio'], reverse='Ratio'))
===================================
    Locus    Region            Ratio
-----------------------------------
NP_005079       Con     9516594.9789
NP_004893       Con      121351.4264
NP_003077       Con           2.5386
NP_055852    NonCon    10933217.7090
NP_005500    NonCon           0.0000
-----------------------------------
```

Getting raw data

For a single column

```
>>> table = LoadTable('stats.txt', sep=',')
>>> raw = table.tolist('Region')
>>> print(raw)
['Con', 'Con', 'Con', 'NonCon', 'NonCon']
```

For multiple columns

```
>>> table = LoadTable('stats.txt', sep=',')
>>> raw = table.tolist(['Locus', 'Region'])
>>> print(raw)
[['NP_003077', 'Con'], ['NP_004893', 'Con'], ...
```

Iterating over table rows

```
>>> table = LoadTable('stats.txt', sep=',')
>>> for row in table:
...     print(row['Locus'])
...
NP_003077
NP_004893
NP_005079
NP_005500
NP_055852
```

Table slicing

Using column names

```
>>> table = LoadTable('stats.txt', sep=',')
>>> print(table[:2, :'Region'])
=========
    Locus
---------
NP_003077
NP_004893
---------
```

Using column indices

```
>>> table = LoadTable('stats.txt', sep=',')
>>> print(table[:2,: 1])
=========
    Locus
---------
NP_003077
NP_004893
---------
```

SQL-like capabilities

Distinct values

```
>>> table = LoadTable('stats.txt', sep=',')
>>> assert table.distinct_values('Region') == set(['NonCon', 'Con'])
```

Counting

```
>>> table = LoadTable('stats.txt', sep=',')
>>> assert table.count("Region == 'NonCon' and Ratio > 1") == 1
```

Joining tables

SQL-like join operations requires tables have different `title` attributes which are not `None`. We do a standard inner join here for a restricted subset. We must specify the columns that will be used for the join. Here we just use `Locus` but multiple columns can be used, and their names can be different between the tables. Note that the second table's title becomes a part of the column names.

```
>>> rows = [['NP_004893', True], ['NP_005079', True],
...         ['NP_005500', False], ['NP_055852', False]]
>>> region_type = LoadTable(header=['Locus', 'LargeCon'], rows=rows,
...                 title='RegionClass')
>>> stats_table = LoadTable('stats.txt', sep=',', title='Stats')
```

```
>>> new = stats_table.joined(region_type, columns_self='Locus')
>>> print(new)
=============================================================
    Locus    Region            Ratio    RegionClass_LargeCon
-------------------------------------------------------------
NP_004893       Con      121351.4264                    True
NP_005079       Con     9516594.9789                    True
NP_005500    NonCon           0.0000                   False
NP_055852    NonCon    10933217.7090                   False
-------------------------------------------------------------
```

Exporting

Writing delimited formats

```
>>> table = LoadTable('stats.txt', sep=',')
>>> table.write('stats_tab.txt', sep='\t')
```

Writing latex format

It is also possible to specify column alignment, table caption and other arguments.

```
>>> table = LoadTable('stats.txt', sep=',')
>>> print(table.tostring(format='latex'))
\begin{longtable}[htp!]{ r r r }
\hline
\bf{Locus} & \bf{Region} & \bf{Ratio} \\
\hline
\hline
NP_003077 &    Con &        2.5386 \\
NP_004893 &    Con &   121351.4264 \\
NP_005079 &    Con &  9516594.9789 \\
NP_005500 & NonCon &        0.0000 \\
NP_055852 & NonCon & 10933217.7090 \\
\hline
\end{longtable}
```

Writing bedGraph format

This format allows display of annotation tracks on genome browsers.

```
>>> rows = [['1', 100, 101, 1.123], ['1', 101, 102, 1.123],
...         ['1', 102, 103, 1.123], ['1', 103, 104, 1.123],
...         ['1', 104, 105, 1.123], ['1', 105, 106, 1.123],
...         ['1', 106, 107, 1.123], ['1', 107, 108, 1.123],
...         ['1', 108, 109, 1], ['1', 109, 110, 1],
...         ['1', 110, 111, 1], ['1', 111, 112, 1],
...         ['1', 112, 113, 1], ['1', 113, 114, 1],
...         ['1', 114, 115, 1], ['1', 115, 116, 1],
...         ['1', 116, 117, 1], ['1', 117, 118, 1],
...         ['1', 118, 119, 2], ['1', 119, 120, 2],
...         ['1', 120, 121, 2], ['1', 150, 151, 2],
```

```
...          ['1', 151, 152, 2], ['1', 152, 153, 2],
...          ['1', 153, 154, 2], ['1', 154, 155, 2],
...          ['1', 155, 156, 2], ['1', 156, 157, 2],
...          ['1', 157, 158, 2], ['1', 158, 159, 2],
...          ['1', 159, 160, 2], ['1', 160, 161, 2]]
...
>>> bgraph = LoadTable(header=['chrom', 'start', 'end', 'value'],
...                    rows=rows)
...
>>> print(bgraph.tostring(format='bedgraph', name='test track',
...     description='test of bedgraph', color=(255,0,0)))
track type=bedGraph name="test track" description="test of bedgraph" color=255,0,0
1   100     108     1.12
1   108     118     1.0
1   118     161     2.0

The bedgraph formatter defaults to rounding values to 2 decimal places. You can␣
↪adjust that precision using the ``digits`` argument.
```

```
>>> print(bgraph.tostring(format='bedgraph', name='test track',
...   description='test of bedgraph', color=(255,0,0), digits=0))
track type=bedGraph name="test track" description="test of bedgraph" color=255,0,0
1   100     118     1.0
1   118     161     2.0
```

Building alignments

Using the cogent3 aligners

Running a pairwise Needleman-Wunsch-Alignment

Running a progressive aligner

We import useful functions and then load the sequences to be aligned.

```
>>> from cogent3 import LoadSeqs, LoadTree, DNA
>>> seqs = LoadSeqs('data/test2.fasta', aligned=False, moltype=DNA)
```

For nucleotides

We load a canned nucleotide substitution model and the progressive aligner `TreeAlign` function.

```
>>> from cogent3.evolve.models import HKY85
>>> from cogent3.align.progressive import TreeAlign
```

We first align without providing a guide tree. The `TreeAlign` algorithm builds pairwise alignments and estimates the substitution model parameters and pairwise distances. The distances are used to build a neighbour joining tree and the median value of substitution model parameters are provided to the substitution model for the progressive alignment step.

```
>>> aln, tree = TreeAlign(HKY85(), seqs)
Param Estimate Summary Stats: kappa
==============================
        Statistic        Value
------------------------------
            Count           10
              Sum        1e+06
           Median        4.256
             Mean        1e+05
StandardDeviation    3.162e+05
         Variance        1e+11
------------------------------
>>> aln
5 x 60 bytes alignment: NineBande[-C-----GCCA...], Mouse[GCAGTGAGCCA...],␣
↪DogFaced[GCAAGGAGCCA...], ...
```

We then align using a guide tree (pre-estimated) and specifying the ratio of transitions to transversions (kappa).

```
>>> tree = LoadTree(treestring='(((NineBande:0.0128202449453,Mouse:0.184732725695):0.
↪0289459522137,DogFaced:0.0456427810916):0.0271363715538,Human:0.0341320714654,
↪HowlerMon:0.0188456837006)root;')
>>> params={'kappa': 4.0}
>>> aln, tree = TreeAlign(HKY85(), seqs, tree=tree, param_vals=params)
>>> aln
5 x 60 bytes alignment: NineBande[-C-----GCCA...], Mouse[GCAGTGAGCCA...],␣
↪DogFaced[GCAAGGAGCCA...], ...
```

For codons

We load a canned codon substitution model and use a pre-defined tree and parameter estimates.

```
>>> from cogent3.evolve.models import MG94HKY
>>> tree = LoadTree(treestring='((NineBande:0.0575781680031,Mouse:0.594704139406):0.
↪078919659556,DogFaced:0.142151930069,(HowlerMon:0.0619991555435,Human:0.
↪10343006422):0.0792423439112)')
>>> params={'kappa': 4.0, 'omega': 1.3}
>>> aln, tree = TreeAlign(MG94HKY(), seqs, tree=tree, param_vals=params)
>>> aln
5 x 60 bytes alignment: NineBande[------CGCCA...], Mouse[GCAGTGAGCCA...],␣
↪DogFaced[GCAAGGAGCCA...], ...
```

Converting gaps from aa-seq alignment to nuc seq alignment

We load some unaligned DNA sequences and show their translation.

```
>>> from cogent3 import LoadSeqs, DNA, PROTEIN
>>> seqs = [('hum',
↪'AAGCAGATCCAGGAAAGCAGCGAGAATGGCAGCCTGGCCGCGCGCCAGGAGAGGCAGGCCCAGGTCAACCTCACT'),
...          ('mus',
↪'AAGCAGATCCAGGAGAGCGGCGAGAGCGGCAGCCTGGCCGCGCGGCAGGAGAGGCAGGCCCAAGTCAACCTCACG'),
...          ('rat', 'CTGAACAAGCAGCCACTTTCAAACAAGAAA')]
>>> unaligned_DNA = LoadSeqs(data=seqs, moltype=DNA, aligned=False)
>>> print(unaligned_DNA.to_fasta())
>hum
```

```
AAGCAGATCCAGGAAAGCAGCGAGAATGGCAGCCTGGCCGCGCGCCAGGAGAGGCAGGCCCAGGTCAACCTCACT
>mus
AAGCAGATCCAGGAGAGCGGCGAGAGCGGCAGCCTGGCCGCGCGGCAGGAGAGGCAGGCCCAAGTCAACCTCACG
>rat
CTGAACAAGCAGCCACTTTCAAACAAGAAA
>>> print(unaligned_DNA.get_translation())
>hum
KQIQESSENGSLAARQERQAQVNLT
>mus
KQIQESGESGSLAARQERQAQVNLT
>rat
LNKQPLSNKK
```

We load an alignment of these protein sequences.

```
>>> aligned_aa_seqs = [('hum', 'KQIQESSENGSLAARQERQAQVNLT'),
...                    ('mus', 'KQIQESGESGSLAARQERQAQVNLT'),
...                    ('rat', 'LNKQ------PLS---------NKK')]
>>> aligned_aa = LoadSeqs(data=aligned_aa_seqs, moltype=PROTEIN)
```

We then obtain an alignment of the DNA sequences from the alignment of their translation.

```
>>> aligned_DNA = aligned_aa.replace_seqs(unaligned_DNA, aa_to_codon=True)
>>> print(aligned_DNA)
>hum
AAGCAGATCCAGGAAAGCAGCGAGAATGGCAGCCTGGCCGCGCGCCAGGAGAGGCAGGCCCAGGTCAACCTCACT
>mus
AAGCAGATCCAGGAGAGCGGCGAGAGCGGCAGCCTGGCCGCGCGGCAGGAGAGGCAGGCCCAAGTCAACCTCACG
>rat
CTGAACAAGCAG------------------CCACTTTCA--------------------------AACAAGAAA
```

Setting the argument `aa_to_codons=False` is only useful when the sequences have exactly the length. One use case is to allow introducing the gaps onto another copy of the alignment where there are annotations.

Building phylogenies

Built-in Phylogenetic reconstruction

By distance method

Given an alignment, a phylogenetic tree can be generated based on the pair-wise distance matrix computed from the alignment.

Fast pairwise distance estimation

For a limited number of evolutionary models a fast implementation is available. Here we use the Tamura and Nei 1993 model.

```
>>> from cogent3 import LoadSeqs, DNA
>>> from cogent3.evolve.pairwise_distance import TN93Pair
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> dist_calc = TN93Pair(DNA, alignment=aln)
>>> dist_calc.run(show_progress=False)
```

We can obtain the distances as a `dict` for direct usage in phylogenetic reconstruction

```
>>> dists = dist_calc.get_pairwise_distances()
```

or as a table for display / saving

```
>>> print(dist_calc.dists[:4,:4]) # truncated to fit screens
Pairwise Distances
==========================================
Seq1 \ Seq2    Galago    HowlerMon    Rhesus
------------------------------------------
     Galago          *       0.2157    0.1962
  HowlerMon     0.2157            *    0.0736
     Rhesus     0.1962       0.0736         *
  Orangutan     0.1944       0.0719    0.0411
------------------------------------------
```

Other statistics are also available, such the as the standard errors of the estimates.

```
>>> print(dist_calc.stderr[:4,:4]) # truncated to fit screens
Standard Error of Pairwise Distances
==========================================
Seq1 \ Seq2    Galago    HowlerMon    Rhesus
------------------------------------------
     Galago          *       0.0103    0.0096
  HowlerMon     0.0103            *    0.0054
     Rhesus     0.0096       0.0054         *
  Orangutan     0.0095       0.0053    0.0039
------------------------------------------
```

More general estimation of pairwise distances

The standard `cogent3` likelihood function can also be used to estimate distances. Because these require numerical optimisation they can be significantly slower than the fast estimation approach above.

```
>>> from cogent3 import LoadSeqs, DNA
>>> from cogent3.phylo import distance
>>> from cogent3.evolve.models import F81
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> d = distance.EstimateDistances(aln, submodel=F81())
>>> d.run(show_progress=False)
```

The example above will use the F81 nucleotide substitution model and run the `distance.EstimateDistances()` method with the default options for the optimiser. To configure the optimiser a dictionary of optimisation options can be passed onto the `run` command. The example below configures the `Powell` optimiser to run a maximum of 10000 evaluations, with a maximum of 5 restarts (a total of 5 x 10000 = 50000 evaluations).

```
>>> dist_opt_args = dict(max_restarts=5, max_evaluations=10000,
...                      show_progress=False)
>>> d.run(dist_opt_args=dist_opt_args)
>>> print(d)
=============================================================================================
Seq1 \ Seq2    Galago    HowlerMon    Rhesus    Orangutan    Gorilla     Human     ␣
↪Chimpanzee
---------------------------------------------------------------------------------------------
↪------
```

```
     Galago          *       0.2112    0.1930       0.1915    0.1891    0.1934
↪0.1892
  HowlerMon    0.2112            *    0.0729       0.0713    0.0693    0.0729
↪0.0697
     Rhesus    0.1930       0.0729         *       0.0410    0.0391    0.0421
↪0.0395
  Orangutan    0.1915       0.0713    0.0410            *    0.0136    0.0173
↪0.0140
    Gorilla    0.1891       0.0693    0.0391       0.0136         *    0.0086
↪0.0054
      Human    0.1934       0.0729    0.0421       0.0173    0.0086         *
↪0.0089
 Chimpanzee    0.1892       0.0697    0.0395       0.0140    0.0054    0.0089
↪     *
-------------------------------------------------------------------------------
↪------
```

Building A Phylogenetic Tree From Pairwise Distances

Phylogenetic Trees can be built by using the neighbour joining algorithm by providing a dictionary of pairwise distances. This dictionary can be obtained either from the output of `distance.EstimateDistances()`

```
>>> from cogent3.phylo import nj
>>> njtree = nj.nj(d.get_pairwise_distances())
>>> njtree = njtree.balanced()
>>> print(njtree.ascii_art())
                    /-Rhesus
          /edge.1--|
         |         |          /-HowlerMon
         |          \edge.0--|
         |                    \-Galago
-root----|
         |--Orangutan
         |
         |          /-Human
          \edge.2--|
                   |          /-Gorilla
                    \edge.3--|
                              \-Chimpanzee
```

Or created manually as shown below.

```
>>> dists = {('a', 'b'): 2.7, ('c', 'b'): 2.33, ('c', 'a'): 0.73}
>>> njtree2 = nj.nj(dists)
>>> print(njtree2.ascii_art())
          /-a
         |
-root----|--b
         |
          \-c
```

By least-squares

We illustrate the phylogeny reconstruction by least-squares using the F81 substitution model. We use the advanced-stepwise addition algorithm to search tree space. Here `a` is the number of taxa to exhaustively evaluate all possible

phylogenies for. Successive taxa will are added to the top `k` trees (measured by the least-squares metric) and `k` trees are kept at each iteration.

```
>>> import pickle
>>> from cogent3.phylo.least_squares import WLS
>>> dists = pickle.load(open('data/dists_for_phylo.pickle', 'rb'))
>>> ls = WLS(dists)
>>> stat, tree = ls.trex(a=5, k=5, show_progress=False)
```

Other optional arguments that can be passed to the `trex` method are: `return_all`, whether the `k` best trees at the final step are returned as a `ScoredTreeCollection` object; `order`, a series of tip names whose order defines the sequence in which tips will be added during tree building (this allows the user to randomise the input order).

By ML

We illustrate the phylogeny reconstruction using maximum-likelihood using the F81 substitution model. We use the advanced-stepwise addition algorithm to search tree space, setting

```
>>> from cogent3 import LoadSeqs, DNA
>>> from cogent3.phylo.maximum_likelihood import ML
>>> from cogent3.evolve.models import F81
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> ml = ML(F81(), aln)
```

The `ML` object also has the `trex` method and this can be used in the same way as for above, i.e. `ml.trex()`. We don't do that here because this is a very slow method for phylogenetic reconstruction.

Evolutionary analysis using likelihood

Specifying substitution models

Canned models

Many standard evolutionary models come pre-defined in the `cogent3.evolve.models` module.

The available nucleotide, codon and protein models are

```
>>> from cogent3.evolve import models
>>> print(models.nucleotide_models)
['JC69', 'K80', 'F81', 'HKY85', 'TN93', 'GTR']
>>> print(models.codon_models)
['CNFGTR', 'CNFHKY', 'MG94HKY', 'MG94GTR', 'GY94', 'H04G', 'H04GK', 'H04GGK']
>>> print(models.protein_models)
['DSO78', 'AH96', 'AH96_mtmammals', 'JTT92', 'WG01']
```

While those values are strings, a function of the same name exists within the module so creating the substitution models requires only calling that function. I demonstrate that for a nucleotide model here.

```
>>> from cogent3.evolve.models import F81
>>> sub_mod = F81()
```

We'll be using these for the examples below.

Rate heterogeneity models

We illustrate this for the gamma distributed case using examples of the canned models displayed above. Creating rate heterogeneity variants of the canned models can be done by using optional arguments that get passed to the substitution model class.

For nucleotide

We specify a general time reversible nucleotide model with gamma distributed rate heterogeneity.

```
>>> from cogent3.evolve.models import GTR
>>> sub_mod = GTR(with_rate=True, distribution='gamma')
>>> print(sub_mod)

Nucleotide ( name = 'GTR'; type = 'None'; params = ['A/C', 'A/G', 'A/T', 'C/G', 'C/T
↪']; number of motifs = 4; motifs = ['T', 'C', 'A', 'G'])
```

For codon

We specify a conditional nucleotide frequency codon model with nucleotide general time reversible parameters and a parameter for the ratio of nonsynonymous to synonymous substitutions (omega) with gamma distributed rate heterogeneity.

```
>>> from cogent3.evolve.models import CNFGTR
>>> sub_mod = CNFGTR(with_rate=True, distribution='gamma')
>>> print(sub_mod)

Codon ( name = 'CNFGTR'; type = 'None'; params = ['A/C', 'A/G', 'A/T', 'C/G', 'C/T',
↪'omega']; ...
```

For protein

We specify a Jones, Taylor and Thornton 1992 empirical protein substitution model with gamma distributed rate heterogeneity.

```
>>> from cogent3.evolve.models import JTT92
>>> sub_mod = JTT92(with_rate=True, distribution='gamma')
>>> print(sub_mod)

Empirical ( name = 'JTT92'; type = 'None'; number of motifs = 20; motifs = ['A', 'C'..
↪.
```

Specifying likelihood functions

Making a likelihood function

You start by specifying a substitution model and use that to construct a likelihood function for a specific tree.

```
>>> from cogent3 import LoadTree
>>> from cogent3.evolve.models import F81
>>> sub_mod = F81()
```

```
>>> tree = LoadTree(treestring='(a,b,(c,d))')
>>> lf = sub_mod.make_likelihood_function(tree)
```

Providing an alignment to a likelihood function

You need to load an alignment and then provide it a likelihood function. I construct very simple trees and alignments for this example.

```
>>> from cogent3 import LoadTree, LoadSeqs
>>> from cogent3.evolve.models import F81
>>> sub_mod = F81()
>>> tree = LoadTree(treestring='(a,b,(c,d))')
>>> lf = sub_mod.make_likelihood_function(tree)
>>> aln = LoadSeqs(data=[('a', 'ACGT'), ('b', 'AC-T'), ('c', 'ACGT'),
...                      ('d', 'AC-T')])
...
>>> lf.set_alignment(aln)
```

Scoping parameters on trees

For many evolutionary analyses, it's desirable to allow different branches on a tree to have different values of a parameter. We show this for a simple codon model case here where we want the great apes (the clade that includes human and orangutan) to have a different value of the ratio of nonsynonymous to synonymous substitutions. This parameter is identified in the precanned `CNFGTR` model as `omega`.

```
>>> from cogent3 import LoadTree
>>> from cogent3.evolve.models import CNFGTR
>>> tree = LoadTree('data/primate_brca1.tree')
>>> print(tree.ascii_art())
          /-Galago
         |
-root----|--HowlerMon
         |
         |          /-Rhesus
          \edge.3--|
                   |          /-Orangutan
                    \edge.2--|
                             |          /-Gorilla
                              \edge.1--|
                                       |          /-Human
                                        \edge.0--|
                                                  \-Chimpanzee
>>> sm = CNFGTR()
>>> lf = sm.make_likelihood_function(tree, digits=2)
>>> lf.set_param_rule('omega', tip_names=['Human', 'Orangutan'], outgroup_name='Galago
↪', is_clade=True, init=0.5)
```

We've set an *initial* value for this clade so that the edges affected by this rule are evident below.

```
>>> print(lf)
Likelihood function statistics
number of free parameters = 78
====================================
 A/C     A/G     A/T     C/G     C/T
------------------------------------
```

```
1.00    1.00    1.00    1.00    1.00
-------------------------------------
=====================================
      edge    parent    length    omega
-------------------------------------
    Galago      root      1.00     1.00
 HowlerMon      root      1.00     1.00
    Rhesus    edge.3      1.00     1.00
 Orangutan    edge.2      1.00     0.50
   Gorilla    edge.1      1.00     0.50
     Human    edge.0      1.00     0.50
Chimpanzee    edge.0      1.00     0.50
    edge.0    edge.1      1.00     0.50
    edge.1    edge.2      1.00     0.50
    edge.2    edge.3      1.00     1.00
    edge.3      root      1.00     1.00
-------------------------------------...
```

A more extensive description of capabilities is in *Allowing substitution model parameters to differ between branches*.

Specifying parameter values

Specifying a parameter as constant

This means the parameter will not be modified during likelihood maximisation. We show this here by making the `omega` parameter constant at the value 1 – essentially the condition of selective neutrality.

```
>>> from cogent3 import LoadTree
>>> from cogent3.evolve.models import CNFGTR
>>> tree = LoadTree('data/primate_brca1.tree')
>>> sm = CNFGTR()
>>> lf = sm.make_likelihood_function(tree, digits=2)
>>> lf.set_param_rule('omega', is_constant=True)
```

Providing a starting value for a parameter

This can be useful to improve performance, the closer you are to the maximum likelihood estimator the quicker optimisation will be.

```
>>> from cogent3 import LoadTree
>>> from cogent3.evolve.models import CNFGTR
>>> tree = LoadTree('data/primate_brca1.tree')
>>> sm = CNFGTR()
>>> lf = sm.make_likelihood_function(tree, digits=2)
>>> lf.set_param_rule('omega', init=0.1)
```

Setting parameter bounds for optimisation

This can be useful for stopping optimisers from getting stuck in a bad part of parameter space. The following is for `omega` in a codon model. I'm also providing an initial guess for the parameter (`init=0.1`) as well as a lower bound. An initial guess that is close to the maximum likelihood estimate will speed up optimisation.

```
>>> from cogent3 import LoadTree
>>> from cogent3.evolve.models import CNFGTR
>>> tree = LoadTree('data/primate_brca1.tree')
>>> sm = CNFGTR()
>>> lf = sm.make_likelihood_function(tree, digits=2)
>>> lf.set_param_rule('omega', init=0.1, lower=1e-9, upper=20.0)
```

Setting an upper bound for branch length

If the branch length estimates seem too large, setting just an upper bound can be sensible. This will apply to all edges on the tree.

```
>>> from cogent3 import LoadTree
>>> from cogent3.evolve.models import F81
>>> tree = LoadTree('data/primate_brca1.tree')
>>> sm = F81()
>>> lf = sm.make_likelihood_function(tree)
>>> lf.set_param_rule('length', upper=1.0)
```

Note: If, after optimising, the branch lengths equal to the upper value you set then the function has not been fully maximised and you should consider adjusting the boundary again.

Specifying rate heterogeneity functions

We extend the simple gamma distributed rate heterogeneity case for nucleotides from above to construction of the actual likelihood function. We do this for 4 bins and constraint the bin probabilities to be equal.

```
>>> from cogent3 import LoadTree, LoadSeqs
>>> from cogent3.evolve.models import GTR
>>> sm = GTR(with_rate=True, distribution='gamma')
>>> tree = LoadTree('data/primate_brca1.tree')
>>> lf = sm.make_likelihood_function(tree, bins=4, digits=2)
>>> lf.set_param_rule('bprobs', is_constant=True)
```

For more detailed discussion of defining and using these models see *Analysis of rate heterogeneity*.

Specifying Phylo-HMMs

```
>>> from cogent3 import LoadTree, LoadSeqs
>>> from cogent3.evolve.models import GTR
>>> sm = GTR(with_rate=True, distribution='gamma')
>>> tree = LoadTree('data/primate_brca1.tree')
>>> lf = sm.make_likelihood_function(tree, bins=4, sites_independent=False,
...                                  digits=2)
>>> lf.set_param_rule('bprobs', is_constant=True)
```

For more detailed discussion of defining and using these models see *Evaluate process heterogeneity using a Hidden Markov Model*.

Fitting likelihood functions

Choice of optimisers

There are 2 types of optimiser: simulated annealing, a *global* optimiser; and Powell, a *local* optimiser. The simulated annealing method is slow compared to Powell and in general Powell is an adequate choice. I setup a simple nucleotide model to illustrate these.

```
>>> from cogent3 import LoadTree, LoadSeqs
>>> from cogent3.evolve.models import F81
>>> tree = LoadTree('data/primate_brca1.tree')
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> sm = F81()
>>> lf = sm.make_likelihood_function(tree, digits=3, space=2)
>>> lf.set_alignment(aln)
```

The default is to use the simulated annealing optimiser followed by Powell.

```
>>> lf.optimise(show_progress=False)
```

We can specify just using the local optimiser. To do so, it's recommended to set the `max_restarts` argument since this provides a mechanism for Powell to attempt restarting the optimisation from slightly different sport which can help in overcoming local maxima.

```
>>> lf.optimise(local=True, max_restarts=5, show_progress=False)
```

We might want to do crude simulated annealing following by more rigorous Powell.

```
>>> lf.optimise(show_progress=False, global_tolerance=1.0, tolerance=1e-8,
...             max_restarts=5)
```

Checkpointing runs

See *Checkpointing optimisation runs*.

How to check your optimisation was successful.

There is no guarantee that an optimised function has achieved a global maximum. We can, however, be sure that a maximum was achieved by validating that the optimiser stopped because the specified tolerance condition was met, rather than exceeding the maximum number of evaluations. The latter number is set to ensure optimisation doesn't proceed endlessly. If the optimiser exited because this limit was exceeded you can be sure that the function **has not** been successfully optimised.

We can monitor this situation using the `limit_action` argument to `optimise`. Providing the value `raise` causes an exception to be raised if this condition occurs, as shown below. Providing `warn` (default) instead will cause a warning message to be printed to screen but execution will continue. The value `ignore` hides any such message.

```
>>> from cogent3 import LoadTree, LoadSeqs
>>> from cogent3.evolve.models import F81
>>> tree = LoadTree('data/primate_brca1.tree')
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> sm = F81()
>>> lf = sm.make_likelihood_function(tree, digits=3, space=2)
>>> lf.set_alignment(aln)
>>> max_evals = 10
```

```
>>> lf.optimise(show_progress=False, limit_action='raise',
...              max_evaluations=max_evals, return_calculator=True)
...
Traceback (most recent call last):
ArithmeticError: FORCED EXIT from optimiser after 10 evaluations
```

Note: We recommend using `limit_action='raise'` and catching the `ArithmeticError` error explicitly. You really shouldn't be using results from such an optimisation run.

Getting statistics out of likelihood functions

Model fit statistics

Log likelihood and number of free parameters

```
>>> from cogent3 import LoadTree, LoadSeqs
>>> from cogent3.evolve.models import GTR
>>> sm = GTR()
>>> tree = LoadTree('data/primate_brca1.tree')
>>> lf = sm.make_likelihood_function(tree)
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> lf.set_alignment(aln)
```

We get the log-likelihood and the number of free parameters.

```
>>> lnL = lf.get_log_likelihood()
>>> print(lnL)
-24601.9...
>>> nfp = lf.get_num_free_params()
>>> print(nfp)
16
```

Warning: The number of free parameters (nfp) refers only to the number of parameters that were modifiable by the optimiser. Typically, the degrees-of-freedom of a likelihood ratio test statistic is computed as the difference in nfp between models. This will not be correct for models in which boundary conditions exist (rate heterogeneity models where a parameter value boundary is set between bins).

Information theoretic measures

Aikake Information Criterion

Note: this measure only makes sense when the model has been optimised, a step I'm skipping here in the interests of speed.

```
>>> from cogent3 import LoadTree, LoadSeqs
>>> from cogent3.evolve.models import GTR
```

```
>>> sm = GTR()
>>> tree = LoadTree('data/primate_brca1.tree')
>>> lf = sm.make_likelihood_function(tree)
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> lf.set_alignment(aln)
>>> AIC = lf.get_aic()
>>> AIC
49235.869...
```

We can also get the second-order AIC.

```
>>> AICc = lf.get_aic(second_order=True)
>>> AICc
49236.064...
```

Bayesian Information Criterion

Note: this measure only makes sense when the model has been optimised, a step I'm skipping here in the interests of speed.

```
>>> from cogent3 import LoadTree, LoadSeqs
>>> from cogent3.evolve.models import GTR
>>> sm = GTR()
>>> tree = LoadTree('data/primate_brca1.tree')
>>> lf = sm.make_likelihood_function(tree)
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> lf.set_alignment(aln)
>>> BIC = lf.get_bic()
>>> BIC
49330.9475...
```

Getting maximum likelihood estimates

We fit the model defined in the previous section and use that in the following.

One at a time

We get the statistics out individually. We get the `length` for the Human edge and the exchangeability parameter `A/G`.

```
>>> lf.optimise(local=True, show_progress=False)
>>> a_g = lf.get_param_value('A/G')
>>> print(a_g)
5.25...
>>> human = lf.get_param_value('length', 'Human')
>>> print(human)
0.006...
```

Just the motif probabilities

```
>>> mprobs = lf.get_motif_probs()
>>> print(mprobs)
====================================
     T         C         A         G
------------------------------------
0.2406    0.1742    0.3757    0.2095
------------------------------------
```

On the tree object

If written to file in xml format, then model parameters will be saved. This can be useful for later plotting or recreating likelihood functions.

```
>>> annot_tree = lf.get_annotated_tree()
>>> print(annot_tree.get_xml())
<?xml version="1.0"?>
<clade>
  <clade>
     <name>Galago</name>
     <param><name>A/G</name><value>5.25342689214</value></param>
     <param><name>A/C</name><value>1.23159157151</value></param>
     <param><name>C/T</name><value>5.97001104267</value></param>
     <param><name>length</name><value>0.173114172705</value></param>...
```

Warning: This method fails for some rate-heterogeneity models.

As tables

```
>>> tables = lf.get_statistics(with_motif_probs=True, with_titles=True)
>>> for table in tables:
...     if 'global' in table.title:
...         print(table)
global params
=============================================
   A/C       A/G       A/T       C/G       C/T
---------------------------------------------
1.2316    5.2534    0.9585    2.3159    5.9700
---------------------------------------------
```

Testing hypotheses

Using likelihood ratio tests

We test the molecular clock hypothesis for human and chimpanzee lineages. The null has these two branches constrained to be equal.

```
>>> from cogent3 import LoadTree, LoadSeqs
>>> from cogent3.evolve.models import F81
>>> tree = LoadTree('data/primate_brca1.tree')
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> sm = F81()
>>> lf = sm.make_likelihood_function(tree, digits=3, space=2)
>>> lf.set_alignment(aln)
>>> lf.set_param_rule('length', tip_names=['Human', 'Chimpanzee'],
...         outgroup_name='Galago', is_clade=True, is_independent=False)
...
>>> lf.set_name('Null Hypothesis')
>>> lf.optimise(local=True, show_progress=False)
>>> null_lnL = lf.get_log_likelihood()
>>> null_nfp = lf.get_num_free_params()
>>> print(lf)
Null Hypothesis
log-likelihood = -7177.4403
number of free parameters = 10
=========================
      edge  parent  length
-------------------------
    Galago    root   0.167
 HowlerMon    root   0.044
    Rhesus  edge.3   0.021
 Orangutan  edge.2   0.008
   Gorilla  edge.1   0.002
     Human  edge.0   0.004
Chimpanzee  edge.0   0.004
    edge.0  edge.1   0.000...
```

The alternate allows the human and chimpanzee branches to differ by just setting all lengths to be independent.

```
>>> lf.set_param_rule('length', is_independent=True)
>>> lf.set_name('Alt Hypothesis')
>>> lf.optimise(local=True, show_progress=False)
>>> alt_lnL = lf.get_log_likelihood()
>>> alt_nfp = lf.get_num_free_params()
>>> print(lf)
Alt Hypothesis
log-likelihood = -7175.7756
number of free parameters = 11
=========================
      edge  parent  length
-------------------------
    Galago    root   0.167
 HowlerMon    root   0.044
    Rhesus  edge.3   0.021
 Orangutan  edge.2   0.008
   Gorilla  edge.1   0.002
     Human  edge.0   0.006
Chimpanzee  edge.0   0.003
    edge.0  edge.1   0.000
    edge.1  edge.2   0.003
    edge.2  edge.3   0.012
    edge.3    root   0.009
-------------------------
=============
motif  mprobs
```

```
-------------
    T   0.241
    C   0.174
    A   0.376
    G   0.209
-------------
```

We import the function for computing the probability of a chi-square test statistic, compute the likelihood ratio test statistic, degrees of freedom and the corresponding probability.

```
>>> from cogent3.maths.stats import chisqprob
>>> LR = 2 * (alt_lnL - null_lnL) # the likelihood ratio statistic
>>> df = (alt_nfp - null_nfp) # the test degrees of freedom
>>> p = chisqprob(LR, df)
>>> print('LR=%.4f ; df = %d ; p=%.4f' % (LR, df, p))
LR=3.3294 ; df = 1 ; p=0.0681
```

By parametric bootstrapping

If we can't rely on the asymptotic behaviour of the LRT, e.g. due to small alignment length, we can use a parametric bootstrap. Convenience functions for that are described in more detail here *Performing a parametric bootstrap*.

In general, however, this capability derives from the ability of any defined `evolve` likelihood function to simulate an alignment. This property is provided as `simulate_alignment` method on likelihood function objects.

```
>>> from cogent3 import LoadTree, LoadSeqs
>>> from cogent3.evolve.models import F81
>>> tree = LoadTree('data/primate_brca1.tree')
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> sm = F81()
>>> lf = sm.make_likelihood_function(tree, digits=3, space=2)
>>> lf.set_alignment(aln)
>>> lf.set_param_rule('length', tip_names=['Human', 'Chimpanzee'],
...         outgroup_name='Galago', is_clade=True, is_independent=False)
...
>>> lf.set_name('Null Hypothesis')
>>> lf.optimise(local=True, show_progress=False)
>>> sim_aln = lf.simulate_alignment()
>>> print(repr(sim_aln))
7 x 2814 dna alignment: Galago...
```

Determining confidence intervals on MLEs

The profile method is used to calculate a confidence interval for a named parameter. We show it here for a global substitution model exchangeability parameter (*kappa*, the ratio of transition to transversion rates) and for an edge specific parameter (just the human branch length).

```
>>> from cogent3 import LoadTree, LoadSeqs
>>> from cogent3.evolve.models import HKY85
>>> tree = LoadTree('data/primate_brca1.tree')
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> sm = HKY85()
>>> lf = sm.make_likelihood_function(tree)
>>> lf.set_alignment(aln)
>>> lf.optimise(local=True, show_progress=False)
```

```
>>> kappa_lo, kappa_mle, kappa_hi = lf.get_param_interval('kappa')
>>> print("lo=%.2f ; mle=%.2f ; hi = %.2f" % (kappa_lo, kappa_mle, kappa_hi))
lo=3.78 ; mle=4.44 ; hi = 5.22
>>> human_lo, human_mle, human_hi = lf.get_param_interval('length', 'Human')
>>> print("lo=%.2f ; mle=%.2f ; hi = %.2f" % (human_lo, human_mle, human_hi))
lo=0.00 ; mle=0.01 ; hi = 0.01
```

Saving results

Use either the annotated tree or statistics tables to obtain objects that can easily be written to file.

Visualising statistics on trees

We look at the distribution of `omega` from the CNF codon model family across different primate lineages. We allow each edge to have an independent value for `omega`.

```
>>> from cogent3 import LoadTree, LoadSeqs
>>> from cogent3.evolve.models import CNFGTR
>>> tree = LoadTree('data/primate_brca1.tree')
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> sm = CNFGTR()
>>> lf = sm.make_likelihood_function(tree, digits=2, space=2)
>>> lf.set_param_rule('omega', is_independent=True, upper=10.0)
>>> lf.set_alignment(aln)
>>> lf.optimise(show_progress=False, local=True)
>>> print(lf)
Likelihood function statistics
log-likelihood = -6755.9726
number of free parameters = 27
============================
 A/C   A/G   A/T   C/G   C/T
----------------------------
1.07  3.88  0.79  1.96  4.09
----------------------------

================================
      edge  parent  length  omega
--------------------------------
    Galago    root    0.53   0.85
 HowlerMon    root    0.14   0.71
    Rhesus  edge.3    0.07   0.58
 Orangutan  edge.2    0.02   0.49
   Gorilla  edge.1    0.01   0.43
     Human  edge.0    0.02   2.44
Chimpanzee  edge.0    0.01   2.28
    edge.0  edge.1    0.00   0.00
    edge.1  edge.2    0.01   0.55
    edge.2  edge.3    0.04   0.33
    edge.3    root    0.02   1.10
--------------------------------

===============
motif    mprobs
---------------
  AAA      0.06
  AAC      0.02
  AAG      0.03
```

```
  AAT       0.06
  ACA       0.02
  ...        ...
  TGT       0.02
  TTA       0.02
  TTC       0.01
  TTG       0.01
  TTT       0.02
---------------
```

We need an annotated tree object to do the drawing, we write this out to an XML formatted file so it can be reloaded for later reuse.

```
>>> annot_tree = lf.get_annotated_tree()
>>> annot_tree.write('result_tree.xml')
```

We first import an unrooted dendrogram and then generate a heat mapped image to file where edges are colored red by the magnitude of `omega` with maximal saturation when `omega=1`.

```
>>> from cogent3.draw.dendrogram import ContemporaneousDendrogram
>>> dend = ContemporaneousDendrogram(annot_tree)
>>> fig = dend.make_figure(height=6, width=6, shade_param='omega',
...                        max_value=1.0, stroke_width=2)
>>> fig.savefig('omega_heat_map.png')
```

Reconstructing ancestral sequences

We first fit a likelihood function.

```
>>> from cogent3 import LoadTree, LoadSeqs
>>> from cogent3.evolve.models import F81
>>> tree = LoadTree('data/primate_brca1.tree')
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> sm = F81()
>>> lf = sm.make_likelihood_function(tree, digits=3, space=2)
>>> lf.set_alignment(aln)
>>> lf.optimise(show_progress=False, local=True)
```

We then get the most likely ancestral sequences.

```
>>> ancestors = lf.likely_ancestral_seqs()
>>> print(ancestors)
>root
TGTGGCACAAATACTCATGCCAGCTCATTACAGCA...
```

Or we can get the posterior probabilities (returned as a `DictArray`) of sequence states at each node.

```
>>> ancestral_probs = lf.reconstruct_ancestral_seqs()
>>> print(ancestral_probs['root'])
=============================================
             T         C         A         G
---------------------------------------------
   0    0.1816    0.0000    0.0000    0.0000
   1    0.0000    0.0000    0.0000    0.1561
   2    0.1816    0.0000    0.0000    0.0000
   3    0.0000    0.0000    0.0000    0.1561...
```

Tips for improved performance

Sequentially build the fitting

There's nothing that improves performance quite like being close to the maximum likelihood values. So using the `set_param_rule` method to provide good starting values can be very useful. As this can be difficult to do one easy way is to build simpler models that are nested within the one you're interested in. Fitting those models and then relaxing constraints until you're at the parameterisation of interest can markedly improve optimisation speed.

Being able to save results to file allows you to do this between sessions.

Sampling

If you're dealing with a very large alignment, another approach is to use a subset of the alignment to fit the model then try fitting the entire alignment. The alignment method does have an method to facilitate this approach. The following samples 99 codons without replacement.

```
>>> from cogent3 import LoadSeqs
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> smpl = aln.sample(n=99, with_replacement=False, motif_length=3)
>>> len(smpl)
297
```

While this samples 99 nucleotides without replacement.

```
>>> smpl = aln.sample(n=99, with_replacement=False)
>>> len(smpl)
99
```

Standard statistical analyses

Random Numbers

Many of the code snippets in this section use random numbers. These can be obtained using functions from Python's `random` module, or using `numpy.random`. To facilitate testing, the examples "seed" the random number generator, which ensures the same results each time the code is run.

```
>>> import random
>>> random.seed(157)
>>> random.choice((-1,1))
1
>>> random.choice(range(1000))
224
>>> random.gauss(mu=50, sigma=3)
52.7668...
>>> import numpy as np
>>> np.random.seed(157)
>>> np.random.random_integers(-1,1,5)
array([-1,  1,  1,  1,  0])
```

For the last example, note that the range includes 0.

```
>>> np.random.normal(loc=50,scale=3,size=2)
array([ 42.8217253 ,  49.90008293])
>>> np.random.randn(3)
array([ 1.26613052,  1.59533412,  0.95612413])
```

Summary statistics

Population mean and median

PyCogent3's functions for statistical analysis operate on `numpy` arrays

```
>>> import random
>>> import numpy as np
>>> import cogent3.maths.stats.test as stats
>>> random.seed(157)
>>> nums = [random.gauss(mu=50, sigma=3) for i in range(1000)]
>>> arr = np.array(nums)
>>> stats.mean(arr)
49.9927...
```

but in some cases they will also accept a simple list of values

```
>>> stats.mean(range(1,8))
4.0
>>> stats.var(range(1,8))
4.66666...
```

The keyword argument `axis` controls whether a function operates by rows (`axis=0`) or by columns (`axis=1`), or on all of the values (`axis=None`)

```
>>> import cogent3.maths.stats.test as stats
>>> import numpy as np
>>> nums = list(range(1,6)) + [50] + list(range(10,60,10)) + [500]
>>> arr = np.array(nums)
>>> arr.shape = (2,6)
>>> arr
array([[  1,   2,   3,   4,   5,  50],
       [ 10,  20,  30,  40,  50, 500]])
>>> stats.mean(arr, axis=0)
array([   5.5,   11. ,   16.5,   22. ,   27.5,  275. ])
>>> stats.mean(arr, axis=1)
array([  10.83333333,  108.33333333])
>>> stats.mean(arr)
59.58333...
>>> stats.median(arr, axis=0)
array([   5.5,   11. ,   16.5,   22. ,   27.5,  275. ])
>>> stats.median(arr, axis=1)
array([  3.5,  35. ])
>>> stats.median(arr)
15.0
```

Population variance and standard deviation

```
>>> print(stats.var(arr, axis=0))
[  4.05000000e+01   1.62000000e+02   3.64500000e+02   6.48000000e+02
   1.01250000e+03   1.01250000e+05]
>>> print(stats.std(arr, axis=0))
[   6.36396103   12.72792206   19.09188309   25.45584412   31.81980515
  318.19805153]
>>> print(stats.var(arr, axis=1))
[   370.16666667  37016.66666667]
>>> print(stats.std(arr, axis=1))
[  19.23971587  192.39715868]
>>> print(stats.var(arr, axis=None))
19586.6287879
>>> print(stats.std(arr, axis=None))
139.952237524
```

The variance (and standard deviation) are unbiased

```
>>> import numpy as np
>>> import cogent3.maths.stats.test as stats
>>> arr = np.array([1,2,3,4,5])
>>> m = np.mean(arr)
>>> stats.var(arr)
2.5
>>> 1.0 * sum([(n-m)**2 for n in arr]) / (len(arr) - 1)
2.5
```

Distributions

Binomial

The binomial distribution can be used for calculating the probability of specific frequencies of states occurring in discrete data. The two alternate states are typically referred to as a success or failure. This distribution is used for sign tests.

```
>>> import cogent3.maths.stats.distribution as distr
>>> distr.binomial_low(successes=5, trials=10, prob=0.5)
0.623...
>>> distr.binomial_high(successes=5, trials=10, prob=0.5)
0.376...
>>> distr.binomial_exact(successes=5, trials=10, prob=0.5)
0.246...
```

Chi-square

A convenience function for computing the probability of a chi-square statistic is provided at the `stats` top level.

```
>>> from cogent3.maths.stats import chisqprob
>>> chisqprob(3.84, 1)
0.05...
```

which is just a reference to the `chi_high` function.

```
>>> from cogent3.maths.stats.distribution import chi_high
>>> chi_high(3.84, 1)
0.05...
```

Getting the inverse

Given a probability we can determine the corresponding chi-square value for a given degrees-of-freedom.

```
>>> from cogent3.maths.stats.distribution import chdtri
>>> chdtri(1, 0.05)
3.84...
>>> chdtri(2, 0.05)
5.99...
```

Normal

The function `zprob()` takes a z-score or standard deviation and computes the fraction of the normal distribution (mean=0, std=1) which lies farther away from the mean than that value. For example, only about 4.5% of the values are more than 2 standard deviations away from the mean, so that more than 95% of the values are at least that close to the mean.

```
>>> import cogent3.maths.stats.distribution as distr
>>> for z in range(5):
...     print('%s %.4f' % (z, distr.zprob(z)))
...
0 1.0000
1 0.3173
2 0.0455
3 0.0027
4 0.0001
```

Use the functions `z_low()` and `z_high()` to compute the normal distribution in a directional fashion. Here we see that a z-score of 1.65 has a value greater than 95% of the values in the distribution, and similarly a z-score of 1.96 has a value greater than 97.5% of the values in the distribution.

```
>>> z = 0
>>> while distr.z_low(z) < 0.95:
...     z += 0.01
...
>>> z
1.6500...
>>> z = 0
>>> while distr.z_low(z) < 0.975:
...     z += 0.01
...
>>> z
1.9600...
```

Normalizing data (as Z-scores)

The function `z_test()` takes a sample of values as the first argument, and named arguments for the population parameters: `popmean` and `popstddev` (with default values of 0 and 1), and returns the z-score and its probability.

In this example, we grab a sample from a population with `mean=50` and `std=3`, and call `z_test()` with the population mean specified as 50 and the `popstddev` assuming its default value of 1:

```
>>> import numpy as np
>>> import cogent3.maths.stats.test as stats
>>> np.random.seed(157)
>>> arr = np.random.normal(loc=50, scale=3, size=1000)
>>> round(stats.mean(arr), 1)
49.9...
>>> round(stats.std(arr), 1)
3.1...
>>> z, prob = stats.z_test(arr, popmean=50.0)
>>> print(z)
-3.08...
```

Resampling based statistics

The Jackknife

This is a data resampling based approach to estimating the confidence in measures of location (like the mean). The method is based on omission of one member of a sample and recomputing the statistic of interest. This measures the influence of individual observations on the sample and also the confidence in the statistic.

The `Jackknife` class relies on our ability to handle a set of indexes for sub-setting our data and re-computing our statistic. The client code must be able to take a indices and generate a new statistic.

We demo using the jackknife the estimate of mean GC% for an alignment. We first write a factory function to compute the confidence in the mean GC% for an alignment by sampling specific columns.

```
>>> def CalcGc(aln):
...     def calc_gc(indices):
...         new = aln.take_positions(indices)
...         probs = new.get_motif_probs()
...         gc = sum(probs[b] for b in 'CG')
...         total = sum(probs[b] for b in 'ACGT')
...         return gc / total
...     return calc_gc
```

We then create an instance of this factory function with a specific alignment.

```
>>> from cogent3 import LoadSeqs, DNA
>>> aln = LoadSeqs('data/test.paml', moltype=DNA)
>>> calc_gc = CalcGc(aln)
```

We now create a `Jackknife` instance, passing it the `calc_gc` instance we have just made and obtain the sampling statistics. We specify how many elements we're interested in (in this case, the positions in the alignment).

```
>>> from cogent3.maths.stats.jackknife import JackknifeStats
>>> jk = JackknifeStats(len(aln), calc_gc)
>>> print(jk.SampleStat)
0.4766...
>>> print(jk.SummaryStats)
Summary Statistics
================================================
Sample Stat    Jackknife Stat    Standard Error
------------------------------------------------
```

```
     0.4767            0.4767            0.0584
-------------------------------------------------
```

We also display the sub-sample statistics.

```
>>> print(jk.SubSampleStats)
Subsample Stats
============
 i    Stat-i
------------
 0    0.4678
 1    0.4678
 2    0.4847
 3    0.4814...
```

Note: You can provide a custom index generation function that omits groups of observations, for instance. This can be assigned to the `gen_index` argument of the `Jackknife` constructor.

Permutations

Random

```
>>> from numpy.random import permutation as perm
>>> import numpy as np
>>> np.random.seed(153)
>>> arr = np.array(range(5))
>>> for i in range(3):
...     print(perm(arr))
...
[2 1 3 0 4]
[0 3 2 4 1]
[4 0 1 2 3]
```

Ordered

To be written.

Differences in means

Consider a single sample of 50 value:

```
>>> import numpy as np
>>> import cogent3.maths.stats.test as stats
>>> np.random.seed(1357)
>>> nums1 = np.random.normal(loc=45,scale=10,size=50)
```

Although we don't know the population values for the mean and standard deviation for this sample, we can evaluate the probability that the sample could have been drawn from some population with known values, as shown above in Normalizing data (as Z-scores).

If we have a second sample, whose parent population mean and standard deviation are also unknown:

```
>>> nums2 = np.random.normal(loc=50,scale=10,size=50)
```

Suppose we believe (before we see any data) that the mean of the first population is different than the second but we don't know in which direction the change lies, we estimate the standard deviation. We use the standard error of the mean as an estimate for how close the mean of sample 2 is to the mean of its parent population (and vice-versa).

```
>>> mean_nums2 = stats.mean(nums2)
>>> sd_nums2 = stats.std(nums2)
>>> se_nums2 = sd_nums2 / np.sqrt(len(nums2))
>>> se_nums2
1.1113...
>>> mean_nums1 = stats.mean(nums1)
>>> mean_nums1
46.5727...
>>> mean_nums2
50.3825...
>>> mean_nums1 < mean_nums2 - 1.96 * se_nums2
True
```

t-Tests

Small sample sizes can be handled by the use of t-tests. The function `t_two_sample()` is used for two independent samples.

```
>>> subsample1 = nums1[:5]
>>> [str(round(n,2)) for n in subsample1]
['49.25', '38.87', '47.06', '44.49', '43.73']
>>> stats.mean(subsample1)
44.67901...
>>> subsample2 = nums2[:5]
>>> [str(round(n,2)) for n in subsample2]
['51.57', '40.6', '49.62', '46.69', '59.34']
>>> stats.mean(subsample2)
49.56494...
>>> t, prob = stats.t_two_sample(subsample1,subsample2)
>>> t
-1.3835...
>>> prob
0.20388...
```

The two sample means are not significantly different.

If there is one small sample and we want to ask whether it is unlikely to have come from a population with a known mean, use the function `t_one_sample()`

```
>>> import cogent3.maths.stats.test as stats
>>> arr = [52.6, 51.3, 49.8]
>>> t, prob = stats.t_one_sample(arr, popmean=48, tails='high')
>>> t
3.99681...
>>> prob
0.02863...
```

For related samples (pre- and post-treatment), use the function `t_paired()`

```
>>> import cogent3.maths.stats.test as stats
>>> pre =  [52.6, 51.3, 49.8]
>>> post = [62.6, 75.0, 65.2]
>>> t, prob = stats.t_paired(pre, post, tails='low')
>>> t
-4.10781...
>>> prob
0.02723...
```

Sign test

This is essentially just a test using the binomial distribution where the probability of success = 0.5.

```
>>> from cogent3.maths.stats.test import sign_test
>>> sign_test(40, 100)
0.056...
```

Differences in proportions

To be written.

Association

We create some data for testing for association.

```
>>> import numpy as np
>>> np.random.seed(13)
>>> x_nums = range(1,11)
>>> error = [1.5 * random.random() for i in range(len(x_nums))]
>>> error = [e * random.choice((-1,1)) for e in error]
>>> y_nums = [(x * 0.5) + e for x, e in zip(x_nums, error)]
>>> x_array = np.array(x_nums)
>>> y_array = np.array(y_nums)
```

We then compute Kendall's tau and associated probability, which tests the null hypothesis that x and y are not associated.

```
>>> from cogent3.maths.stats.test import kendall_correlation
>>> tau, prob = kendall_correlation(x_array, y_array)
>>> print(tau)
0.688...
>>> print(prob)
0.00468...
```

Correlation

For this example, we generate y-values as one-half the x-value plus a bit of random error

```
>>> import numpy as np
>>> np.random.seed(13)
>>> x_array = np.arange(1,11)
```

```
>>> error = np.random.normal(size=10)
>>> y_array = x_array * 0.5 + error
>>> x_array
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> [str(round(n,2)) for n in y_array]
['-0.21', '1.75', '1.46', '2.45', '3.85', '3.53', '4.85', '4.86', '5.98', '3.95']
```

The function `correlation()` returns the Pearson correlation between x and y, as well as its significance

```
>>> import cogent3.maths.stats.test as stats
>>> r, prob = stats.correlation(x_array, y_array)
>>> r
0.8907...
>>> prob
0.0005...
```

The function `regress()` returns the coefficients to the regression line "y=ax+b"

```
>>> slope, y_intercept = stats.regress(x_array, y_array)
>>> slope
0.5514...
>>> y_intercept
0.2141...
```

Calculate the R^2 value for the regression of x and y

```
>>> R_squared = stats.regress_R2(x_array, y_array)
>>> R_squared
0.7934...
```

And finally, the residual error for each point from the linear regression

```
>>> error = stats.regress_residuals(x_array, y_array)
>>> error = [str(round(e,2)) for e in error]
>>> error
['-0.98', '0.44', '-0.41'...
```

Differences in variances

To be written.

Chi-Squared test

Calculus class data (from Grinstead and Snell, Introduction to Probability). There seems to be a disparity in the number of 'A' grades awarded when broken down by student gender. As input to the function `chi_square_from_Dict2D()` we need a `Dict2D` object containing the observed counts that has been processed by `calc_contingency_expected()` to add the expected counts for each element of the table

`Expected = row_total x column_total / overall_total`

```
>>> from cogent3.util.dict2d import Dict2D
>>> import cogent3.maths.stats.test as stats
>>> F_grades = {'A':37,'B':63,'C':47,'F':5}
>>> M_grades = {'A':56,'B':60,'C':43,'F':8}
>>> grades = {'F':F_grades,'M':M_grades}
```

```
>>> data = Dict2D(grades)
>>> data
{'M': {'A': 56...
>>> OE_data = stats.calc_contingency_expected(data)
>>> OE_data
{'M': {'A': [56, 48.686...
>>> test, chi_high = stats.chi_square_from_Dict2D(OE_data)
>>> test
4.12877...
>>> chi_high
0.24789...
```

Nearly 25% of the time we would expect a Chi-squared statistic as extreme as this one or more (with df = 3), so the result is not significant.

Goodness-of-fit calculation with the same data

```
>>> g_val, prob = stats.G_fit_from_Dict2D(OE_data)
>>> g_val
4.1337592429166437
>>> prob
0.76424854978813872
```

Scatterplots

In this example, we generate the error as above, but separately from the x-value, and subsequently transform using matrix multiplication

```
>>> import random
>>> import numpy as np
>>> import cogent3.maths.stats.test as stats
>>> random.seed(13)
>>> x_nums = list(range(1,11))
>>> error = [1.5 * random.random() for i in range(len(x_nums))]
>>> error = [e * random.choice((-1,1)) for e in error]
>>> arr = np.array(x_nums + error)
>>> arr.shape = (2, len(x_nums))
>>> arr
array([[  1.        ,   2.        ,   3.        ,   4.        ,
          5.        ,   6.        ,   7.        ,   8.        ,
          9.        ,  10.        ],
       [  0.38851274,  -1.02788699,  -1.02612288,  -1.27400424,
          0.27858626,   0.34583791,  -0.22073988,  -0.3377444 ,
         -1.1010354 ,   0.19531953]])
```

We can use a transformation matrix to rotate the points

```
>>> from math import sqrt
>>> z = 1.0/sqrt(2)
>>> t = np.array([[z,-z],[z,z]])
>>> rotated_x, rotated_y = np.dot(t,arr)
```

The plotting code uses matplotlib.

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
```

```
>>> ax = fig.add_subplot(111)
>>> ax.scatter(arr[0],arr[1],s=250,color='b',marker='s')
<matplotlib.collections.RegularPolyCollection object...
>>> ax.scatter(rotated_x,rotated_y,s=250,color='r',marker='o')
<matplotlib.collections.CircleCollection object...
>>> plt.axis('equal')
(0.0, 12.0, -2.0, 8.0)
```

Plot the least squares regression lines too

```
>>> slope, y_intercept = stats.regress(rotated_x, rotated_y)
>>> slope
0.9547989732316251
>>> max_x = 10
>>> ax.plot([0, max_x],[y_intercept, max_x * slope + y_intercept],
...     linewidth=4.0, color='k')
...
[<matplotlib.lines.Line2D object...
>>> slope, y_intercept = stats.regress(arr[0],arr[1])
>>> ax.plot([0, max_x],[y_intercept, max_x * slope + y_intercept],
...     linewidth=4.0, color='0.6')
...
[<matplotlib.lines.Line2D object...
>>> plt.grid(True)
>>> plt.savefig('scatter_example.pdf')
```

(If you want to plot the lines under the points, specify `zorder=n` to the plot commands, where `zorder` for the lines < `zorder` for the points).

Histograms

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> plt.clf()
>>> mu, sigma = 100, 15
>>> x = mu + sigma*np.random.randn(10000)
>>> n, bins, patches = plt.hist(x, 60, normed=1, facecolor='0.75')
```

add a "best fit" line

```
>>> import matplotlib.mlab as mlab
>>> y = mlab.normpdf( bins, mu, sigma)
>>> l = plt.plot(bins, y, 'r--', linewidth=3)
>>> plt.grid(True)
>>> plt.savefig('hist_example.png')
```

Heat Maps

Representing numbers as colors is a powerful data visualization technique. This example does not actually use any functionality from PyCogent3, it simply highlights a convenient matplotlib method for constructing a heat map.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> data = [i * 0.01 for i in range(100)]
```

```
>>> data = np.array(data)
>>> data.shape = (10,10)
```

The plot code

```
>>> fig = plt.figure()
>>> plt.hot()
>>> plt.pcolormesh(data)
<matplotlib.collections.QuadMesh object ...
>>> plt.colorbar()
<matplotlib.colorbar.Colorbar instance ...
>>> plt.savefig('heatmap_example.png')
```

HPC environments

Distribution of tasks across multiple CPUs using `mpi`

Warning: This example requires execution on 2 CPUs. It can be run using: `$ mpirun -n 2 python path/to/parallel_tasks.rst`

All I'm doing here is illustrating the use of `parallel.map` with the simplest example I could come up with. I create a list which will have an integer appended to it – hardly useful, but hopefully demonstrates how a series of data can be mapped onto a function in parallel. In this case, the data is just a list of numbers.

```
>>> from cogent3.util import parallel
>>> passed_indices = []
>>> series = list(range(20))
>>> result = parallel.map(passed_indices.append, series)
>>> assert passed_indices == list(range(0,20,2)) or passed_indices == list(range(1,20,
↪2))
```

The result is either the list of even numbers up to (but not including) 20 or the list of odd numbers.

Distribution of tasks across multiple CPUs using `multiprocess`

Note: Using multiprocess requires no extra commands on invoking the script!

In the `multiprocess` case, `foo()` is called in a *temporary* subprocesses. Communication from a subprocess back into the top process is via the value that `foo()` returns.

I illustrate the use of `parallel.map` here with an example that collects both the process id and the integer.

```
>>> from cogent3.util import parallel
>>> import os, time
>>> parallel.use_multiprocessing(2)
>>> def foo(val):
...     # do some real intensive work in here, which I've simulated by
...     # using sleep
...     time.sleep(0.01)
```

```
...     return os.getpid(), val
>>> data = range(20)
>>> result = parallel.map(foo, data)
```

For the purpose of testing the code is executing correctly, I'll check that there are 2 pid's returned.

```
>>> pids, nums = zip(*result)
>>> len(set(pids)) == 2
True
```

The `result` list is a series of tuples of process id's and the integer, the latter not necessarily being in order.

```
>>> print(result)
[(7332, 0), (7333, 1), (7332, 2), (7333, 3), (7332, 4), (7333, 5), (7332, 6), (7333,␣
↪7), (7332, 8), (7333, 9), (7332, 10), (7333, 11), (7332, 12), (7333, 13), (7332,␣
↪14), (7333, 15), (7332, 16), (7333, 17), (7332, 18), (7333, 19)]
```

Checkpointing optimisation runs

A common problem in HPC systems is to make sure a long running process is capable of restarting after interruptions by restoring to the last check pointed state. The optimiser class code has this capability, for example and we'll illustrate that here. We first construct a likelihood function object.

```
>>> from cogent3 import LoadSeqs, LoadTree
>>> from cogent3.evolve.models import F81
>>> aln = LoadSeqs('data/primate_brca1.fasta')
>>> tree = LoadTree('data/primate_brca1.tree')
>>> sub_model = F81()
>>> lf = sub_model.make_likelihood_function(tree)
>>> lf.set_alignment(aln)
```

We then start an optimisation, providing a filename for checkpointing and specifying a time-interval in (which we make very short here to ensure something get's written, for longer running functions the default `interval` setting is fine). Calling `optimise` then results in the notice that checkpoint's are being written.

```
>>> checkpoint_fn = 'checkpoint_this.txt'
>>> lf.optimise(filename=checkpoint_fn, interval=100, show_progress=False)
CHECKPOINTING to file 'checkpoint_this.txt'...
```

Recovering from a real run that was interrupted generates an additional notification: `RESUMING from file ...` For the purpose of this snippet we just show that the checkpoint file exists.

```
>>> import pickle
>>> data = pickle.load(open(checkpoint_fn, 'rb'))
>>> print(data)
<cogent3.maths.simannealingoptimiser.AnnealingRun object...
```

Checkpointing phylogenetic optimisation runs

The built-in phylogeny code is also capable of checkpointing it's internal state. We illustrate here for the least-squares approach but the same approach also holds for maximum-likelihood. We load some stored distances.

```
>>> import pickle
>>> dists = pickle.load(open('data/dists_for_phylo.pickle', 'rb'))
```

We make the weighted least-squares calculator.

```
>>> from cogent3.phylo import distance, least_squares
>>> ls = least_squares.WLS(dists)
```

We start searching for trees, providing the name of the file to checkpoint to.

```
>>> checkpoint_phylo_fn = 'checkpoint_phylo.txt'
>>> score, tree = ls.trex(a=5, k=1, filename=checkpoint_phylo_fn, interval=100)
```

Useful Utilities

Using PyCogent3's optimisers for your own functions

You have a function that you want to maximise/minimise. The parameters in your function may be bounded (must lie in a specific interval) or not. The `cogent3` optimisers can be applied to these cases. The `Powell` (a local optimiser) and `SimulatedAnnealing` (a global optimiser) classes in particular have had their interfaces standardised for such use cases. We demonstrate for a very simple function below.

We write a simple factory function that uses a provided value for omega to compute the squared deviation from an estimate, then use it to create our optimisable function.

```
>>> import numpy
>>> def DiffOmega(omega):
...     def omega_from_S(S):
...         omega_est = S/(1-numpy.e**(-1*S))
...         return abs(omega-omega_est)**2
...     return omega_from_S
>>> omega = 0.1
>>> f = DiffOmega(omega)
```

We then import the minimise function and use it to minimise the function, obtaining the fit statistic and the associated estimate of S. Note that we provide lower and upper bounds (which are optional) and an initial guess for our parameter of interest (`S`).

```
>>> from cogent3.maths.optimisers import minimise, maximise
>>> S = minimise(f,  # the function
...     xinit=1.0, # the initial value
...     bounds=(-100, 100), # [lower,upper] bounds for the parameter
...     local=True, # just local optimisation, not Simulated Annealing
...     show_progress=False)
>>> assert 0.0 <= f(S) < 1e-6
>>> print('S=%.4f' % S)
S=-3.6150
```

The minimise and maximise functions can also handle multidimensional optimisations, just make xinit (and the bounds) lists rather than scalar values.

Fitting a function to a giving set of x and y values

Giving a set of values for `x` and `y` fit a function `func` that has `n_params` using simplex iterations to minimize the error between the model `func` to fit and the given values. Here we fitting an exponential function.

```
>>> from numpy import array, arange, exp
>>> from numpy.random import rand, seed
>>> from cogent3.maths.fit_function import fit_function
>>> # creating x values
>>> x = arange(-1,1,.01)
>>>
>>> # defining our fitting function
>>> def f(x,a):
...     return exp(a[0]+x*a[1])
...
>>> # getting our real y
>>> y = f(x,a=[2,5])
>>>
>>> # creating our noisy y
>>> y_noise = y + rand(len(y))*5
>>>
>>> # fitting our noisy data to the function using 1 iteration
>>> params = fit_function(x, y_noise, f, 2, 1)
>>> params
array([ 2.0399908 ,  4.96109191])
>>>
>>> # fitting our noisy data to the function using 1 iteration
>>> params = fit_function(x, y_noise, f, 2, 5)
>>> params
array([ 2.0399641 ,  4.96112469])
```

Miscellaneous functions

Identity testing

Basic `identity` function to avoid having to test explicitly for None

```
>>> from cogent3.util.misc import identity
>>> my_var = None
>>> if identity(my_var):
...   print("foo")
... else:
...   print("bar")
...
bar
```

One-line if/else statement

Convenience function for performing one-line if/else statements. This is similar to the C-style ternary operator:

```
>>> from cogent3.util.misc import if_
>>> result = if_(4 > 5, "Expression is True", "Expression is False")
>>> result
'Expression is False'
```

However, the value returned is evaluated, but not called. For instance:

```
>>> from cogent3.util.misc import if_
>>> def foo():
```

```
...     print("in foo")
...
>>> def bar():
...     print("in bar")
...
>>> if_(4 > 5, foo, bar)
<function bar at...
```

Force a variable to be iterable

This support method will force a variable to be an iterable, allowing you to guarantee that the variable will be safe for use in, say, a `for` loop.

```
>>> from cogent3.util.misc import iterable
>>> my_var = 10
>>> for i in my_var:
...     print("will not work")
...
Traceback (most recent call last):
TypeError: 'int' object is not iterable
>>> for i in iterable(my_var):
...     print(i)
...
10
```

Obtain the index of the largest item

To determine the index of the largest item in any iterable container, use `max_index`:

```
>>> from cogent3.util.misc import max_index
>>> l = [5,4,2,2,6,8,0,10,0,5]
>>> max_index(l)
7
```

Note: Will return the lowest index of duplicate max values

Obtain the index of the smallest item

To determine the index of the smallest item in any iterable container, use `min_index`:

```
>>> from cogent3.util.misc import min_index
>>> l = [5,4,2,2,6,8,0,10,0,5]
>>> min_index(l)
6
```

Note: Will return the lowest index of duplicate min values

Remove a nesting level

To flatten a 2-dimensional list, you can use `flatten`:

```
>>> from cogent3.util.misc import flatten
>>> l = ['abcd','efgh','ijkl']
>>> flatten(l)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l']
```

Convert a nested tuple into a list

Conversion of a nested `tuple` into a `list` can be performed using `deep_list`:

```
>>> from cogent3.util.misc import deep_list
>>> t = ((1,2),(3,4),(5,6))
>>> deep_list(t)
[[1, 2], [3, 4], [5, 6]]
```

Simply calling `list` will not convert the nested items:

```
>>> list(t)
[(1, 2), (3, 4), (5, 6)]
```

Convert a nested list into a tuple

Conversion of a nested `list` into a `tuple` can be performed using `deep_list`:

```
>>> from cogent3.util.misc import deep_tuple
>>> l = [[1,2],[3,4],[5,6]]
>>> deep_tuple(l)
((1, 2), (3, 4), (5, 6))
```

Simply calling `tuple` will not convert the nested items:

```
>>> tuple(l)
([1, 2], [3, 4], [5, 6])
```

Testing if an item is between two values

Same as: min <= number <= max, although it is quickly readable within code

```
>>> from cogent3.util.misc import between
>>> between((3,5),4)
True
>>> between((3,5),6)
False
```

Return combinations of items

`Combinate` returns all k-combinations of items. For instance:

```
>>> from cogent3.util.misc import combinate
>>> list(combinate([1,2,3],0))
[[]]
>>> list(combinate([1,2,3],1))
[[1], [2], [3]]
>>> list(combinate([1,2,3],2))
[[1, 2], [1, 3], [2, 3]]
>>> list(combinate([1,2,3],3))
[[1, 2, 3]]
```

Save and load gzip'd files

These handy methods will `pickle` an object and automagically gzip the file. You can also then reload the object at a later date.

```
>>> from cogent3.util.misc import gzip_dump, gzip_load
>>> class foo(object):
...     some_var = 5
...
>>> bar = foo()
>>> bar.some_var = 10
>>> # gzip_dump(bar, 'test_file')
>>> # new_bar = gzip_load('test_file')
>>> # isinstance(new_bar, foo)
```

Note: The above code does work, but pickle won't write out within doctest

Curry a function

curry(f,x)(y) = f(x,y) or = lambda y: f(x,y). This was modified from the Python Cookbook. Docstrings are also carried over.

```
>>> from cogent3.util.misc import curry
>>> def foo(x,y):
...     """Some function"""
...     return x + y
...
>>> bar = curry(foo, 5)
>>> print(bar.__doc__)
 curry(foo,5)
== curried from foo ==
 Some function
>>> bar(10)
15
```

Test to see if an object is iterable

Perform a simple test to see if an object supports iteration

```
>>> from cogent3.util.misc import is_iterable
>>> can_iter = [1,2,3,4]
```

```
>>> cannot_iter = 1.234
>>> is_iterable(can_iter)
True
>>> is_iterable(cannot_iter)
False
```

Test to see if an object is a single char

Perform a simple test to see if an object is a single character

```
>>> from cogent3.util.misc import is_char
>>> class foo:
...   pass
...
>>> is_char('a')
True
>>> is_char('ab')
False
>>> is_char(foo())
False
```

Flatten a deeply nested iterable

To flatten a deeply nested iterable, use `recursive_flatten`. This method supports multiple levels of nesting, and multiple iterable types

```
>>> from cogent3.util.misc import recursive_flatten
>>> l = [[[[1,2], 'abcde'], [5,6]], [7,8], [9,10]]
>>> recursive_flatten(l)
[1, 2, 'a', 'b', 'c', 'd', 'e', 5, 6, 7, 8, 9, 10]
```

Test to determine if `list` of `tuple`

Perform a simple check to see if an object is not a list or a tuple

```
>>> from cogent3.util.misc import not_list_tuple
>>> not_list_tuple(1)
True
>>> not_list_tuple([1])
False
>>> not_list_tuple('ab')
True
```

Unflatten items to row-width

Unflatten an iterable of items to a specified row-width. This does reverse the effect of `zip` as the lists produced are not interleaved.

```
>>> from cogent3.util.misc import unflatten
>>> l = [1,2,3,4,5,6,7,8]
>>> unflatten(l,1)
[[1], [2], [3], [4], [5], [6], [7], [8]]
```

```
>>> unflatten(l,2)
[[1, 2], [3, 4], [5, 6], [7, 8]]
>>> unflatten(l,3)
[[1, 2, 3], [4, 5, 6]]
>>> unflatten(l,4)
[[1, 2, 3, 4], [5, 6, 7, 8]]
```

Unzip items

Reverse the effects of a `zip` method, i.e. produces separate lists from tuples

```
>>> from cogent3.util.misc import unzip
>>> l = ((1,2),(3,4),(5,6))
>>> unzip(l)
[[1, 3, 5], [2, 4, 6]]
```

Select items in order

Select items in a specified order

```
>>> from cogent3.util.misc import select
>>> select('ea', {'a':1,'b':5,'c':2,'d':4,'e':6})
[6, 1]
>>> select([0,4,8], 'abcdefghijklm')
['a', 'e', 'i']
```

Find overlapping pattern occurrences

Find all of the overlapping occurrences of a pattern within a text

```
>>> from cogent3.util.misc import find_all
>>> text = 'aaaaaaa'
>>> pattern = 'aa'
>>> find_all(text, pattern)
[0, 1, 2, 3, 4, 5]
>>> text = 'abababab'
>>> pattern = 'aba'
>>> find_all(text, pattern)
[0, 2, 4]
```

Find multiple pattern occurrences

Find all of the overlapping occurrences of multiple patterns within a text. Returned indices are sorted, each index is the start position of one of the patterns

```
>>> from cogent3.util.misc import find_many
>>> text = 'abababcabab'
>>> patterns = ['ab','abc']
>>> find_many(text, patterns)
[0, 2, 4, 4, 7, 9]
```

Safely remove a trailing underscore

'Unreserve' a mutation of Python reserved words

```
>>> from cogent3.util.misc import unreserve
>>> unreserve('class_')
'class'
>>> unreserve('class')
'class'
```

Create a case-insensitive iterable

Create a case-insensitive object, for instance, if you want the key 'a' and 'A' to point to the same item in a dict

```
>>> from cogent3.util.misc import add_lowercase
>>> d = {'A':5,'B':6,'C':7,'foo':8,42:'life'}
>>> add_lowercase(d)
{'A': 5, 'a': 5, 'C': 7, 'B': 6, 42: 'life', 'c': 7, 'b': 6, 'foo': 8}
```

Extract data delimited by differing left and right delimiters

Extract data from a line that is surrounded by different right/left delimiters

```
>>> from cogent3.util.misc import extract_delimited
>>> line = "abc[def]ghi"
>>> extract_delimited(line,'[',']')
'def'
```

Invert a dictionary

Get a dictionary with the values set as keys and the keys set as values

```
>>> from cogent3.util.misc import InverseDict
>>> d = {'some_key':1,'some_key_2':2}
>>> InverseDict(d)
{1: 'some_key', 2: 'some_key_2'}
```

Note: An arbitrary key will be set if there are multiple keys with the same value

Invert a dictionary with multiple keys having the same value

Get a dictionary with the values set as keys and the keys set as values. Can handle the case where multiple keys point to the same values

```
>>> from cogent3.util.misc import InverseDictMulti
>>> d = {'some_key':1,'some_key_2':1}
>>> InverseDictMulti(d)
{1: ['some_key_2', 'some_key']}
>>>
```

Get mapping from sequence item to all positions

`DictFromPos` returns the positions of all items seen within a sequence. This is useful for obtaining, for instance, nucleotide counts and positions

```
>>> from cogent3.util.misc import DictFromPos
>>> seq = 'aattggttggaaggccgccgttagacg'
>>> DictFromPos(seq)
{'a': [0, 1, 10, 11, 22, 24], 'c': [14, 15, 17, 18, 25], 't': [2, 3, 6, 7, 20, 21], 'g
↪': [4, 5, 8, 9, 12, 13, 16, 19, 23, 26]}
```

Get the first index of occurrence for each item in a sequence

`DictFromFirst` will return the first location of each item in a sequence

```
>>> from cogent3.util.misc import DictFromFirst
>>> seq = 'aattggttggaaggccgccgttagacg'
>>> DictFromFirst(seq)
{'a': 0, 'c': 14, 't': 2, 'g': 4}
```

Get the last index of occurrence for each item in a sequence

`DictFromLast` will return the last location of each item in a sequence

```
>>> from cogent3.util.misc import DictFromLast
>>> seq = 'aattggttggaaggccgccgttagacg'
>>> DictFromLast(seq)
{'a': 24, 'c': 25, 't': 21, 'g': 26}
```

Construct a distance matrix lookup function

Automatically construct a distance matrix lookup function. This is useful for maintaining flexibility about whether a function is being computed or if a lookup is being used

```
>>> from cogent3.util.misc import DistanceFromMatrix
>>> from numpy import array
>>> m = array([[1,2,3],[4,5,6],[7,8,9]])
>>> f = DistanceFromMatrix(m)
>>> f(0,0)
1
>>> f(1,2)
6
```

Get all pairs from groups

Get all of the pairs of items present in a list of groups. A key will be created (i,j) iff i and j share a group

```
>>> from cogent3.util.misc import PairsFromGroups
>>> groups = ['ab','xyz']
>>> PairsFromGroups(groups)
{('a', 'a'): None, ('b', 'b'): None, ('b', 'a'): None, ('x', 'y'): None, ('z', 'x'):␣
↪None, ('y', 'y'): None, ('x', 'x'): None, ('y', 'x'): None, ('z', 'y'): None, ('x',
↪'z'): None, ('a', 'b'): None, ('y', 'z'): None, ('z', 'z'): None}
```

Check class types

Check an object against base classes or derived classes to see if it is acceptable

```
>>> from cogent3.util.misc import ClassChecker
>>> class not_okay(object):
...   pass
...
>>> no = not_okay()
>>> class okay(object):
...   pass
...
>>> o = okay()
>>> class my_dict(dict):
...   pass
...
>>> md = my_dict()
>>> cc = ClassChecker(str, okay, dict)
>>> o in cc
True
>>> no in cc
False
>>> 5 in cc
False
>>> {'a':5} in cc
True
>>> 'asasas' in cc
True
>>> md in cc
True
```

Delegate to a separate object

Delegate object method calls, properties and variables to the appropriate object. Useful to combine multiple objects together while assuring that the calls will go to the correct object.

```
>>> from cogent3.util.misc import Delegator
>>> class ListAndString(list, Delegator):
...   def __init__(self, items, string):
...     Delegator.__init__(self, string)
...     for i in items:
...       self.append(i)
...
>>> ls = ListAndString([1,2,3], 'ab_cd')
>>> len(ls)
3
>>> ls[0]
1
>>> ls.upper()
'AB_CD'
>>> ls.split('_')
['ab', 'cd']
```

Wrap a function to hide from a class

Wrap a function to hide it from a class so that it isn't a method.

```
>>> from cogent3.util.misc import FunctionWrapper
>>> f = FunctionWrapper(str)
>>> f
<cogent3.util.misc.FunctionWrapper object at ...
>>> f(123)
'123'
```

Construct a constrained container

Wrap a container with a constraint. This is useful for enforcing that the data contained is valid within a defined context. PyCogent3 provides a base `ConstrainedContainer` which can be used to construct user-defined constrained objects. PyCogent3 also provides `ConstrainedString`, `ConstrainedList`, and `ConstrainedDict`. These provided types fully cover the builtin types while staying integrated with the `ConstrainedContainer`.

Here is a light example of the `ConstrainedDict`

```
>>> from cogent3.util.misc import ConstrainedDict
>>> d = ConstrainedDict({'a':1,'b':2,'c':3}, constraint='abc')
>>> d
{'a': 1, 'c': 3, 'b': 2}
>>> d['d'] = 5
Traceback (most recent call last):
ConstraintError: Item 'd' not in constraint 'abc'
```

PyCogent3 also provides mapped constrained containers for each of the default types provided, `MappedString`, `MappedList`, and `MappedDict`. These behave the same, except that they map a mask onto `__contains__` and `__getitem__`

```
>>> def mask(x):
...     return str(int(x) + 3)
...
>>> from cogent3.util.misc import MappedString
>>> s = MappedString('12345', constraint='45678', mask=mask)
>>> s
'45678'
>>> s + '123'
'45678456'
>>> s + '9'
Traceback (most recent call last):
ConstraintError: Sequence '9' doesn't meet constraint
```

Check the location of an application

Determine if an application is available on a system

```
>>> from cogent3.util.misc import app_path
>>> app_path('ls')
'/bin/ls'
>>> app_path('does_not_exist')
False
```

CHAPTER 7

For Developers

Anyone can contribute to the development of PyCogent3, not just registered developers. If you figure out a solution to something using PyCogent3 that you'd like to share, or if you have ideas for improving the current documentation, please consider forking and submitting a pull request and we'll look into including it in PyCogent3!

Grabbing from Bitbucket

To grab PyCogent3 from Bitbucket, do the following:

```
$ hg clone ssh://hg@bitbucket.org/pycogent3/pycogent3 PyCogent3
```

Building/testing the documentation

To build the documentation or `doctest` the contents, you'll need to install Sphinx:

```
$ pip install sphinx
```

Generating the html form of the documentation requires changing into the `doc` directory and executing a `make` command:

```
$ cd path/to/PyCogent3/doc
$ make html
... # bunch of output
Build finished. The HTML pages are in _build/html.
```

The index file czan be found in `PyCogent3/doc/_build/html/index.html` is the root of the documentation.

You can also generate a pdf file, using the Sphinx latex generation capacity. This is slightly more involved. (It also requires that you have an installation of TeTex.)

First generate the latex

```
$ make latex
... # bunch of output
Build finished; the LaTeX files are in _build/latex.
Run `make all-pdf' or `make all-ps' in that directory to run these through (pdf)latex.
```

then change into the `latex` dir and build the pdf

```
$ cd _build/latex
$ make all-pdf
```

You can now open `PyCogent3.pdf`.

To actually test the documentation, you need to be in the `doc` directory and then execute another `make` command:

```
$ cd path/to/PyCogent3/doc
$ make doctest
```

The results are in `_build/doctest/output.txt`.

Note: The documentation does not test for presence of 3rd party dependencies (such as applications or python modules) like the PyCogent3 `unittest` test suite. If you don't have all the 3rd party applications installed you will see failures. At this point **no effort** is being expended to hide such failures.

Adding to the documentation

You can maximise the cogent3 user experience for yourself and others by contributing to the documentation. If you solve a problem that you think might prove useful to others then fork, add it into the documentation and do a pull request. If you can think of ways to improve the existing documents let us know via a ticket.

For guidance on adding documentation, look at any of the existing examples. The restructured text format is pretty easy to write (for overview see the Sphinx rest overview). The conventions adopted by PyCogent3 are to use heading levels to be consistent with the Python.org standard (taken from Sphinx headings). They are

- # with overline, for parts
- * with overline, for chapters
- =, for sections
- -, for subsections
- ^, for subsubsections
- ", for paragraphs
- +, added for sub-paragraphs (non-standard)

If it's a use-case, create your file in the `examples` directory, giving it a `.rst` suffix. Link it into the documentation tree, adding a line into the `examples/index.rst` file. If it's something you think should be added into the cookbook, add it into the appropriate cookbook document.

The new documentation checklist

Things you should check before committing your new document:

- Add a line at the beginning with yourself as author (`.. sectionauthor:: My Name`) so people can contact you with feedback.
- Add any data files used in your documentation under `PyCogent3/doc/data/`
- Add a download link to those files to `PyCogent3/doc/data_file_links.rst` following the style employed in that file.
- Spellcheck!!
- Check what you wrote is valid restructured text by building the documents for both html and latex. If your document isn't connected into the table of contents, Sphinx will print a warning to screen.
- Check you have correctly marked up the content and that it looks OK. Make sure that python code and shell commands are correctly highlighted and that literals are marked up as literals. In particular, check the latex build since it is common for text to span beyond the page margins. If the latter happens, revise your document!
- Check that it works (rather than testing the entire suite, you can use the convenience script within doc). For instance, the following is a single test of one file:

```
$ cd path/to/PyCogent3/doc
$ python doctest_rsts.py examples/reverse_complement.rst
```

Adding TODOs

Add todo's into the rst files using the `todo` directive as in

```
.. todo::

    some task
```

To see the list of todo's in the project, uncomment the line that sets `todo_include_todos=True` in `doc/conf.py`, then cd into the `doc/` and make the html docs again. The todo's are listed on the main page.

Warning: Be sure to revert the conf.py file back to it's original state so you don't accidentally commit the change as this affects everyone else's documentation too!

Developing C-extensions

Extensions for PyCogent3 should be written in Cython.

If you have any questions, contact Gavin.

CHAPTER 8

Licenses and disclaimer

PyCogent3 License

```
Copyright (c) 2016, PyCogent3 Development Team
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this
  list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice,
  this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

* Neither the name of PyCogent3, cogent3 nor the names of its
  contributors may be used to endorse or promote products derived from
  this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

PyCogent3 is released under the BSD license, a copy of which is included in the distribution (see *PyCogent3 License*). Licenses for other code sources are left in place.

CHAPTER 9

Change log

The porting from PyCogent 1.9 to 3.0 has involved a massive number of changes!

Please see the wiki pages for a summary of the API changes.

But the best way to see them is all to use mercurial.

CHAPTER 10

WARNING & DISCLAIMER

Be warned that PyCogent3 is a significantly changed library from the original PyCogent. Most of these changes involve elimination of modules, conversion to PEP8 compliance and other refactors. These changes are still underway!

Please see the wiki pages for a detailed changelog on the API changes.

CHAPTER 11

YOU CAN HELP!

Posting Bugs

If you discover a bug, especially something that worked in PyCogent but not in PyCogent3, please post a ticket!

When posting a ticket, please provide a minimum working example to reproduce the issue. Also include the versions of the library and other tools (e.g. attach the result of `$ pip freeze > libs.txt`).

Getting Involved

We would greatly appreciate assistance in updating the project to PEP8, or anything else you think needs doing.

Please post a ticket, or comment on an existing one, indicating your intention so we can assist. Then it's the usual "fork", "pull request" dance.

CHAPTER 12

Overview

PyCogent3 is a software library for genomic biology. It is a fully integrated and thoroughly tested framework for: conducting novel probabilistic analyses of biological sequence evolution; and generating publication quality graphics. It is distinguished by many unique built-in capabilities (such as true codon alignment) and the frequent addition of entirely new methods for the analysis of genomic data.

Our primary goal is to provide a collection of rigourously validated tools for the manipulation and analysis of genome biology data sets.

CHAPTER 13

Support

We acknowledge the provision by Wingware of free licenses for their professional IDE. Their committment, over more than a decade, to supporting our development of Open Source software for science is greatly appreciated. (The port of PyCogent to Python 3 was made much easier by using Wing's refactor tools!)

CHAPTER 14

Citation

If you use this software for published work please cite – Knight et al., 2007, Genome Biol, 8, R171.

CHAPTER 15

Search

- search

CHAPTER 16

Support

We acknowledge the provision by Wingware of free licenses for their professional IDE. Their committment, over more than a decade, to supporting our development of Open Source software for science is greatly appreciated. (The port of PyCogent to Python 3 was made much easier by using Wing's refactor tools!)

CHAPTER 17

News and Announcements

PyCogent3 News and Announcements are available via the News and Announcements Blog at http://pycogent.wordpress.com.

Subscribe to the News and Annoucements RSS Feed

Index

C