Phylogenetic Computing With DendroPy

A Special "Tour"-torial

Jeet Sukumaran

Department of Ecology and Evolutionary Biology
University of Michigan
Ann Arbor, MI

jeetsukumaran@gmail.com

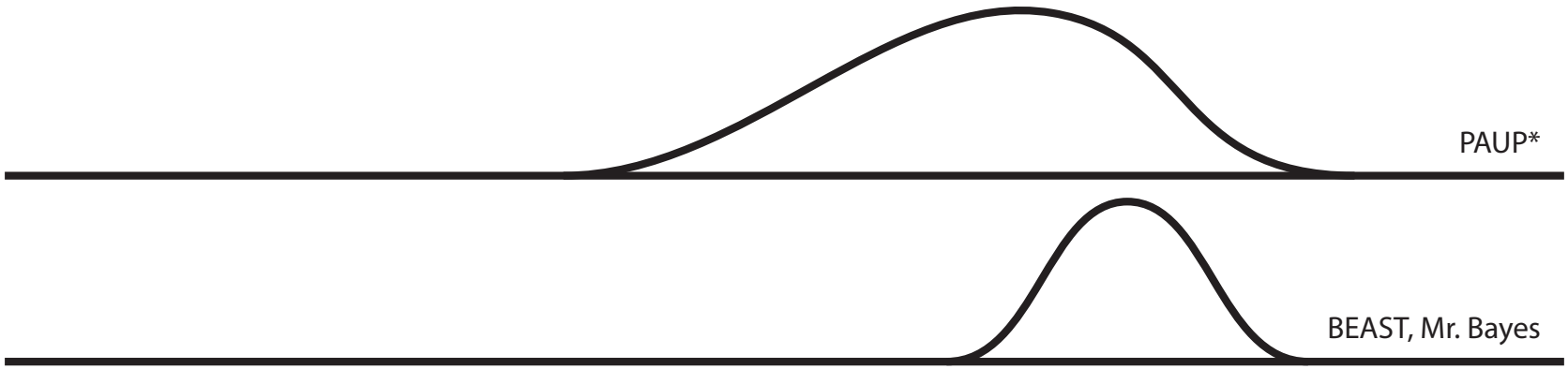
Society Of Systematic Biologists Standalone Meeting
2017, Baton Rouge, LA

The Tour Itinerary

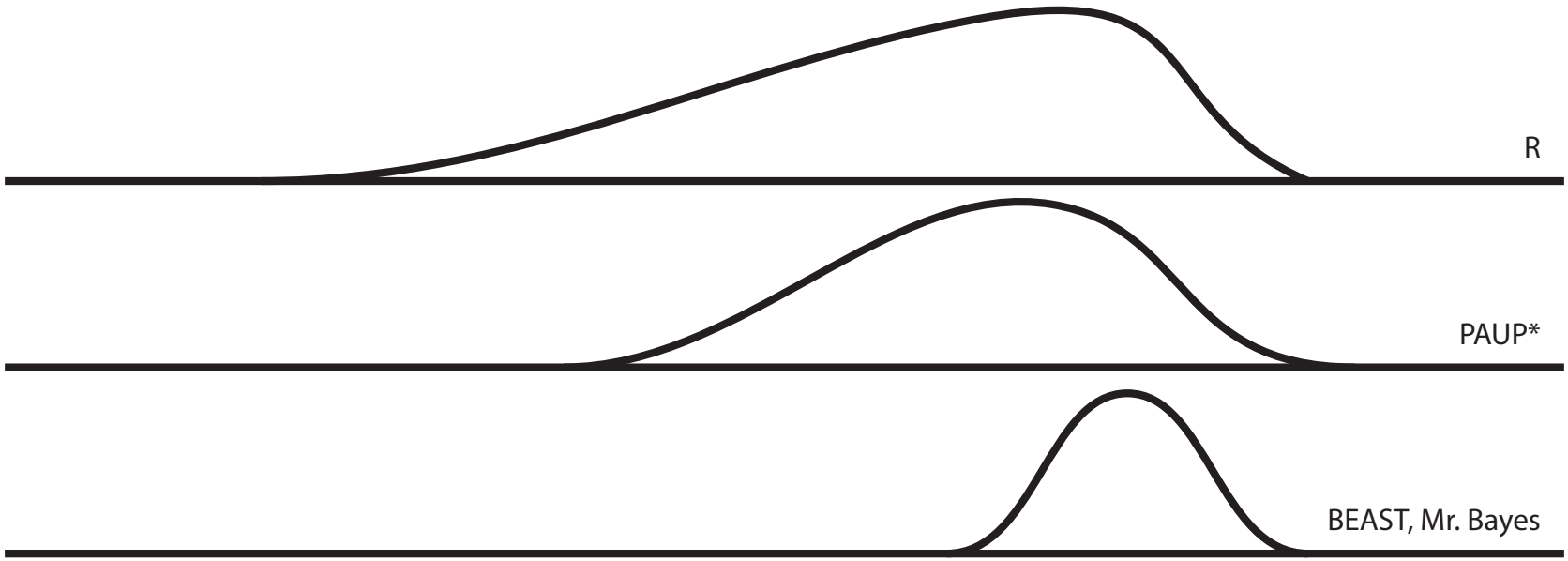

Programmer Role
Power User Role
End User Role


BEAST, Mr. Bayes

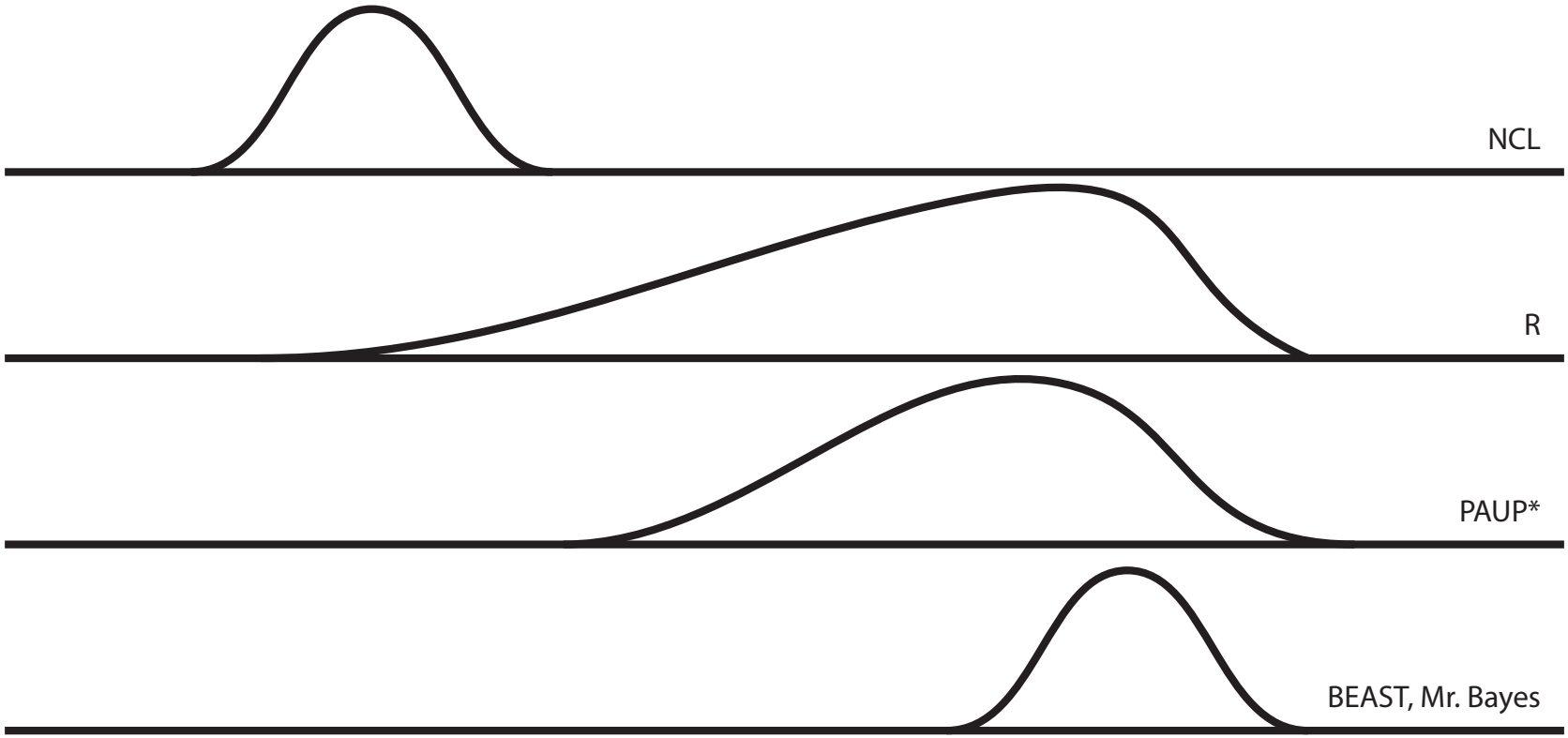

Programmer Role
Power User Role
End User Role



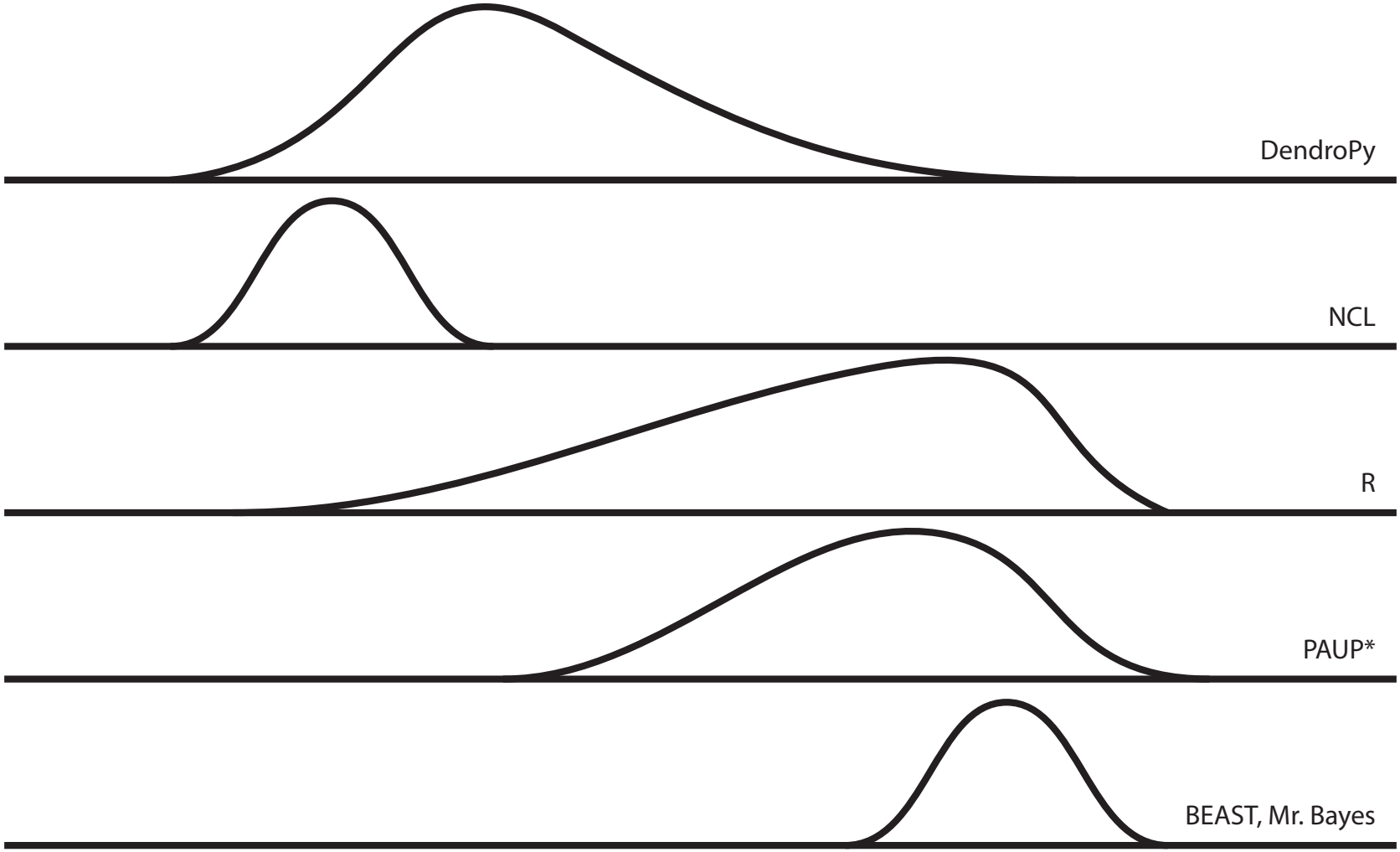
Programmer Role	Power User Role	End User Role



Programmer Role	Power User Role	End User Role



Programmer Role	Power User Role	End User Role



Programmer Role	Power User Role	End User Role

DendroPy: Design Objectives/Criteria

- Expressiveness
- Functionality
- Correctness
- Robustness
- Portability
- "Stay-out-of-your-way" extensibility/usage

DendroPy: Library Components

- Data layer
- I/O layer
- Low-level calculations/manipulation layer
- Mid-level calculations/operations layer
- High-level functions

DendroPy: Library Components

- Data layer
  - Phylogenetic data objects: Rich, fully object-oriented phylogenetic data model.
  - Taxon namespaces, trees, tree lists, alignments, etc.
- I/O layer
- Low-level calculations/manipulation layer
- Mid-level calculations/operations layer
- High-level functions

DendroPy: Library Components

- Data layer
- I/O layer
  - Parsers and writers allow for seamless serialization/deserialization of phylogenetic data objects in a variety of different formats.
  - NEXUS, Newick, NeXML, FASTA, Phylip, etc.
- Low-level calculations/manipulation layer
- Mid-level calculations/operations layer
- High-level functions

DendroPy: Library Components

- Data layer
- I/O layer
- Low-level calculations/manipulation layer
  - Various "under-the-hood" crunching and processing routines that provide both basic and advanced "building blocks" for sophisticated operations.
  - Splits, splits hashes, branch- and node-based operations, metadata annotations, rooting/re-rooting/de-rooting trees, etc.
- Mid-level calculations/operations layer
- High-level functions

DendroPy: Library Components

- Data layer
- I/O layer
- Low-level calculations/manipulation layer
- Mid-level calculations/operations layer
  - Simulation and calculations under range of models and approaches.
  - Coalescent, multispecies coalescent, birth-death, protracted speciation, etc.
- High-level functions

DendroPy: Library Components

- Data layer
- I/O layer
- Low-level calculations/manipulation layer
- Mid-level calculations/operations layer
- High-level functions
    - Summarization of MCMC/bootstrap output, calculation of split support, etc.

DendroPy Phylogenetic Data Objects

A rich object-oriented data model for the expression and manipulation of phylogenetic data:

- `Taxon`
- `TaxonNamespace`
- `Tree`
- `TreeList`
- `CharacterMatrix`
  - `DnaCharacterMatrix`
  - `RnaCharacterMatrix`
  - `NucleotideCharacterMatrix`
  - `ProteinCharacterMatrix`
  - `StandardCharacterMatrix`
  - `ContinuousCharacterMatrix`
  - `RestrictionSitesCharacterMatrix`
  - `InfiniteSitesCharacterMatrix`
- `DataSet`

The `Taxon` Class

- Represents a distinct operational taxonomic unit (OTU) concept.
- Has an attribute, "`label`", that tracks its name as a string.
- **NOTE:** The `Taxon` object is not defined by nor equivalent to its string label!
- Which allows for:
  - Changing the "`label`" of a `Taxon` object while preserving all data semantics and associations (*very* useful!).
  - Multiple `Taxon` objects having the same label (Not so useful?).

```
import dendropy
tns = dendropy.TaxonNamespace()
t1 = tns.new_taxon(label="A")
print(t1)
print(repr(t1))
print(t1.label)
print(t1 == "A")
print(t1.label == "A")
print(type(t1) == type("A"))
```

Output

```
'A'
<Taxon 0x102b9fc50 'A'>
A
False
True
False
```

The `TaxonNamespace` Class

- Represents a *collection* of distinct operational taxonomic unit (OTU) concepts.
- Defines a self-contained universe of taxon names that each map to a distinct operational taxonomic unit concepts.
- Allows for the establishment of identity of OTU's across multiple different data sources.
- Management of taxonomic namespace is critical to correct behavior when dealing with multiple data sources.
- All factory and reader methods support a "`taxon_namespace`" keyword argument that takes a `TaxonNamespace` object as a value: use this to ensure all your data are instantiated in the same namespace.

The `Tree` Class

- Represents an *arborescence*: a fully-connected directed acyclic graph, with nodes (represented by `Node` objects) in unique parent-child relationships with one another.
- Each node as a "`edge`" attribute which has a `Edge` object as a value, managing the state of the edge subtending the node.
- Important `Tree` attributes include:
  - "`is_rooted`": the rooting state of the tree (`True`, `False`, or `None`).
  - "`seed_node`": the initial ("head") `Node` object of the tree; this is equivalent to the root node if the tree is rooted.
  - "`taxon_namespace`": the taxonomic namespace of the OTU's associated with the tree (not all OTU's in the namespace may be on the tree).
- All nodes on a tree, whether leaf or internal, have two *optional* attributes: `label` (typically a string value or `None`) and `taxon` (typically a `Taxon` object value or `None`).

The `Tree` Class

- Very extensive high-, mid-, and low-level methods for tree operations, manipulations, calculations, statistics, etc.
- Various methods of tree-traversal: `preorder_node_iter`, `postorder_node_iter`, `preorder_edge_iter`, `postorder_edge_iter`, `leaf_node_iter`, `internal_node_iter`, and many, many, more.
- Various methods for finding, filtering, and extracting, pruning, rotating, rooting or re-rooting, "slicing", etc.
- Various methods for collecting data and basic metrics: node ages, edge lengths, etc.
- Entire subsystem for calculating, maintaining, correlating, and comparing split identities across trees (c.f. the `Bipartition` object, the `bipartition_encoding` method, etc.).

```
#! /usr/bin/env python

import random
import dendropy

tree_str = """[&r](((a,b),(c,d),e));"""

tree1 = dendropy.Tree.get(
            data=tree_str,
            schema="newick",)
for taxon in tree1.taxon_namespace:
    taxon.label = "Taxon{}".format(taxon.label.upper())
for nd in tree1:
    nd.edge.length = random.expovariate(0.02)
    nd.edge.annotations["color"] = random.choice(["red",
        "blue",
        "green"])
print(tree1.as_string("nexus"))
```

Output

```
#NEXUS

BEGIN TAXA;
    DIMENSIONS NTAX=5;
    TAXLABELS
        TaxonA
        TaxonB
        TaxonC
        TaxonD
        TaxonE
  ;
END;

BEGIN TREES;
    TREE 1 = [&R] (((TaxonA:90.077259372[&color=red],TaxonB:16.2389828671[&color=red])
         :1.41373919661[&color=green],(TaxonC:107.013409825[&color=red],TaxonD:38.4387565978[&
         color=blue]):117.357980588[&color=green],TaxonE:1.05215461815[&color=green])
         :45.0969034841[&color=blue]):20.0400573218[&color=red];
END;
```

The `TreeList` Class

- Represents a *collection* of `Tree` objects sharing the same taxonomic namespace.
- Supports a complete `list` interface: behaves and can be treated exactly like a list, with additional features for taxonomic namespace management when trees are added.

```
import dendropy
from dendropy.calculate import
    treecompare
trees = dendropy.TreeList.get(
        path="pythonidae.random.bd0301.
    tre",
        schema="nexus")
for tree in trees:
    print(tree.as_string("newick"))
print(len(trees))
print(trees[4].as_string("nexus"))
first_10_trees = trees[:10]
last_10_trees = trees[-10:]
trees[4] = trees[5]
trees.remove(trees[-1])
tx = trees.pop()
trees.sort(key=lambda t:t.label)
trees.reverse()
trees.clear()
```

The `CharacterMatrix` Family of Classes

- Set of related classes to deal with character data (aligned or unaligned).
- Specialized for a variety of character types.
- Supports a `dict` interface, with taxa as keys and sequences as values.
- Supports annotations, column types, subsetting, concatenation, etc.

```
import dendropy
dna = dendropy.DnaCharacterMatrix.get(
        path="primates.chars.nexus",
        schema="nexus")
s1 = dna["Macaca sylvanus"]
# access by taxon index
s2 = dna[0]
s4 = dna[-2]
# access by taxon instance
t = dna.taxon_namespace.get_taxon(label=
    "Macaca sylvanus")
s5 = dna[t]
# iterate over taxa
for taxon in dna:
    print("{}: {}".format(taxon.label,
    dna[taxon]))
# iterate over the sequences
for seq in dna.values():
    print(seq)
# iterate over taxon/sequence pairs
for taxon, seq in dna.items():
    print("{}: {}".format(taxon.label,
    seq))
```

The DataSet Class

- Collection of mixed data types: taxon namespaces, trees, and character matrices.
- In many cases, unless the data store is mixed, working with specialized collections such as TreeList, DnaCharacterMatrix, etc. is more efficient and straightforward.

```
import dendropy
# read a mixed data document
ds = dendropy.DataSet.get(
    path="geospiza.nex",
    schema="nexus",)
# how many char matrices in the data?
print(len(ds.char_matrices))
# how many tree collections in the data?
print(len(ds.tree_lists))
# iterate through the char matrices
# printing the type of data
for chm in ds.char_matrices:
    print(chm.data_type)
# iterate through the trees, printing
# a diagram
for tree_list in ds.tree_lists:
    for tree in tree_list:
        print(tree.as_ascii_plot())
# add another tree collection to the
    dataset
ds.read(path="geospiza.tre", schema="
    newick")
```

The `get()` Factory Method

- The `get()` class-method is a factory function that creates an object of the given class from a data source.
- This method takes, at a minumum, two keyword arguments that specify the *source* of the data and the *schema* (or format) of the data.

```
tree1 = dendropy.Tree.get(
    path="pythonidae.tre",
    schema="newick")
tree2 = dendropy.Tree.get(
    data="((A,B),(C,D))",
    schema="newick")
trees = dendropy.TreeList.get(
    file=open("primates.mcmc.nex",
    schema="nexus")
dna = dendropy.DnaCharacterMatrix.get(
    url="http://.../...",
    schema="nexus")
aa = dendropy.ProteinCharacterMatrix.get(
    url="http://.../...",
    schema="fasta")
cc = dendropy.ContinuousCharacterMatrix.get(
    path="owlmetrics.dat",
    schema="phylip")
dataset = dendropy.DataSet.get(
    path="project1.xml",
    schema="nexml")
```

The `get()` Factory Method

- Source keyword argument:
  - "`path`" (string filepath)
  - "`file`" (file-like object or stream)
  - "`data`" (contents given directly as string)
  - "`url`" (URL of online data)
- Format specified by "`schema`" keyword: "nexus", "newick", "phylip", "fasta", etc.

```
tree1 = dendropy.Tree.get(
    path="pythonidae.tre",
    schema="newick")
tree2 = dendropy.Tree.get(
    data="((A,B),(C,D))",
    schema="newick")
trees = dendropy.TreeList.get(
    file=open("primates.mcmc.nex",
    schema="nexus")
dna = dendropy.DnaCharacterMatrix.get(
    url="http://.../...",
    schema="nexus")
aa = dendropy.ProteinCharacterMatrix.get(
    url="http://.../...",
    schema="fasta")
cc = dendropy.ContinuousCharacterMatrix.get(
    path="owlmetrics.dat",
    schema="phylip")
dataset = dendropy.DataSet.get(
    path="project1.xml",
    schema="nexml")
```

The `get()` Factory Method

- Many other optional keyword arguments to control various aspects of the parsing.
- Some are general (e.g., the all-important "`taxon_namespace`") argument.
- Others are format-specific (e.g., treatment of underscores in NEXUS and Newick labels).

```
tree = dendropy.Tree.get(
    path="tree.tre",
    schema="newick",
    label=None,
    taxon_namespace=None,
    collection_offset=None,
    tree_offset=None,
    rooting="default-unrooted",
    edge_length_type=float,
    suppress_edge_lengths=False,
    extract_comment_metadata=True,
    store_tree_weights=False,
    encode_splits=False,
    finish_node_fn=None,
    case_sensitive_taxon_labels=False,
    preserve_underscores=False,
    suppress_internal_node_taxa=True,
    suppress_leaf_node_taxa=False,
    terminating_semicolon_required=True,
    ignore_unrecognized_keyword_arguments=
    False,
    )
```

The `get()` Factory Method

- Many other optional keyword arguments to control various aspects of the parsing.
- Some are general (e.g., the all-important "`taxon_namespace`") argument.
- Others are format-specific (e.g., treatment of underscores in NEXUS and Newick labels).

```
d = dendropy.DnaCharacterMatrix.get(
    path="data.fas",
    schema="fasta",
    label=None,
    taxon_namespace=None,
    ignore_unrecognized_keyword_arguments=
     False,
)
```

The `get()` Factory Method

- Many other optional keyword arguments to control various aspects of the parsing.
- Some are general (e.g., the all-important "`taxon_namespace`") argument.
- Others are format-specific (e.g., treatment of underscores in NEXUS and Newick labels).

```
tree_list = dendropy.TreeList.get(
    path="path/to/file",
    schema="nexus",
    label=None,
    taxon_namespace=None,
    collection_offset=None,
    tree_offset=None,
    rooting="default-unrooted",
    edge_length_type=float,
    suppress_edge_lengths=False,
    extract_comment_metadata=True,
    store_tree_weights=False,
    encode_splits=False,
    finish_node_fn=None,
    case_sensitive_taxon_labels=False,
    preserve_underscores=False,
    suppress_internal_node_taxa=True,
    suppress_leaf_node_taxa=False,
    terminating_semicolon_required=True,
    ignore_unrecognized_keyword_arguments=
    False,
    )
```

The `get()` Factory Method

- Many other optional keyword arguments to control various aspects of the parsing.
- Some are general (e.g., the all-important "`taxon_namespace`") argument.
- Others are format-specific (e.g., treatment of underscores in NEXUS and Newick labels).

```
data_set = dendropy.DataSet.get(
    path="path/to/file",
    schema="nexus",
    label=None,
    taxon_namespace=None,
    exclude_chars=False,
    exclude_trees=False,
    rooting="default-unrooted",
    edge_length_type=float,
    suppress_edge_lengths=False,
    extract_comment_metadata=True,
    store_tree_weights=False,
    encode_splits=False,
    finish_node_fn=None,
    case_sensitive_taxon_labels=False,
    preserve_underscores=False,
    suppress_internal_node_taxa=True,
    suppress_leaf_node_taxa=False,
    terminating_semicolon_required=True,
    ignore_unrecognized_keyword_arguments=
    False,
    )
```

The `get()` Factory Method

- Many other optional keyword arguments to control various aspects of the parsing.
- Some are general (e.g., the all-important "`taxon_namespace`") argument.
- Others are format-specific (e.g., treatment of underscores in NEXUS and Newick labels).

```
data_set = dendropy.DataSet.get(
    path="path/to/file",
    schema="nexml",
    label=None,
    taxon_namespace=None,
    exclude_chars=False,
    exclude_trees=False,
    default_namespace="http://www.nexml.org
    /2009",
    case_sensitive_taxon_labels=False,
    ignore_unrecognized_keyword_arguments=
    False,
    )
```

The `read()` Method

The `get()` and `read()` methods have similar signatures and behaviors, but are semantically very different:

- The `get()` method is a *class factory* method that creates and returns a *new* object based on a data source.
- The `read()` method is an *instance* method that *adds* data to an existing object from a data source.

```
import dendropy

# create a new (initally empty) tree collection
trees = dendropy.TreeList()

# Read in multiple set of trees into SAME
# tree collection
# (using ``offset`` to skip burn-in of initial 500)
trees.read(path="mcmc.run1", schema="nexus", tree_offset=500,)
trees.read(path="mcmc.run2", schema="nexus", tree_offset=500,)
trees.read(path="mcmc.run3", schema="nexus", tree_offset=500,)
trees.read(path="mcmc.run4", schema="nexus", tree_offset=500,)
```

The `write()` Method

The `write()` method is an instance method that is the serialization counterpart to the `get()` method, taking:

- Mandatory keyword argument specifying the destination.
- Mandatory "schema" keyword argument specifying the format.
- Optional keyword arguments to control the output.

```
trees.write(
  path='outputfile.tre',
  schema='newick',
  suppress_leaf_taxon_labels=False,
  suppress_leaf_node_labels=True,
  suppress_internal_taxon_labels=False,
  suppress_internal_node_labels=False,
  suppress_rooting=False,
  suppress_edge_lengths=False,
  unquoted_underscores=False,
  preserve_spaces=False,
  store_tree_weights=False,
  taxon_token_map=None,
  suppress_annotations=True,
  annotations_as_nhx=False,
  suppress_item_comments=True,
  node_label_element_separator=' ',
  node_label_compose_fn=None,
  edge_label_compose_fn=None,
  real_value_format_specifier='',
  ignore_unrecognized_keyword_arguments=
    False,
  )
```

The `write()` Method

The `write()` method is an instance method that is the serialization counterpart to the `get()` method, taking:

- Mandatory keyword argument specifying the destination.
- Mandatory "schema" keyword argument specifying the format.
- Optional keyword arguments to control the output.

```
data.write(
  path='dataset.nex',
  schema='nexus',
  simple=False,
  suppress_taxa_blocks=None,
  suppress_unreferenced_taxon_namespaces=
    False,
  suppress_block_titles=None,
  file_comments=[],
  preamble_blocks=[],
  supplemental_blocks=[],
  allow_multiline_comments=True,
  continuous_character_state_value_format_fn
    =None,
  discrete_character_state_value_format_fn=
    None,
  suppress_leaf_taxon_labels=False,
  suppress_leaf_node_labels=True,
  suppress_internal_taxon_labels=False,
  suppress_internal_node_labels=False,
  suppress_rooting=False,
  suppress_edge_lengths=False,
  unquoted_underscores=False,
```

The `write()` Method

The `write()` method is an instance method that is the serialization counterpart to the `get()` method, taking:

- Mandatory keyword argument specifying the destination.
- Mandatory "schema" keyword argument specifying the format.
- Optional keyword arguments to control the output.

```
chars.write(
  file=(open("data.phylip","w")),
  schema="phylip",
  strict=False,
  spaces_to_underscores=False,
  force_unique_taxon_labels=False,
  suppress_missing_taxa=False,
  ignore_unrecognized_keyword_arguments=
    False,
  )
```

The `write()` Method

The `write()` method is an instance method that is the serialization counterpart to the `get()` method, taking:

- Mandatory keyword argument specifying the destination.
- Mandatory "schema" keyword argument specifying the format.
- Optional keyword arguments to control the output.

```
dna.write(
  path="data.fas",
  schema="fasta",
  wrap=True,
  ignore_unrecognized_keyword_arguments=
    False,
  )
```

Metadata Annotations

DendroPy supports a rich metadata system with serialization/deserialization in NeXML and various NEXUS (NHX, *BEAST*, or simple comments) formats.

```
#! /usr/bin/env python
import dendropy
import random
data="[&R]((A[&color='red'],(B[&color='blue'],C[&color='green'])[&
    support=0.75]));"
tree = dendropy.Tree.get(
        data=data,
        schema="newick",
        )
for nd in tree:
    if nd.is_leaf():
        print(nd.annotations["color"])
    else:
        print(nd.annotations["support"])
    nd.annotations.drop()
    nd.annotations["fruit"] = random.choice([
        "apple", "mango", "guava"])
print(tree.as_string("nexus"))
```

Metadata Annotations

DendroPy supports a rich metadata system with serialization/deserialization in NeXML and various NEXUS (NHX, *BEAST*, or simple comments) formats.

```
support=''
support=''
color=''red''
support='0.75'
color=''blue''
color=''green''
#NEXUS
BEGIN TAXA;
    DIMENSIONS NTAX=3;
    TAXLABELS
        A
        B
        C
  ;
END;
BEGIN TREES;
    TREE 1 = [&R] ((A[&fruit=apple],(B[&fruit=mango],C[&fruit=apple])
    [&fruit=mango])[&fruit=mango])[&fruit=mango];
END;
```

Taxon Namespaces

- *Taxon namespaces* are used to relate OTU identities across different data sources and objects.
- Taxon namespaces are modeled by an object of the `TaxonNamespace` class.
- Every phylogenetic data object (`Tree`, `TreeList`, etc.) has a `taxon_namespace` property that has a `TaxonNamespace` object as its value.
- Two phylogenetic data objects created in different taxon namespaces will NOT share any OTU's *even if the taxon labels are the same*!

Different Taxon Namespaces = Different OTU's

Two phylogenetic data objects created in different taxon namespaces will NOT share any OTU's *even if the taxon labels are the same* ...

```
import dendropy
tree_str = "[&R](A,(B,(C,D)));"

tns1 = dendropy.TaxonNamespace()
tree1 = dendropy.Tree.get(
    data=tree_str,
    schema="newick",
    taxon_namespace=tns1)

tns2 = dendropy.TaxonNamespace()
tree2 = dendropy.Tree.get(
    data=tree_str,
    schema="newick",
    taxon_namespace=tns2)

for leaf1, leaf2 in zip(
    tree1.leaf_node_iter(),
    tree2.leaf_node_iter()):
  print("{} (1) is {} (2): {}".format(
    leaf1.taxon,
    leaf2.taxon,
    leaf1.taxon is leaf2.taxon))
```

Different Taxon Namespaces = Different OTU's

```
import dendropy
tree_str = "[&R](A,(B,(C,D)));"

tns1 = dendropy.TaxonNamespace()
tree1 = dendropy.Tree.get(
    data=tree_str,
    schema="newick",
    taxon_namespace=tns1)

tns2 = dendropy.TaxonNamespace()
tree2 = dendropy.Tree.get(
    data=tree_str,
    schema="newick",
    taxon_namespace=tns2)

for leaf1, leaf2 in zip(
    tree1.leaf_node_iter(),
    tree2.leaf_node_iter()):
  print("{} (1) is {} (2): {}".format(
    leaf1.taxon,
    leaf2.taxon,
    leaf1.taxon is leaf2.taxon))
```

Output

```
'A' (1) is 'A' (2): False
'B' (1) is 'B' (2): False
'C' (1) is 'C' (2): False
'D' (1) is 'D' (2): False
```

Ensuring Common Taxon Namespaces

- Almost every method or function that generates a new phylogenetic data object, whether through parsing an external data source or *de novo* creation (e.g., simulation), takes a `taxon_namespace` argument.
- In DendroPy 4, it is the client code responsibility to ensure that objects that belong in the same "universe" share the same taxon namespace by passing in a common `TaxonNamespace` object via the "`taxon_namespace`" argument.
- If this argument is not a given or is "`None`", a *new* taxon namespace will be created and assigned to the new object.

Ensuring Common Taxon Namespaces

Establish a common namespace by passing in a common `TaxonNamespace` object via the "`taxon_namespace`" argument.

```
import dendropy
tree_str = "[&R](A,(B,(C,D)));"
tns = dendropy.TaxonNamespace()
tree1 = dendropy.Tree.get(
    data=tree_str,
    schema="newick",
    taxon_namespace=tns)
tree2 = dendropy.Tree.get(
    data=tree_str,
    schema="newick",
    taxon_namespace=tns)
for leaf1, leaf2 in zip(
    tree1.leaf_node_iter(),
    tree2.leaf_node_iter()):
  print("{} (1) is {} (2): {}".format(
    leaf1.taxon,
    leaf2.taxon,
    leaf1.taxon is leaf2.taxon))
```

Output

```
'A' (1) is 'A' (2): True
'B' (1) is 'B' (2): True
'C' (1) is 'C' (2): True
'D' (1) is 'D' (2): True
```

Ensuring Common Taxon Namespaces

If a `TaxonNamespace` is not passed to a factory method via the "`taxon_namespace`" argument, or the value passed to this argument is "`None`", a *new* taxon namespace will be created and assigned to the new object.

```
import dendropy
tree_str = "[&R](A,(B,(C,D)));"
tree1 = dendropy.Tree.get(
    data=tree_str,
    schema="newick",)
tree2 = dendropy.Tree.get(
    data=tree_str,
    schema="newick",)
print(tree1.taxon_namespace is tree2.
    taxon_namespace)
for leaf1, leaf2 in zip(
    tree1.leaf_node_iter(),
    tree2.leaf_node_iter()):
  print("{} (1) is {} (2): {}".format(
    leaf1.taxon,
    leaf2.taxon,
    leaf1.taxon is leaf2.taxon))
```

Output

```
False
'A' (1) is 'A' (2): False
'B' (1) is 'B' (2): False
'C' (1) is 'C' (2): False
'D' (1) is 'D' (2): False
```

Ensuring vs. Failing to Ensure Common Taxon Namespaces

```
import dendropy
tree_str = "[&R](A,(B,(C,D)));"
tns = dendropy.TaxonNamespace()
tree1 = dendropy.Tree.get(
    data=tree_str,
    schema="newick",
    taxon_namespace=tns)
tree2 = dendropy.Tree.get(
    data=tree_str,
    schema="newick",
    taxon_namespace=tns)
for leaf1, leaf2 in zip(
    tree1.leaf_node_iter(),
    tree2.leaf_node_iter()):
  print("{} (1) is {} (2): {}".format(
    leaf1.taxon,
    leaf2.taxon,
    leaf1.taxon is leaf2.taxon))
```

Output

```
'A' (1) is 'A' (2): True
'B' (1) is 'B' (2): True
'C' (1) is 'C' (2): True
'D' (1) is 'D' (2): True
```

Ensuring vs. Failing to Ensure Common Taxon Namespaces

```
import dendropy
tree_str = "[&R](A,(B,(C,D)));"
#tns = dendropy.TaxonNamespace()
tree1 = dendropy.Tree.get(
    data=tree_str,
    schema="newick",)
    # taxon_namespace=tns)
tree2 = dendropy.Tree.get(
    data=tree_str,
    schema="newick",)
    # taxon_namespace=tns)
for leaf1, leaf2 in zip(
    tree1.leaf_node_iter(),
    tree2.leaf_node_iter()):
  print("{} (1) is {} (2): {}".format(
    leaf1.taxon,
    leaf2.taxon,
    leaf1.taxon is leaf2.taxon))
```

Output

```
'A' (1) is 'A' (2): False
'B' (1) is 'B' (2): False
'C' (1) is 'C' (2): False
'D' (1) is 'D' (2): False
```

Tree Collections Manage Their Own Taxon Namespace

```
import dendropy
tree_str = "[&R](A,(B,(C,D)));"
trees = dendropy.TreeList()
trees.read(data=tree_str,
    schema="newick",)
trees.read(data=tree_str,
    schema="newick",)
trees.read(data=tree_str,
    schema="newick",)
print(trees[0].taxon_namespace is
        trees.taxon_namespace)
print(trees[1].taxon_namespace is
        trees[0].taxon_namespace)
for leaf1, leaf2 in zip(
    trees[0].leaf_node_iter(),
    trees[1].leaf_node_iter()):
  print("{} (1) is {} (2): {}".format(
    leaf1.taxon,
    leaf2.taxon,
    leaf1.taxon is leaf2.taxon))
```

Output

```
True
True
'A' (1) is 'A' (2): True
'B' (1) is 'B' (2): True
'C' (1) is 'C' (2): True
'D' (1) is 'D' (2): True
```

Multiple Tree Collections: Client Code Must Ensure Same Taxon Namespace Across Collections

```
import dendropy
tree_str = "[&R](A,(B,(C,D)));"
tns = dendropy.TaxonNamespace()
trees1 = dendropy.TreeList(
        taxon_namespace=tns)
trees1.read(data=tree_str,
    schema="newick",)
trees2 = dendropy.TreeList(
        taxon_namespace=tns)
trees2.read(data=tree_str,
    schema="newick",)
print(trees1.taxon_namespace is
        trees2.taxon_namespace)
for leaf1, leaf2 in zip(
    trees1[0].leaf_node_iter(),
    trees2[0].leaf_node_iter()):
  print("{} (1) is {} (2): {}".format(
    leaf1.taxon,
    leaf2.taxon,
    leaf1.taxon is leaf2.taxon))
```

Output

```
True
'A' (1) is 'A' (2): True
'B' (1) is 'B' (2): True
'C' (1) is 'C' (2): True
'D' (1) is 'D' (2): True
```

Multiple Tree Collections: Client Code Must Ensure Same Taxon Namespace Across Collections

```
import dendropy
tree_str = "[&R](A,(B,(C,D)));"
# tns = dendropy.TaxonNamespace()
trees1 = dendropy.TreeList()
        # taxon_namespace=tns)
trees1.read(data=tree_str,
    schema="newick",)
trees2 = dendropy.TreeList()
        # taxon_namespace=tns)
trees2.read(data=tree_str,
    schema="newick",)
print(trees1.taxon_namespace is
        trees2.taxon_namespace)
for leaf1, leaf2 in zip(
    trees1[0].leaf_node_iter(),
    trees2[0].leaf_node_iter()):
  print("{} (1) is {} (2): {}".format(
    leaf1.taxon,
    leaf2.taxon,
    leaf1.taxon is leaf2.taxon))
```

Output

```
False
'A' (1) is 'A' (2): False
'B' (1) is 'B' (2): False
'C' (1) is 'C' (2): False
'D' (1) is 'D' (2): False
```

DataSets: It's Complicated ...

Due to legacy reasons (*Mesquite* and *NeXML* support for multiple independent `TAXA` blocks in a single dataset), by default every independent `DataSet read()` takes place within a new taxon namespace.

```
import dendropy

tree_str1 = "((A,B),C);"

ds = dendropy.DataSet()
ds.read(data=tree_str1, schema="newick")
ds.read(data=tree_str1, schema="newick")

print(len(ds.taxon_namespaces))
print(ds.tree_lists[0].taxon_namespace is ds.tree_lists[1].
    taxon_namespace)
print(ds.tree_lists[0].taxon_namespace[0] is ds.tree_lists[1].
    taxon_namespace[0])

# Results in:
# 2
# False
# False
```

DataSets: It's Complicated ...

So need to manage taxon namespace of reads explicitly ...

```
import dendropy

ds2 = dendropy.DataSet()

tree_str1 = "((A,B),C);"
ds2.read(data=tree_str1, schema="newick")
ds2.read(data=tree_str1, schema="newick",
        taxon_namespace=ds2.tree_lists[0].taxon_namespace)

print(len(ds2.taxon_namespaces))
print(ds2.tree_lists[0].taxon_namespace is ds2.tree_lists[1].
    taxon_namespace)
print(ds2.tree_lists[0].taxon_namespace[0] is ds2.tree_lists[1].
    taxon_namespace[0])

# Results in:
# 1
# True
# True
```

DataSets: It's Complicated ...

So need to manage taxon namespace of reads explicitly ...

```
import dendropy

ds2 = dendropy.DataSet()
tns = ds2.new_taxon_namespace()

tree_str1 = "((A,B),C);"
ds2.read(data=tree_str1, schema="newick", taxon_namespace=tns)
ds2.read(data=tree_str1, schema="newick", taxon_namespace=tns)

print(len(ds2.taxon_namespaces))
print(ds2.tree_lists[0].taxon_namespace is ds2.tree_lists[1].
    taxon_namespace)
print(ds2.tree_lists[0].taxon_namespace[0] is ds2.tree_lists[1].
    taxon_namespace[0])

# Results in:
# 1
# True
# True
```

DataSets: It's Complicated ...

Advanced option: use the "attached taxon namespace" mode.

```
import dendropy

ds = dendropy.DataSet()
taxa = dendropy.TaxonNamespace(label="global")
# All future reads will use the specified
# taxon namespace:
ds.attach_taxon_namespace(taxa)
tree_str1 = "((A,B),C);"
ds.read(data=tree_str1, schema="newick")
ds.read(data=tree_str1, schema="newick")

print(len(ds.taxon_namespaces))
print(ds.tree_lists[0].taxon_namespace is ds.tree_lists[1].
    taxon_namespace)
print(ds.tree_lists[0].taxon_namespace[0] is ds.tree_lists[1].
    taxon_namespace[0])

# Results in:
# 1
# True
# True
```

Taxon Namespaces in DendroPy

- Can be annoying!
- Apologies!
- Future plans: DendroPy will use a single global taxon namespace by default if not instructed otherwise by user.
- In the mean time, creating a single `TaxonNamespace` and explicitly passing it to various factory/parser methods via the "`taxon_namespace`" argument is the way to go.

```
tns = dendropy.TaxonNamepace()
trees1 = dendropy.TreeList.get(path="constr.nex", schema="nexus",
  taxon_namespace=tns)
trees2 = dendropy.TreeList.get(path="unconstr.nex", schema="nexus",
  taxon_namespace=tns)
d1 = dendropy.DnaCharacterMatrix.get(path="gn1.fas", schema="fasta",
  taxon_namespace=tns)
d2 = dendropy.DnaCharacterMatrix.get(path="gn2.fas", schema="fasta",
  taxon_namespace=tns)
```

Rooting State of a Tree

- A `Tree` object has a "`is_rooted`" attribute that determines its rooting state: `True`, `False`, or `None`.
- A rooted tree has this attribute set to `True`, while an unrooted tree has this attribute set to `False`.
- If this attribute's value is `None`, then the tree *is assumed to be* **unrooted**.
- The rooting state is very important when calculating the splits or bipartition hashes (more on this later).

```
tree.is_rooted = None  # defaults to unrooted
tree.is_rooted = False # unrooted
tree.is_rooted = True  # rooted
```

Indicating the Rooting State of Trees in a Data Source: the "[&U]" and "[&R]" Tokens

- Newick and NEXUS formats have a convention where the rooting of the tree is specified by a special comment token preceding the tree statement:
  - "`[&U]`" to indicate an unrooted tree, e.g.: `[&U] ((A,B),(C,D));`
  - "`[&R]`" to indicate a rooted tree, e.g.: `[&R] ((A,B),(C,D));`
- These rooting comment tokens are respected when tree data is read. If no such comment token is given, then the tree is assumed to be *unrooted by default*.

Controlling the Rooting State of Trees in a Data Source: the "rooting" Keyword Argument

You can control the behavior of trees read by using the "rooting" keyword argument:

```
import dendropy

# force tree to be read as rooted
tree = dendropy.Tree.get(path='pythonidae.mle.nex', schema='nexus',
        rooting='force-rooted')

# force tree to be read as unrooted
tree = dendropy.Tree.get(path='pythonidae.mle.nex', schema='nexus',
        rooting='force-unrooted')

# tree defaults to rooted unless ``[&U]'' explicitly specified
tree = dendropy.Tree.get(path='pythonidae.mle.nex', schema='nexus',
        rooting='default-rooted')

# [DEFAULT behavior]
# tree defaults to unrooted unless ``[&R]'' explicitly specified
tree = dendropy.Tree.get(path='pythonidae.mle.nex', schema='nexus',
        rooting='default-unrooted')
```

Rooted vs. Unrooted Trees

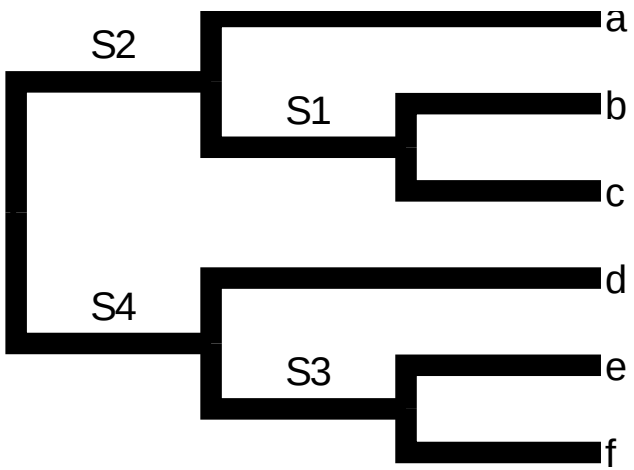
- Rooting/unrooting a tree is not just an cosmetic operation.
- It is a fundamentally structural one that changes the model representing the relationships of taxa.
- In many applications, there might be no practical difference, and thus the operation is not distinguishable from a purely cosmetic procedure.
- In applications in which split/bipartition identity is important, the failure to recognize that the rootedness of a tree is a fundamental aspect of the tree model, and changing it has consequences for the identity of the splits on tree, leads to frustration.
- A frustration which leads to anger.
- An anger which leads to me.

What is a Bipartition?

- A bipartition is a division or sorting of the leaves/tips of a tree into two mutually-exclusive and collectively-exhaustive subsets.
- Every edge on a tree corresponds to a bipartition in the sense that if were were to split or bisect a tree at a particular edge, the leaf sets of each of the two new trees constitute the a bipartition of the leaf set of the original tree.
- In the context of evolutionary trees like a phylogeny, the leaves typically are associated with taxa (OTU's).
- So, just as we view a tree as a schematic representation of the relationships of taxa, we can see
  - Bipartitions as a representation of clusterings of taxa/OTU's.
  - A tree as a collection of bipartitions.

What is a Bipartition Encoding?

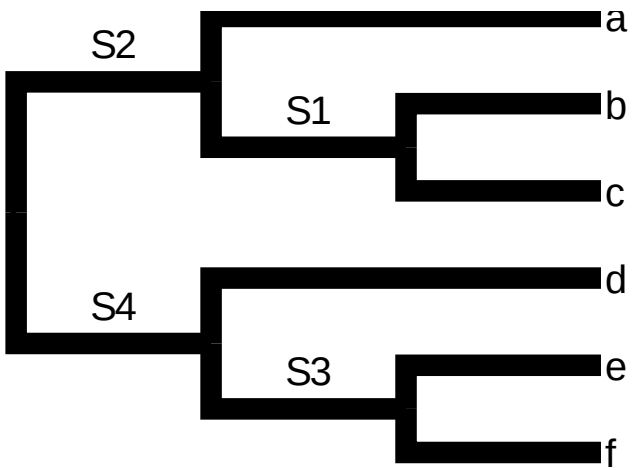
Consider the tree: ((a,(b,c)),(d,(e,f))); this tree not only consists of the following bipartitions, but is also (given the taxa) *exactly* defined by them:



1. $b, c|a, d, e, f$
2. $a, b, c|d, e, f$
3. $e, f|a, b, c, d$
4. $d, e, f|a, b, c$

What is a Bipartition Encoding?

Consider the tree: ((a,(b,c)),(d,(e,f))); this tree not only consists of the following bipartitions, but is also (given the taxa) *exactly* defined by them:



1. $b, c|a, d, e, f$
2. $a, b, c|d, e, f$
3. $e, f|a, b, c, d$
4. $d, e, f|a, b, c$

The collection of bipartitions that exactly define a tree is called the *bipartition encoding* of a tree.

What is a Bipartition Hash?

- A bipartition hash is an internal calculation that allows us to summarize a bipartition identity with a single number.
- A special form of a bipartition hash, in DendroPy called a "*split bitmask*", is used to **establish the identity of bipartitions across multiple different trees** (from potentially different data sources).
- The establishment of this identity is crucial to the functioning of many of DendroPy's advance tree statistics and calculation.
- You too will find this identity useful if you want to compare almost anything across different trees (e.g., similarities/differences in branch lengths, gene conflict).

Calculating the Hash: From a Bipartition to a Bitmask

- Consider a collection of 5 taxa: a, b, c, d, e .
- The split

$$b, c | a, d, e, f$$

divides the taxa into two groups, which we can assign the labels "0" and "1":

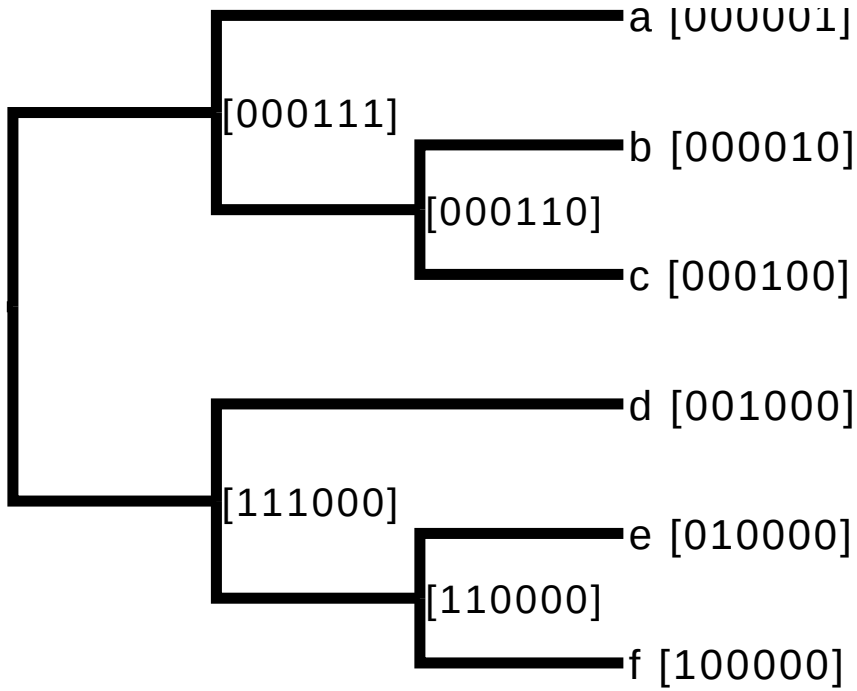
Taxon	f	e	d	c	b	a
Group	0	0	0	1	1	0

- The resulting pattern, 000110, can be interpreted as a binary representation of 6.
- If we were to be consistent about the binary digit "position" that each taxon mapped to across trees, as well as have rules to resolve which group gets the "0" and which the "1", then the integer value 6 would be a robust, reliable, repeatable, and efficient representation of the split.

Calculating the Bipartition Hashes on a Tree

((a,(b,c)),(d,(e,f)));

Split	fedcba	Hash
a|bcdef	000001	1
b|acdef	000010	2
c|abdef	000100	4
d|abcef	001000	8
e|abcdf	010000	16
f|abcdf	100000	32
bc|adef	000110	6
abc|def	000111	7
ef|abcd	110000	48
def|abc	111000	56



Letting DendroPy Do the Calculations For You

- The "`encode_bipartitions`" method of the `Tree` class will automatically calculate all the bipartitions hashes for you, representing each one as a `Bipartition` object.
- An attribute, "`bipartition_encoding`" will be added to the `Tree` instance, containing a (list) collection of `Bipartition` objects corresponding to all bipartitions on the tree.
- Another attribute, "`bipartition_edge_map`" will also be added to the `Tree` instance, containing a dictionary with the `Bipartition` objects as keys and the `Edge` instance on the `Tree` instance to which the `Bipartition` object corresponds as values.
- These `Bipartition` objects can be compared across `Tree` instances: even if generated on different trees, if they correspond to the same split, they will evaluate to being equal.
- Thus you can use the same `Bipartition` object as a key to dereference corresponding edges (and thus clades, cliques, or splits) on different trees.

Letting DendroPy Do the Calculations For You

```
import dendropy
tns = dendropy.TaxonNamespace()
t1 = dendropy.Tree.get(
  data="[&R] (a,(b,(c,(d,(e,f)))));",
  schema="newick",
  taxon_namespace=tns)
t2 = dendropy.Tree.get(
  data="[&R] (((a,(b,c)),((d,e),f)));",
  schema="newick",
  taxon_namespace=tns)
t1.encode_bipartitions()
t2.encode_bipartitions()
# iterate over bipartitions in tree1
for bp1 in t1.bipartition_edge_map:
    # check if it exists in tree2
    if bp1 in t2.bipartition_edge_map:
        print("In common: {}".format(bp1))
    else:
        print("In t1: {}".format(bp1))
for bp2 in t2.bipartition_edge_map:
    if bp2 not in t1.bipartition_edge_map:
        print("In t2: {}".format(bp2))
```

```
In common: 100000
In common: 000001
In common: 000010
In common: 000100
In t1: 110000
In common: 001000
In common: 010000
In common: 111000
In t1: 111100
In t1: 111110
In common: 111111
In t2: 000110
In t2: 000111
In t2: 011000
```

Letting DendroPy Do the Calculations For You

```
import dendropy
data = """\
[&R] (((a[&color=red],(b[&color=blue],c[&color=green])[&color=red])[&
    color=blue],
      ((d[&color=green],e[&color=red]),f[&color=blue]))[&color=green
    ])[&color=red];
[&R] (((a[&color=blue],(b[&color=blue],c[&color=red])[&color=green])
    [&color=red],
      ((d[&color=blue],e[&color=red]),f[&color=red]))[&color=red])[&
    color=blue];
"""
trees = dendropy.TreeList.get(
        data=data,
        schema="newick")
t1 = trees[0]
t2 = trees[1]
t1.encode_bipartitions()
t2.encode_bipartitions()
```

Letting DendroPy Do the Calculations For You

```
t1.encode_bipartitions()
t2.encode_bipartitions()
for bp in t1.bipartition_encoding:
    assert bp in t2.bipartition_edge_map
    t1_edge = t1.bipartition_edge_map[bp]
    t2_edge = t2.bipartition_edge_map[bp]
    color1 = t1_edge.head_node.annotations["color"].value
    color2 = t2_edge.head_node.annotations["color"].value
    if color1 != color2:
        print("Split {}: differ in color: '{}' vs '{}'".format(
            bp, color1, color2))
```

Letting DendroPy Do the Calculations For You

```
t1.encode_bipartitions()
t2.encode_bipartitions()
for bp in t1.bipartition_encoding:
    assert bp in t2.bipartition_edge_map
    t1_edge = t1.bipartition_edge_map[bp]
    t2_edge = t2.bipartition_edge_map[bp]
    color1 = t1_edge.head_node.annotations["color"].value
    color2 = t2_edge.head_node.annotations["color"].value
    if color1 != color2:
        print("Split {}: differ in color: '{}' vs '{}'".format(
            bp, color1, color2))
```

```
Split 000001: differ in color: 'red' vs 'blue'
Split 000100: differ in color: 'green' vs 'red'
Split 000110: differ in color: 'red' vs 'green'
Split 000111: differ in color: 'blue' vs 'red'
Split 001000: differ in color: 'green' vs 'blue'
Split 100000: differ in color: 'blue' vs 'red'
Split 111111: differ in color: 'green' vs 'red'
```

Letting DendroPy Do the Calculations For You

```
import collections
import dendropy
data = """\
[&R] (a:0.6,((b:0.1,c:0.8):0.9,(d:0.1,e:0.4):0.1):0.3):0.5;
[&R] (a:0.4,((c:0.5,b:0.6):0.8,(d:0.5,e:0.6):0.2):0.6):0.4;
[&R] (a:0.4,((b:0.2,c:0.2):0.3,(e:0.4,d:0.4):0.7):0.6):0.4;
[&R] (a:0.3,((b:0.4,c:0.1):0.4,(d:0.4,e:0.2):0.2):0.6):0.1;
[&R] (a:0.3,((b:0.4,c:0.9):0.2,(d:0.6,e:0.8):0.4):0.9):0.4;
[&R] (a:0.2,((b:0.5,c:0.4):0.9,(d:0.1,e:0.3):0.4):0.4):0.8;
[&R] (a:0.1,((b:0.2,c:0.5):0.2,(d:0.3,e:0.2):0.4):0.9):0.6;
[&R] (a:0.4,((b:0.1,c:0.8):0.8,(d:0.7,e:0.2):0.2):0.4):0.1;
[&R] (a:0.7,((b:0.9,c:0.3):0.2,(d:0.2,e:0.4):0.5):0.2):0.5;
[&R] (a:0.8,((b:0.3,c:0.7):0.2,(d:0.5,e:0.1):0.9):0.3):0.1;
"""
```

Letting DendroPy Do the Calculations For You

```
trees = dendropy.TreeList.get(
        data=data,
        schema="newick")
bp_lens = collections.defaultdict(list)
for tree in trees:
    tree.encode_bipartitions()
    for bp in tree.bipartition_edge_map:
        edge = tree.bipartition_edge_map[bp]
        bp_lens[bp].append(edge.length)
target_tree = dendropy.Tree.from_bipartition_encoding(
        bipartition_encoding=list(bp_lens),
        taxon_namespace=trees.taxon_namespace,
        is_rooted=True,
        )
target_tree.encode_bipartitions()
for bp in target_tree.bipartition_encoding:
    ttlen = sum(bp_lens[bp]) / len(bp_lens[bp])
    target_tree.bipartition_edge_map[bp].length = ttlen
print(target_tree.as_string("newick"))
```

Letting DendroPy Do the Calculations For You

```
trees = dendropy.TreeList.get(
        data=data,
        schema="newick")
bp_lens = collections.defaultdict(list)
for tree in trees:
    tree.encode_bipartitions()
    for bp in tree.bipartition_edge_map:
        edge = tree.bipartition_edge_map[bp]
        bp_lens[bp].append(edge.length)
target_tree = dendropy.Tree.from_bipartition_encoding(
        bipartition_encoding=list(bp_lens),
        taxon_namespace=trees.taxon_namespace,
        is_rooted=True,
        )
target_tree.encode_bipartitions()
for bp in target_tree.bipartition_encoding:
    ttlen = sum(bp_lens[bp]) / len(bp_lens[bp])
    target_tree.bipartition_edge_map[bp].length = ttlen
print(target_tree.as_string("newick"))
```

```
[&R] (a:0.42,((b:0.37,c:0.52):0.49,(d:0.38,e:0.36):0.4):0.52):0.39;
```

Calculating Bipartitions on Unrooted Trees

- When calculating bipartition hashes on unrooted trees, the basal split gets collapsed.
- This is necessary for correct behavior, to avoid a redundant bipartition hash.
- This also is not changing the "real" structure of a tree: on an unrooted tree, the basal bifurcation is an **algorithmic artifact**: *it does not exist*.
- This upsets many people.
- This causes a lot of anger.
- This anger is usually directed at me, telling me DendroPy is broken.
- It is not.
- If you want to preserve the basal bifurcation, you want rooted trees: say "`rooting='force-rooted'`" to get correct behavior, and be done with it.

DendroPy Calculates Bipartitions in the Background

- Many operations, calculations, etc. in DendroPy make use of bipartition hashes in the background.
- So you may be using bipartitions without realizing it.
- This means you should make it a point to explicitly control the rooting state of your trees even if you are not using bipartitions hash directly yourself.

More Information Here

https://pythonhosted.org/DendroPy/primer/bipartitions.html

DendroPy ...

- Efficient single-tree-at-a-time operations: `Tree.yield_from_files`.
- Efficient handling of very large collections of very large trees: `TreeArray`.
- Simulation of trees under many models: birth-death; coalescent; multispecies coalescent ("gene trees in species trees" or the "censored/truncated coalescent"); etc.
- Simulation of characters on trees under various discrete and continuous character models.
- Calculation of: many tree and tree shape statistics; likelihood under the coalescent and multispecies coalescent; parsimony score; gene/species tree fitting/reconcilation; patristic distances; community ecology distance statistics; etc.
- Interoperability with: Genbank, GBIF, PAUP, MUSCLE, SeqGen, RaXML.