Tutorial: Environment for Tree Exploration

Release 2.1

Jaime Huerta-Cepas

June 29, 2012

Contents

[Download PDF documentation]ǁ

CHAPTER 1

Download and Install

Contents

1.1 GNU/Linux

ETE requires python>=2.5 (python 3 is not supported) as well as several dependencies (required only to enable particular functions, but highly recommended):

- python-qt4 (>=4.5) [Enables tree visualization and image rendering]
- python-mysqldb (>=1.2) [Enables programmatic access to PhylomeDB]
- python-numpy (Required to work with clustering trees)
- python-lxml [Required to work whit NexML and PhyloXML phylogenetic formats]
- python-setuptools [Optional. Allows to install and upgrade ETE]

1.1.1 Meeting dependencies (Debian based distributions)

ETE is developed and tested under Debian based distributions (i.e. Ubuntu). All the above mentioned dependencies are in the official repositories of Debian and can be installed using a package manager such as APT, Synaptics or Adept. For instance, in Ubuntu you can use the following shell command to install all dependencies:

```
$ apt-get install python-setuptools python-numpy python-qt4 python-scipy python-mysqldb
```

1.1.2 Meeting dependencies (other GNU/Linux distributions)

In Non Debian based distributions, dependencies may not be necessarily found in their official repositories. If this occurs, libraries should be downloaded separately or installed from third part repositories. In general, this process should not entail important difficulties, except for PyQt4, which is a python binding to the new Qt4 libraries. Some distributions (i.e. CentOS, Fefora) do not include recent packages and cross-dependencies for such libraries yet. In such cases, manual compilation of libraries could be required.

1.1.3 Installing or Upgrading ETE

Easy Install is the best way to install ETE and keep it up to date. EasyInstall is a python module bundled with setuptools that lets you automatically download, build, install, and manage Python packages. If the easy_install command is available in your system, you can execute this shell command to install/update ETE.

```
$ easy_install -U ete2
```

Alternatively, you can download the last version of ETE from http://pypi.python.org/pypi/ete2/, decompress the file and install the package by executing the setup installer:

```
$ python setup.py install
```

1.2 MacOS

ETE and all its dependencies are supported by MacOS Intel environments, however you will need to install some libraries from the external GNU/open-source repositories. This can be done easily by using MacPorts.

The following recipe has been reported to work in MacOS 10.5.8 (thanks to Marco Mariotti and Alexis Grimaldi at the CRG):

1. Install Mac Developer tools and X11 (required by Macports)
2. Install Macports in your system: http://www.macports.org/install.php
3. Install the following packages from the macports repository by using the “sudo port install [package_name]” syntax (note that some packages may take a long time to be built and that you will need to have an active internet connection during the installation process): * python26 * py26-numpy * py26-scipy * py26-pyqt4 * py26-mysql * py26-lxml
4. Download the setup installer of the last ETE version (http://ete.cgenomics.org/releases/ete2), uncompress it, enter its folder and run: “sudo python setup.py install” Once the installation has finished, you will be able to load ETE (import ete2) when running the “right” python binary.

Note: If step 4 doesn’t work, make sure that the python version your are using to install ETE is the one installed by MacPorts. This is usually located in `/opt/local/Library/Frameworks/Python.framework/Versions/2.6/bin/python2.6`. By contrast, non-Macport python version is the one located in `/Library/Frameworks/Python.framework/Versions/2.6/bin/python2.6`, so check that you are using the correct python executable.

1.3 Older Versions

Older ETE versions can be found at http://ete.cgenomics.org/releases/ete2/

CHAPTER 2

Changelog history

2.1 What's new in ETE 2.1

- A basic standalone tree visualization program called "ete2" is now installed along with the package.
- The drawing engine has been completely rewritten to provide the following new features:
  - Added `TreeStyle` class allowing to set the following
    - Added **circular tree drawing** mode
    - Added tree *title face block* (Text or images that rendered on top of the tree)
    - Added tree *legend face block* (Text or images that rendered as image legend)
    - Added support for *tree rotation and orientation*
    - Possibility of drawing *aligned faces as a table*
    - Added header and footer regions for aligned faces.
    - And more! Check `TreeStyle` documentation
  - Added new face positions **float**, **branch-top** and **branch-bottom**. See tutorial (*Node faces*) for more details.
  - Added several `Face` attributes:
    - face border
    - face background color
    - left, right, top and bottom margins
    - face opacity
    - horizontal and vertical alignment (useful when faces are rendered as table)
  - Added support for predefined `NodeStyle`, which can be set outside the layout function (allows to save and export image rendering info)
  - **Added new face types:**
      - `CircleFace` (basic circle/sphere forms)

    - `TreeFace` (trees within trees)
    - `StaticItemFace` and `DynamicItemFace` (create custom and interactive QtGraphicsItems)
  - **Improved faces:**
    - `AttrFace` accepts prefix and suffix text, as well as a text formatter function. `fstyle` argument can be set to `italic`
    - `TextFace`: `fstyle` argument can be set to `italic`
  - **Save and export images**
    - Added full support for SVG image rendering
    - Added more options to the `TreeNode.render()` function to control image size and resolution
  - Added support for `SVG_COLORS` names in faces and node styles
- **Core methods:**
  - Added `TreeNode.copy()`: returns an exact and independent copy of node and all its attributes
  - Added `TreeNode.convert_to_ultrametric()`: converts all branch lengths to allow leaves to be equidistant to root
  - Added `TreeNode.sort_descendants()`: sort tree branches according to node names.
  - Added `TreeNode.ladderize()`: sort tree branches according to partition size
  - Added `TreeNode.get_partitions()`: return the set of all possible partitions grouping leaf nodes
  - Tree nodes can now be fully exported using cPickle
  - Newick parser can read and export branch distances and support values using scientific notation
  - `TreeNode.swap_childs()` method has changed to `TreeNode.swap_children()`
- Added `ete2.nexml` module (read and write nexml format)
- Added `ete2.phyloxml` module (read and write phyloxml format)
- Added `ete2.webplugin` module: Allows to create interactive web tree applications
- Tree visualization GUI checks now for newer version of the ETE package.
- Added `PhylomeDB3Connector`
- Added `PhyloNode.get_farthest_oldest_node()` function, which allows to find the best outgroup node in a tree, even if it is an internal node.
- **Bug Fixes and improvements:**
  - Fix: `TreeNode.get_common_ancestor()` accepts a single argument (node or list of nodes) instead of a succession or nodes. It can also return the path of each node to the parent.

  - Fix: Fast scroll based zoom-in was producing tree image inversions
  - Fix: Phylip parser does not truncate long names by default
  - Fix: "if not node" syntax was using a len(node) test, which made it totally inefficient. Now, the same expression returns always *True*
  - Improvement: Traversing methods are now much faster (specially preorder and level-order)
  - Improvement: Faster populate function (added possibility of random and non-random branch lengths)
  - Improvement: Faster prune function
  - Improvement: unicode support for tree files
  - Improvement: Added newick support for scientific notation in branch lengths

- **Improved documentation and examples:**
  - Online and PDF tutorial
  - Better library reference
  - A set of examples is now provided with the installation package and here

CHAPTER 3

The ETE tutorial

Contents:

3.1 Working With Tree Data Structures

Contents

3.1.1 Trees

Trees are a widely-used type of data structure that emulates a tree design with a set of linked nodes. Formally, a tree is considered an acyclic and connected graph. Each node in a tree has zero or more child nodes, which are below it in the tree (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent.

The height of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The depth of a node is the length of the path to its root (i.e., its root path).

- The topmost node in a tree is called the root node. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin (although some algorithms begin with the leaf nodes and work up ending at the root). All other nodes can be reached from it by following edges or links. Every node in a tree can be seen as the root node of the subtree rooted at that node.
- Nodes at the bottommost level of the tree are called leaf nodes. Since they are at the bottommost level, they do not have any children.

- An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node.
- A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T, together with all the nodes below it, comprise a subtree of T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset).

In bioinformatics, trees are the result of many analyses, such as phylogenetics or clustering. Although each case entails specific considerations, many properties remains constant among them. In this respect, ETE is a python toolkit that assists in the automated manipulation, analysis and visualization of any type of hierarchical trees. It provides general methods to handle and visualize tree topologies, as well as specific modules to deal with phylogenetic and clustering trees.

3.1.2 Reading and Writing Newick Trees

The Newick format is one of the most widely used standard representation of trees in bioinformatics. It uses nested parentheses to represent hierarchical data structures as text strings. The original newick standard is able to encode information about the tree topology, branch distances and node names. Nevertheless, it is not uncommon to find slightly different formats using the newick standard.

ETE can read and write many of them:

FORMAT	DESCRIPTION	SAMPLE
0	flexible with support values	((D:0.723274,F:0.567784)1.000000:0.067192,(B:0.279326,H:0.756049)1.00
1	flexible with internal node names	((D:0.723274,F:0.567784)E:0.067192,(B:0.279326,H:0.756049)B:0.807788)
2	all branches + leaf names + internal supports	((D:0.723274,F:0.567784)1.000000:0.067192,(B:0.279326,H:0.756049)1.00
3	all branches + all names	((D:0.723274,F:0.567784)E:0.067192,(B:0.279326,H:0.756049)B:0.807788)
4	leaf branches + leaf names	((D:0.723274,F:0.567784),(B:0.279326,H:0.756049));
5	internal and leaf branches + leaf names	((D:0.723274,F:0.567784):0.067192,(B:0.279326,H:0.756049):0.807788);
6	internal branches + leaf names	((D,F):0.067192,(B,H):0.807788);
7	leaf branches + all names	((D:0.723274,F:0.567784)E,(B:0.279326,H:0.756049)B);
8	all names	((D,F)E,(B,H)B);
9	leaf names	((D,F),(B,H));
100	topology only	((,),(,));

Formats labeled as *flexible* allow for missing information. For instance, format 0 will be able to load a newick tree even if it does not contain branch support information (it will be initialized with the default value). However, format 2 would raise an exception. In other words, if you want to control that your newick files strictly follow a given pattern you should use **strict** format definitions.

Reading newick trees

In order to load a tree from a newick text string you can use the constructor `TreeNode` or its `Tree` alias, provided by the main module `ete2`. You will only need to pass a text string containing the newick structure and the format that should be used to parse it (0 by default). Alternatively, you can pass the path to a text file containing the newick string.

```
from ete2 import Tree

# Loads a tree structure from a newick string. The returned variable 't' is the root noc
t = Tree("(A:1,(B:1,(E:1,D:1):0.5):0.5);" )

# Load a tree structure from a newick file.
t = Tree("genes_tree.nh")

# You can also specify the newick format. For instance, for named internal nodes we wil
t = Tree("(A:1,(B:1,(E:1,D:1)Internal_1:0.5)Internal_2:0.5)Root;", format=1)
```

Writing newick trees

Any ETE tree instance can be exported using newick notation using the `Tree.write()` method, which is available in any tree node instance. It also allows for format selection (*Reading and Writing Newick Trees*), so you can use the same function to convert between newick formats.

```
from ete2 import Tree

# Loads a tree with internal node names
t = Tree("(A:1,(B:1,(E:1,D:1)Internal_1:0.5)Internal_2:0.5)Root;", format=1)

# And prints its newick using the default format

print t.write() # (A:1.000000,(B:1.000000,(E:1.000000,D:1.000000)1.000000:0.500000)1.000

# To print the internal node names you need to change the format:

print t.write(format=1) # (A:1.000000,(B:1.000000,(E:1.000000,D:1.000000)Internal_1:0.50

# We can also write into a file
t.write(format=1, outfile="new_tree.nw")
```

3.1.3 Understanding ETE Trees

Any tree topology can be represented as a succession of **nodes** connected in a hierarchical way. Thus, for practical reasons, ETE makes no distinction between tree and node concepts, as any tree can be represented by its root node. This allows to use any internal node within a tree as another sub-tree instance.

Once trees are loaded, they can be manipulated as normal python objects. Given that a tree is actually a collection of nodes connected in a hierarchical way, what you usually see as a tree will be the root node instance from which the tree structure is hanging. However, every node within a ETE's tree structure can be also considered a subtree. This means, for example, that all the operational methods that we will review in the following sections are available at any possible level within a tree. Moreover, this feature will allow you to separate large trees into smaller partitions, or concatenate several trees into a single structure. For this reason, you will find that the `TreeNode` and `Tree` classes are synonymous.

3.1.4 Basic tree attributes

Each tree node has two basic attributes used to establish its position in the tree: `TreeNode.up` and `TreeNode.children`. The first is a pointer to parent's node, while the later is a list of children nodes. Although it is possible to modify the structure of a tree by changing these attributes, it is strongly recommend not to do it. Several methods are provided to manipulate each node's connections in a safe way (see *Modifying Tree Topology*).

In addition, three other basic attributes are always present in any tree node instance:

Method	Description	Default value
`TreeNode.dist`	stores the distance from the node to its parent (branch length). Default value = 1.0	1.0
`TreeNode.support`	informs about the reliability of the partition defined by the node (i.e. bootstrap support)	1.0
`TreeNode.name`	Custom node's name.	NoName

In addition, several methods are provided to perform basic operations on tree node instances:

Method	Description
`TreeNode.is_leaf()`	returns True if *node* has no children
`TreeNode.is_root()`	returns True if *node* has no parent
`TreeNode.get_tree_root()`	returns the top-most node within the same tree structure as *node*
`len(TreeNode)`	returns the number of leaves under *node*
`print node`	prints a text-based representation of the tree topology under *node*
`if node in tree`	returns true if *node* is a leaf under *tree*
`for leaf in node`	iterates over all leaves under *node*
`TreeNode.show()`	Explore node graphically using a GUI.

This is an example on how to access such attributes:

```
from ete2 import Tree
t = Tree()
# We create a random tree topology
t.populate(15)
print t
print t.children
print t.get_children()
print t.up
print t.name
print t.dist
print t.is_leaf()
print t.get_tree_root()
print t.children[0].get_tree_root()
print t.children[0].children[0].get_tree_root()
# You can also iterate over tree leaves using a simple syntax
for leaf in t:
  print leaf.name
```

Root node on unrooted trees?

When a tree is loaded from external sources, a pointer to the top-most node is returned. This is called the tree root, and **it will exist even if the tree is conceptually considered as unrooted**. This is, the root node can be considered as the master node, since it represents the whole tree structure. Unrooted trees can be identified as trees in which master root node has more than two children.

```
from ete2 import Tree
unrooted_tree = Tree( "(A,B,(C,D));" )
```

```
print unrooted_tree
#
#     /-A
#    |
#----|--B
#    |
#    |     /-C
#     \---|
#          \-D

rooted_tree = Tree( "((A,B).(C,D));" )
print rooted_tree
#
#          /-A
#     /---|
#    |     \-B
#----|
#    |     /-C
#     \---|
#          \-D
```

3.1.5 Browsing trees

One of the most basic operations for tree analysis is *tree browsing*. This is, essentially, visiting nodes within a tree. ETE provides a number of methods to search for specific nodes or to navigate over the hierarchical structure of a tree.

Getting Leaves, Descendants and Node's Relatives

TreeNode instances contain several functions to access their descendants. Available methods are self explanatory:

`TreeNode.get_descendants`([strategy, is_leaf_fn])	Returns a list of all (leaves and internal) descendant nodes
`TreeNode.get_leaves`([is_leaf_fn])	Returns the list of terminal nodes (leaves) under this node
`TreeNode.get_leaf_names`([is_leaf_fn])	Returns the list of terminal node names under the current
`TreeNode.get_children`()	Returns an independent list of node's children.
`TreeNode.get_sisters`()	Returns an indepent list of sister nodes.

Traversing (browsing) trees

Often, when processing trees, all nodes need to be visited. This is called tree traversing. There are different ways to traverse a tree structure depending on the order in which children nodes are visited. ETE implements the three most common strategies: **preorder**, **levelorder** and **postorder**. The following scheme shows the differences in the strategy for visiting nodes (note that in both cases the whole tree is browsed):

- preorder: 1)Visit the root, 2) Traverse the left subtree , 3) Traverse the right subtree.
- postorder: 1) Traverse the left subtree , 2) Traverse the right subtree, 3) Visit the root
- levelorder (default): every node on a level before is visited going to a lower level

Note:

- Preorder traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right)
- Inorder traversal sequence: A, B, C, D, E, F, G, H, I (left, root, right); note how this produces a sorted sequence
- Postorder traversal sequence: A, C, E, D, B, H, I, G, F (left, right, root)
- Level-order traversal sequence: F, B, G, A, D, I, C, E, H

Every node in a tree includes a `TreeNode.traverse()` method, which can be used to visit, one by one, every node node under the current partition. In addition, the `TreeNode.iter_descendants()` method can be set to use either a post- or a preorder strategy. The only different between `TreeNode.traverse()` and `TreeNode.iter_descendants()` is that the first will include the root node in the iteration.

`TreeNode.traverse`([strategy, is_leaf_fn])	Returns an iterator to traverse the tree structure under th
`TreeNode.iter_descendants`([strategy, is_leaf_fn])	Returns an iterator over all descendant nodes.
`TreeNode.iter_leaves`([is_leaf_fn])	Returns an iterator over the leaves under this node.

strategy can take one of the following values: `"postorder"`, `"preorder"` or `"levelorder"`

```
# we load a tree
t = Tree('((((H,K)D,(F,I)G)B,E)A,((L,(N,Q)O)J,(P,S)M)C);', format=1)

for node in t.traverse("postorder"):
  # Do some analysis on node
  print node.name

# If we want to iterate over a tree excluding the root node, we can
# use the iter_descendant method
for node in t.iter_descendants("postorder"):
  # Do some analysis on node
  print node.name
```

Additionally, you can implement your own traversing function using the structural attributes of nodes. In the following example, only nodes between a given leaf and the tree root are visited.

```
from ete2 import Tree
tree = Tree( "(A:1,(B:1,(C:1,D:1):0.5):0.5);" )

# Browse the tree from a specific leaf to the root
node = t.search_nodes(name="C")[0]
while node:
   print node
   node = node.up
```

Iterating instead of Getting

As commented previously, methods starting with **get_** are all prepared to return results as a closed list of items. This means, for instance, that if you want to process all tree leaves and you ask for them using the `TreeNode.get_leaves()` method, the whole tree structure will be browsed before returning the final list of terminal nodes. This is not a problem in most of the cases, but in large trees, you can speed up the browsing process by using iterators.

Most **get_** methods have their homologous iterator functions. Thus, `TreeNode.get_leaves()` could be substituted by `TreeNode.iter_leaves()`. The same occurs with `TreeNode.iter_descendants()` and `TreeNode.iter_search_nodes()`.

When iterators are used (note that is only applicable for looping), only one step is processed at a time. For instance, `TreeNode.iter_search_nodes()` will return one match in each iteration. In practice, this makes no differences in the final result, but it may increase the performance of loop functions (i.e. in case of finding a match which interrupts the loop).

Finding nodes by their attributes

Both terminal and internal nodes can be located by searching along the tree structure. Several methods are available:

method	Description
t.search_nodes(attr=value)	Returns a list of nodes in which attr is equal to value, i.e. name=A
t.iter_search_nodes(attr=value)	Iterates over all matching nodes matching attr=value. Faster when you only need to get the first occurrence
t.get_leaves_by_name(name)	Returns a list of leaf nodes matching a given name. Only leaves are browsed.
t.get_common_ancestor([node1, node2, node3])	Return the first internal node grouping node1, node2 and node3
t&”A”	Shortcut for t.search_nodes(name=”A”)[0]

Search_all nodes matching a given criteria

A custom list of nodes matching a given name can be easily obtain through the `TreeNode.search_node()` function.

```
from ete2 import Tree
t = Tree( '((H:1,I:1):0.5, A:1, (B:1,(C:1,D:1):0.5):0.5);' )
print t
#                    /-H
#          /--------|
#         |          \-I
#         |
#---------|--A
#         |
#         |          /-B
#          \--------|
#                   |          /-C
#                    \--------|
#                              \-D

# I get D
D = t.search_nodes(name="D")[0]

# I get all nodes with distance=0.5
nodes = t.search_nodes(dist=0.5)
print len(nodes), "nodes have distance=0.5"

# We can limit the search to leaves and node names (faster method).
D = t.get_leaves_by_name(name="D")
print D
```

Search nodes matching a given criteria (iteration)

A limitation of the `TreeNode.search_nodes()` method is that you cannot use complex conditional statements to find specific nodes. When search criteria is too complex, you may need to create your own search function.

```
from ete2 import Tree

def search_by_size(node, size):
    "Finds nodes with a given number of leaves"
    matches = []
    for n in node.traverse():
       if len(n) == size:
          matches.append(n)
    return matches

t = Tree()
t.populate(40)
# returns nodes containing 6 leaves
search_by_size(t, size=6)
```

Find the first common ancestor

Searching for the first common ancestor of a given set of nodes it is a handy way of finding internal nodes.

```
from ete2 import Tree
t = Tree( "((H:0.3,I:0.1):0.5, A:1, (B:0.4,(C:0.5,(J:1.3, (F:1.2, D:0.1):0.5):0.5):0.5):
print t
ancestor = t.get_common_ancestor("C", "J", "B")
```

Custom searching functions

A limitation of the previous methods is that you cannot use complex conditional statements to find specific nodes. However you can user traversing methods to meet your custom filters. A possible general strategy would look like this:

```
from ete2 import Tree
t = Tree("((H:0.3,I:0.1):0.5, A:1, (B:0.4,(C:1,D:1):0.5):0.5);")
# Create a small function to filter your nodes
def conditional_function(node):
    if node.dist > 0.3:
        return True
    else:
        return False

# Use previous function to find matches. Note that we use the traverse
# method in the filter function. This will iterate over all nodes to
# assess if they meet our custom conditions and will return a list of
# matches.
matches = filter(conditional_function, t.traverse())
print len(matches), "nodes have ditance >0.3"

# depending on the complexity of your conditions you can do the same
# in just one line with the help of lambda functions:
matches = filter(lambda n: n.dist>0.3 and n.is_leaf(), t.traverse() )
print len(matches), "nodes have ditance >0.3 and are leaves"
```

Shortcuts

Finally, ETE implements a built-in method to find the first node matching a given name, which is one of the most common tasks needed for tree analysis. This can be done through the operator & (AND). Thus, TreeNode&"A" will always return the first node whose name is "A" and that is under the tree "MyTree". The syntaxis may seem confusing, but it can be very useful in some situations.

```
from ete2 import Tree
t = Tree("((H:0.3,I:0.1):0.5, A:1, (B:0.4,(C:1,(J:1, (F:1, D:1):0.5):0.5):0.5):0.5);")
# Get the node D in a very simple way
D = t&"D"
# Get the path from B to the root
node = D
path = []
while node.up:
  path.append(node)
  node = node.up
print t
# I substract D node from the total number of visited nodes
print "There are", len(path)-1, "nodes between D and the root"
# Using parentheses you can use by-operand search syntax as a node
# instance itself
Dsparent= (t&"C").up
Bsparent= (t&"B").up
Jsparent= (t&"J").up
# I check if nodes belong to certain partitions
print "It is", Dsparent in Bsparent, "that C's parent is under B's ancestor"
print "It is", Dsparent in Jsparent, "that C's parent is under J's ancestor"
```

3.1.6 Node annotation

Every node contains three basic attributes: name (TreeNode.name), branch length (TreeNode.dist) and branch support (TreeNode.support). These three values are encoded in the newick format. However, any extra data could be linked to trees. This is called tree annotation.

The TreeNode.add_feature() and TreeNode.add_features() methods allow to add extra attributes (features) to any node. The first allows to add one one feature at a time, while the second can be used to add many features with the same call.

Once extra features are added, you can access their values at any time during the analysis of a tree. To do so, you only need to access to the TreeNode.feature_name attributes.

Similarly, TreeNode.del_feature() can be used to delete an attribute.

```
import random
from ete2 import Tree
# Creates a tree
t = Tree( '((H:0.3,I:0.1):0.5, A:1, (B:0.4,(C:0.5,(J:1.3, (F:1.2, D:0.1):0.5):0.5):0.5):
```

```
# Let's locate some nodes using the get common ancestor method
ancestor=t.get_common_ancestor("J", "F", "C")
# the search_nodes method (I take only the first match )
A = t.search_nodes(name="A")[0]
# and using the shorcut to finding nodes by name
C= t&"C"
H= t&"H"
I= t&"I"

# Let's now add some custom features to our nodes. add_features can be
# used to add many features at the same time.
C.add_features(vowel=False, confidence=1.0)
A.add_features(vowel=True, confidence=0.5)
ancestor.add_features(nodetype="internal")

# Or, using the oneliner notation
(t&"H").add_features(vowel=False, confidence=0.2)

# But we can automatize this. (note that i will overwrite the previous
# values)
for leaf in t.traverse():
   if leaf.name in "AEIOU":
      leaf.add_features(vowel=True, confidence=random.random())
   else:
      leaf.add_features(vowel=False, confidence=random.random())

# Now we use these information to analyze the tree.
print "This tree has", len(t.search_nodes(vowel=True)), "vowel nodes"
print "Which are", [leaf.name for leaf in t.iter_leaves() if leaf.vowel==True]

# But features may refer to any kind of data, not only simple
# values. For example, we can calculate some values and store them
# within nodes.
#
# Let's detect leaf nodes under "ancestor" with distance higher thatn
# 1. Note that I'm traversing a subtree which starts from "ancestor"
matches = [leaf for leaf in ancestor.traverse() if leaf.dist>1.0]

# And save this pre-computed information into the ancestor node
ancestor.add_feature("long_branch_nodes", matches)

# Prints the precomputed nodes
print "These are nodes under ancestor with long branches", \
   [n.name for n in ancestor.long_branch_nodes]

# We can also use the add_feature() method to dynamically add new features.
label = raw_input("custom label:")
value = raw_input("custom label value:")
ancestor.add_feature(label, value)
print "Ancestor has now the [", label, "] attribute with value [", value, "]"
```

Unfortunately, newick format does not support adding extra features to a tree. Because of this drawback, several improved formats haven been (or are being) developed to read and write tree based information. Some of these new formats are based in a completely new standard (*Phylogenetic XML standards*), while others are extensions of the original newick formar (NHX http://phylosoft.org/NHX/http://phylosoft.org/NHX/).

Currently, ETE includes support for the New Hampshire eXtended format (NHX), which uses the original newick standard and adds the possibility of saving additional date related to each tree node. Here is an example of a extended newick representation in which extra information is added to an internal node:

```
(A:0.35,(B:0.72,(D:0.60,G:0.12):0.64[&&NHX:conf=0.01:name=INTERNAL]):0.56);
```

As you can notice, extra node features in the NHX format are enclosed between brackets. ETE is able to read and write features using such format, however, the encoded information is expected to be exportable as plain text.

The NHX format is automatically detected when reading a newick file, and the detected node features are added using the `TreeNode.add_feature()` method. Consequently, you can access the information by using the normal ETE's feature notation: `node.feature_name`. Similarly, features added to a tree can be included within the normal newick representation using the NHX notation. For this, you can call the `TreeNode.write()` method using the `features` argument, which is expected to be a list with the features names that you want to include in the newick string. Note that all nodes containing the suplied features will be exposed into the newick string. Use an empty features list (`features=[]`) to include all node's data into the newick string.

```
import random
from ete2 import Tree
# Creates a normal tree
t = Tree('((H:0.3,I:0.1):0.5, A:1,(B:0.4,(C:0.5,(J:1.3,(F:1.2, D:0.1):0.5):0.5):0.5):0.5
print t
# Let's locate some nodes using the get common ancestor method
ancestor=t.get_common_ancestor("J", "F", "C")
# Let's label leaf nodes
for leaf in t.traverse():
    if leaf.name in "AEIOU":
      leaf.add_features(vowel=True, confidence=random.random())
    else:
      leaf.add_features(vowel=False, confidence=random.random())

# Let's detect leaf nodes under "ancestor" with distance higher thatn
# 1. Note that I'm traversing a subtree which starts from "ancestor"
matches = [leaf for leaf in ancestor.traverse() if leaf.dist>1.0]

# And save this pre-computed information into the ancestor node
ancestor.add_feature("long_branch_nodes", matches)
print
print "NHX notation including vowel and confidence attributes"
print
print t.write(features=["vowel", "confidence"])
print
print "NHX notation including all node's data"
print

# Note that when all features are requested, only those with values
# equal to text-strings or numbers are considered. "long_branch_nodes"
# is not included into the newick string.
print t.write(features=[])
print
print "basic newick formats are still available"
print
print t.write(format=9, features=["vowel"])
# You don't need to do anything speciall to read NHX notation. Just
# specify the newick format and the NHX tags will be automatically
# detected.
nw = """
(((ADH2:0.1[&&NHX:S=human:E=1.1.1.1], ADH1:0.11[&&NHX:S=human:E=1.1.1.1])
:0.05[&&NHX:S=Primates:E=1.1.1.1:D=Y:B=100], ADHY:0.1[&&NHX:S=nematode:
E=1.1.1.1],ADHX:0.12[&&NHX:S=insect:E=1.1.1.1]):0.1[&&NHX:S=Metazoa:
E=1.1.1.1:D=N], (ADH4:0.09[&&NHX:S=yeast:E=1.1.1.1],ADH3:0.13[&&NHX:S=yeast:
E=1.1.1.1], ADH2:0.12[&&NHX:S=yeast:E=1.1.1.1],ADH1:0.11[&&NHX:S=yeast:E=1.1.1.1]):0.1
[&&NHX:S=Fungi])[&&NHX:E=1.1.1.1:D=N];
"""
# Loads the NHX example found at http://www.phylosoft.org/NHX/
t = Tree(nw)
# And access node’s attributes.
for n in t.traverse():
    if hasattr(n,"S"):
       print n.name, n.S
```

3.1.7 Modifying Tree Topology

Creating Trees from Scratch

If no arguments are passed to the TreeNode class constructor, an empty tree node will be returned. Such an orphan node can be used to populate a tree from scratch. For this, the TreeNode.up, and TreeNode.children attributes should never be used (unless it is strictly necessary). Instead, several methods exist to manipulate the topology of a tree:

TreeNode.populate(size[, names_library, ...])	Generates a random topology by populating current node.
TreeNode.add_child([child, name, dist, support])	Adds a new child to this node.
TreeNode.add_child([child, name, dist, support])	Adds a new child to this node.
TreeNode.delete([prevent_nondicotomic])	Deletes node from the tree structure.
TreeNode.detach()	Detachs this node (and all its descendants) from its parent ar

```
from ete2 import Tree
t = Tree() # Creates an empty tree
A = t.add_child(name="A") # Adds a new child to the current tree root
                           # and returns it
B = t.add_child(name="B") # Adds a second child to the current tree
                           # root and returns it
C = A.add_child(name="C") # Adds a new child to one of the branches
D = C.add_sister(name="D") # Adds a second child to same branch as
                             # before, but using a sister as the starting
                             # point
R = A.add_child(name="R") # Adds a third child to the
                           # branch. Multifurcations are supported
# Next, I add 6 random leaves to the R branch names_library is an
# optional argument. If no names are provided, they will be generated
# randomly.
R.populate(6, names_library=["r1","r2","r3","r4","r5","r6"])
# Prints the tree topology
print t
#                      /-C
#                     |
#                     |--D
#                     |
#           /--------|                              /-r4
#          |         |                    /--------|
#          |         |          /--------|          \-r3
#          |         |         |         |
#          |         |         |          \-r5
#          |          \--------|
# ---------|                   |                    /-r6
#          |                   |          /--------|
#          |                    \--------|          \-r2
#          |                             |
#          |                              \-r1
#          |
#           \-B
# a common use of the populate method is to quickly create example
# trees from scratch. Here we create a random tree with 100 leaves.
t = Tree()
t.populate(100)
```

Deleting (eliminating) and Removing (detaching) nodes

As currently implemented, there is a difference between detaching and deleting a node. The former disconnects a complete partition from the tree structure, so all its descendants are also disconnected from the tree. There are two methods to perform this action: `TreeNode.remove_child()` and `TreeNode.detach()`. In contrast, deleting a node means eliminating such node without affecting its descendants. Children from the deleted node are automatically connected to the next possible parent. This is better understood with the following example:

```
from ete2 import Tree
# Loads a tree. Note that we use format 1 to read internal node names
t = Tree(’((((H,K)D,(F,I)G)B,E)A,((L,(N,Q)O)J,(P,S)M)C);’, format=1)
print "original tree looks like this:"
# This is an alternative way of using "print t". Thus we have a bit
# more of control on how tree is printed. Here i print the tree
# showing internal node names
print t.get_ascii(show_internal=True)
#
#                                        /-H
#                              /D-------|
#                             |          \-K
#                    /B-------|
#                   |         |          /-F
#          /A-------|          \G-------|
#         |         |                    \-I
#         |         |
#         |          \-E
#-NoName--|
#         |                    /-L
#         |          /J-------|
#         |         |         |          /-N
#         |         |          \O-------|
#          \C-------|                    \-Q
#                   |
#                   |          /-P
#                    \M-------|
#                              \-S
# Get pointers to specific nodes
G = t.search_nodes(name="G")[0]
J = t.search_nodes(name="J")[0]
C = t.search_nodes(name="C")[0]
# If we remove J from the tree, the whole partition under J node will
# be detached from the tree and it will be considered an independent
# tree. We can do the same thing using two approaches: J.detach() or
# C.remove_child(J)
removed_node = J.detach() # = C.remove_child(J)
# if we know print the original tree, we will see how J partition is
# no longer there.
print "Tree after REMOVING the node J"
print t.get_ascii(show_internal=True)
#                                        /-H
#                              /D-------|
#                             |          \-K
#                    /B-------|
#                   |         |          /-F
#          /A-------|          \G-------|
#         |         |                    \-I
#         |         |
#-NoName--|          \-E
#         |
#         |                    /-P
#          \C------- /M-------|
#                              \-S
# however, if we DELETE the node G, only G will be eliminated from the
# tree, and all its descendants will then hang from the next upper
# node.
G.delete()
print "Tree after DELETING the node G"
print t.get_ascii(show_internal=True)
#                                        /-H
#                              /D-------|
#                             |          \-K
#                    /B-------|
#                   |         |--F
#          /A-------|         |
#         |         |          \-I
#         |         |
#-NoName--|          \-E
#         |
#         |                    /-P
#          \C------- /M-------|
#                              \-S
```

3.1.8 Pruning trees

Pruning a tree means to obtain the topology that connects a certain group of items by removing the unnecessary edges. To facilitate this task, ETE implements the `TreeNode.prune()` method, which can be used by providing the list of terminal and/or internal nodes that must be kept in the tree.

```
from ete2 import Tree
# Let's create simple tree
t = Tree('((((H,K),(F,I)G),E),((L,(N,Q)O),(P,S)));')
print "Original tree looks like this:"
print t
#
#                                        /-H
#                              /--------|
#                             |          \-K
#                    /--------|
#                   |         |          /-F
#          /--------|          \--------|
#         |         |                    \-I
#         |         |
#         |          \-E
#---------|
#         |                    /-L
#         |          /--------|
#         |         |         |          /-N
#         |         |          \--------|
#          \--------|                    \-Q
#                   |
#                   |          /-P
#                    \--------|
#                              \-S
# Prune the tree in order to keep only some leaf nodes.
t.prune(["H","F","E","Q", "P"])
print "Pruned tree"
print t
#
#                              /-F
#                    /--------|
#          /--------|          \-H
#         |         |
#---------|          \-E
#         |
#         |          /-Q
#          \--------|
#                    \-P
# Let’s re-create the same tree again
```

3.1.9 Concatenating trees

Given that all tree nodes share the same basic properties, they can be connected freely. In fact, any node can add a whole subtree as a child, so we can actually *cut & paste* partitions. To do so, you only need to call the `TreeNode.add_child()` method using another tree node as a first argument. If such a node is the root node of a different tree, you will concatenate two structures. But caution!!, this kind of operations may result into circular tree structures if add an node’s ancestor as a new node’s child. Some basic checks are internally performed by the ETE topology related methods, however, a fully qualified check of this issue would affect seriously the performance of the method. For this reason, users themselves should take care about not creating circular structures by mistake.

```
from ete2 import Tree
# Loads 3 independent trees
t1 = Tree(’(A,(B,C));’)
t2 = Tree(’((D,E), (F,G));’)
t3 = Tree(’(H, ((I,J), (K,L)));’)
print "Tree1:", t1
#            /-A
#  ---------|
#           |          /-B
#            \--------|
#                      \-C
print "Tree2:", t2
#                      /-D
#            /--------|
#           |          \-E
#  ---------|
#           |          /-F
#            \--------|
#                      \-G
print "Tree3:", t3
#            /-H
#           |
#  ---------|                    /-I
#           |          /--------|
#           |         |          \-J
#            \--------|
#                     |          /-K
#                      \--------|
#                                \-L
# Locates a terminal node in the first tree
A = t1.search_nodes(name=’A’)[0]
# and adds the two other trees as children.
A.add_child(t2)
A.add_child(t3)
print "Resulting concatenated tree:", t1
#                                          /-D
#                                /--------|
#                               |          \-E
#                      /--------|
#                     |         |          /-F
#                     |          \--------|
#            /--------|                    \-G
#           |         |
#           |         |          /-H
#           |         |         |
#           |          \--------|                    /-I
#           |                   |          /--------|
#  ---------|                   |         |          \-J
#           |                    \--------|
#           |                             |          /-K
#           |                              \--------|
#           |                                        \-L
#           |
#           |          /-B
#            \--------|
#                      \-C
```

3.1.10 Tree Rooting

Tree rooting is understood as the technique by with a given tree is conceptually polarized from more basal to more terminal nodes. In phylogenetics, for instance, this a crucial step prior to the interpretation of trees, since it will determine the evolutionary relationships among the species involved. The concept of rooted trees is different than just having a root node, which is always necessary to handle a tree data structure. Usually, the way in which a tree is differentiated between rooted and unrooted, is by counting the number of branches of the current root node. Thus, if the root node has more than two child branches, the tree is considered unrooted. By contrast, when only two main branches exist under the root node, the tree is considered rooted.

Having an unrooted tree means that any internal branch within the tree could be regarded as the root node, and there is no conceptual reason to place the root node where it is placed at the moment. Therefore, in an unrooted tree, there is no information about which internal nodes are more basal than others. By setting the root node between a given edge/branch of the tree structure the tree is polarized, meaning that the two branches under the root node are the most basal nodes. In practice, this is usually done by setting an **outgroup node**, which would represent one of these main root branches. The second one will be, obviously, the brother node. When you set an outgroup on unrooted trees, the multifurcations at the current root node are solved.

In order to root an unrooted tree or re-root a tree structure, ETE implements the `TreeNode.set_outgroup()` method, which is present in any tree node instance. Similarly, the `TreeNode.unroot()` method can be used to perform the opposite action.

```
from ete2 import Tree
# Load an unrooted tree. Note that three branches hang from the root
# node. This usually means that no information is available about
# which of nodes is more basal.
t = Tree(’(A,(H,F)(B,(E,D)));’)
print "Unrooted tree"
print t
#          /-A
#         |
#         |          /-H
#---------|---------|
#         |          \-F
#         |
#         |          /-B
#          \--------|
#                   |          /-E
#                    \--------|
#                              \-D
#
# Let’s define that the ancestor of E and D as the tree outgroup.  Of
# course, the definition of an outgroup will depend on user criteria.
ancestor = t.get_common_ancestor("E","D")
t.set_outgroup(ancestor)
print "Tree rooteda at E and D’s ancestor is more basal that the others."
print t
#
#                    /-B
#          /--------|
#         |         |          /-A
#         |          \--------|
#         |                   |          /-H
#---------|                    \--------|
#         |                              \-F
#         |
#         |          /-E
#          \--------|
#                    \-D
#
# Note that setting a different outgroup, a different interpretation
# of the tree is possible
t.set_outgroup( t&"A" )
print "Tree rooted at a terminal node"
print t
#                              /-H
#                    /--------|
#                   |          \-F
#          /--------|
#         |         |          /-B
#         |          \--------|
#---------|                   |          /-E
#         |                    \--------|
#         |                              \-D
#         |
#          \-A
```

Note that although **rooting** is usually regarded as a whole-tree operation, ETE allows to root subparts of the tree without affecting to its parent tree structure.

```
from ete2 import Tree
t = Tree('(((A,C),((H,F),(L,M))),((B,(J,K))(E,D)));')
print "Original tree:"
print t
#                              /-A
#                    /--------|
#                   |          \-C
#                   |
#          /--------|                    /-H
#         |         |          /--------|
#         |         |         |          \-F
#         |          \--------|
#         |                   |          /-L
#         |                    \--------|
#---------|                              \-M
#         |
#         |                    /-B
#         |          /--------|
#         |         |         |          /-J
#         |         |          \--------|
#          \--------|                    \-K
#                   |
#                   |          /-E
#                    \--------|
#                              \-D
#
# Each main branch of the tree is independently rooted.
node1 = t.get_common_ancestor("A","H")
node2 = t.get_common_ancestor("B","D")
node1.set_outgroup("H")
node2.set_outgroup("E")
print "Tree after rooting each node independently:"
print t
#
#                              /-F
#                             |
#                    /--------|                    /-L
#                   |         |          /--------|
#                   |         |         |          \-M
#                    |           \---------|
#          /---------|                     |           /-A
#         |          |                      \---------|
#         |          |                                 \-C
#         |          |
#         |           \-H
#---------|
#         |                      /-D
#         |           /---------|
#         |          |          |           /-B
#         |          |           \---------|
#          \---------|                     |           /-J
#                    |                      \---------|
#                    |                                 \-K
#                    |
#                     \-E
```

3.1.11 Working with branch distances

The branch length between one node an its parent is encoded as the `TreeNode.dist` attribute. Together with tree topology, branch lengths define the relationships among nodes.

Getting distances between nodes

The `TreeNode.get_distance()` method can be used to calculate the distance between two connected nodes. There are two ways of using this method: a) by querying the distance between two descendant nodes (two nodes are passed as arguments) b) by querying the distance between the current node and any other relative node (parental or descendant).

```
from ete2 import Tree

# Loads a tree with branch lenght information. Note that if no
# distance info is provided in the newick, it will be initialized with
# the default dist value = 1.0
nw = """((A:0.1, B:0.01):0.001, C:0.0001):1.0,
(((((D:0.00001:0,I:0):0,F:0):0,G:0):0,H:0):0,
E:0.000001):0.0000001):2.0;"""
t = Tree(nw)
print t
#                                /-A
#                     /---------|
#          /---------|           \-B
#         |          |
#         |           \-C
#         |
#         |                                                       /-D
#         |                                            /---------|
#---------|                                 /---------|           \-I
#         |                                |          |
#         |                      /---------|           \-F
#         |                     |          |
#         |           /---------|           \-G
#         |          |          |
#          \---------|           \-H
#                    |
#                     \-E
#
# Locate some nodes
A = t&"A"
C = t&"C"
# Calculate distance from current node
print "The distance between A and C is",  A.get_distance("C")
# Calculate distance between two descendants of current node
print "The distance between A and C is",  t.get_distance("A","C")
# Calculate the toplogical distance (number of nodes in between)
print "The number of nodes between A and D is ",  \
    t.get_distance("A","D", topology_only=True)
```

Additionally to this, ETE incorporates two more methods to calculate the most distant node from a given point in a tree. You can use the `TreeNode.get_farthest_node()` method to retrieve the most distant point from a node within the whole tree structure. Alternatively, `TreeNode.get_farthest_leaf()` will return the most distant descendant (always a leaf). If more than one node matches the farthest distance, the first occurrence is returned.

Distance between nodes can also be computed as the number of nodes between them (considering all branch lengths equal to 1.0). To do so, the **topology_only** argument must be set to **True** for all the above mentioned methods.

```
# Calculate the farthest node from E within the whole structure
farthest, dist = (t&"E").get_farthest_node()
print "The farthest node from E is", farthest.name, "with dist=", dist
# Calculate the farthest node from E within the whole structure,
# regarding the number of nodes in between as distance value
# Note that the result is diffent.
farthest, dist = (t&"E").get_farthest_node(topology_only=True)
print "The farthest (topologically) node from E is", \
    farthest.name, "with", dist, "nodes in between"
# Calculate farthest node from an internal node
farthest, dist = t.get_farthest_node()
print "The farthest node from root is is", farthest.name, "with dist=", dist
#
# The program results in the following information:
#
# The distance between A and C is 0.1011
# The distance between A and C is 0.1011
# The number of nodes between A and D is  8.0
# The farthest node from E is A with dist= 1.1010011
# The farthest (topologically) node from E is I with 5.0 nodes in between
# The farthest node from root is is A with dist= 1.101
```

getting midpoint outgroup

In order to obtain a balanced rooting of the tree, you can set as the tree outgroup that partition which splits the tree in two equally distant clusters (using branch lengths). This is called the midpoint outgroup.

The `TreeNode.get_midpoint_outgroup()` method will return the outgroup partition that splits current node into two balanced branches in terms of node distances.

```
from ete2 import Tree
# generates a random tree
```

```
t = Tree();
t.populate(15);
print t
#
#
#                    /-qogjl
#          /--------|
#         |          \-vxbgp
#         |
#         |          /-xyewk
#---------|         |
#         |         |                    /-opben
#         |         |                   |
#         |         |          /--------|                    /-xoryn
#          \--------|         |         |          /--------|
#                   |         |         |         |         |          /-wdima
#                   |         |          \--------|          \--------|
#                   |         |                   |                    \-qxovz
#                   |         |                   |
#                   |         |                    \-isngq
#                    \--------|
#                             |                    /-neqsc
#                             |                   |
#                             |                   |                              /-waxkv
#                             |          /--------|                    /--------|
#                             |         |         |          /--------|          \-djeoh
#                             |         |         |         |         |
#                             |         |          \--------|          \-exmsn
#                              \--------|                   |
#                                       |                   |          /-udspq
#                                       |                    \--------|
#                                       |                              \-buxpw
#                                       |
#                                        \-rkzwd
# Calculate the midpoint node
R = t.get_midpoint_outgroup()
# and set it as tree outgroup
t.set_outgroup(R)
print t
#                              /-opben
#                             |
#                    /--------|                    /-xoryn
#                   |         |          /--------|
#                   |         |         |         |          /-wdima
#                   |          \--------|          \--------|
#          /--------|                   |                    \-qxovz
#         |         |                   |
#         |         |                    \-isngq
#         |         |
#         |         |          /-xyewk
#         |          \--------|
#         |                   |          /-qogjl
#         |                    \--------|
#---------|                              \-vxbgp
#         |
#         |                    /-neqsc
#         |                   |
#         |                   |                              /-waxkv
#         |          /--------|                    /--------|
#         |         |         |          /--------|          \-djeoh
#         |         |         |         |         |
#         |         |          \--------|          \-exmsn
#          \--------|                   |
#                   |                   |          /-udspq
#                   |                    \--------|
#                   |                              \-buxpw
#                   |
#                    \-rkzwd
```

3.2 The Programmable Tree Drawing Engine

Contents

3.2.1 Overview

ETE's treeview extension provides a highly programmable drawing system to render any hierarchical tree structure as PDF, SVG or PNG images. Although several predefined visualization layouts are included with the default installation, custom styles can be easily created from scratch.

Image customization is performed through four elements: **a)** `TreeStyle`, setting general options about the image (shape, rotation, etc.), **b)** `NodeStyle`, which defines the specific aspect of each node (size, color, background, line type, etc.), **c)** node `faces.Face` which are small pieces of extra graphical information that can be added to nodes (text labels, images, graphs, etc.) **d)** a `layout` function, a normal python function that controls how node styles and faces are dynamically added to nodes.

Images can be rendered as **PNG**, **PDF** or **SVG** files using the `TreeNode.render()` method or interactively visualized using a built-in Graphical User Interface (GUI) invoked by the `TreeNode.show()` method.

3.2.2 Interactive visualization of trees

ETE's tree drawing engine is fully integrated with a built-in graphical user interface (GUI). Thus, ETE allows to visualize trees using an interactive interface that allows to explore and manipulate node's properties and tree topology. To start the visualization of a node (tree or subtree), you can simply call the `TreeNode.show()` method.

One of the advantages of this on-line GUI visualization is that you can use it to interrupt a given program/analysis, explore the tree, manipulate them, and continuing with the execution thread. Note that **changes made using the GUI will be kept after quiting the GUI**. This feature is specially useful for using during python sessions, in which analyses are performed interactively.

The GUI allows many operations to be performed graphically, however it does not implement all the possibilities of the programming toolkit.

```
from ete2 import Tree
t = Tree( "((a,b),c);" )
t.show()
```

3.2.3 Rendering trees as images

Tree images can be directly written as image files. SVG, PDF and PNG formats are supported. Note that, while PNG images are raster images, PDF and SVG pictures are rendered as vector graphics, thus allowing its later modification and scaling.

To generate an image, the `TreeNode.render()` method should be used instead of `TreeNode.show()`. The only required argument is the file name, whose extension will determine the image format (.PDF, .SVG or .PNG). Several parameters regarding the image size and resolution can be adjusted:

Argument	Description
`units`	"**px**": pixels, "**mm**": millimeters, "**in**": inches
`h`	height of the image in `units`.
`w`	weight of the image in `units`.
`dpi`	dots per inches.

Note: If `h` and `w` values are both provided, image size will be adjusted even if it requires to break the original aspect ratio of the image. If only one value (`h` or `w`) is provided, the other will be estimated to maintain aspect ratio. If no sizing values are provided, image will be adjusted to A4 dimensions.

```
from ete2 import Tree
t = Tree( "((a,b),c);" )
t.render("mytree.png", w=183, units="mm")
```

3.2.4 Customizing the aspect of trees

Image customization is performed through four main elements:

Tree style

The TreeStyle class can be used to create a custom set of options that control the general aspect of the tree image. Tree styles can be passed to the TreeNode.show() and TreeNode.render() methods. For instance, TreeStyle allows to modify the scale used to render tree branches or choose between circular or rectangular tree drawing modes.

```
from ete2 import Tree, TreeStyle

t = Tree( "((a,b),c);" )
circular_style = TreeStyle()
circular_style.mode = "c" # draw tree in circular mode
circular_style.scale = 20
t.render("mytree.png", w=183, units="mm", tree_style=circular_style)
```

> **Warning:** A number of parameters can be controlled through custom tree style objetcs, check TreeStyle documentation for a complete list of accepted values.

Some common uses include:

Show leaf node names, branch length and branch support

Change branch length scale (zoom in X)

Change branch separation between nodes (zoom in Y)

Rotate a tree

circular tree in 180 degrees

Add legend and title

```
from ete2 import Tree, TreeStyle, TextFace
t = Tree( "((a,b),c);" )
ts = TreeStyle()
ts.show_leaf_name = True
ts.title.add_face(TextFace("Hello ETE", fsize=20), column=0)
t.show(tree_style=ts)
```

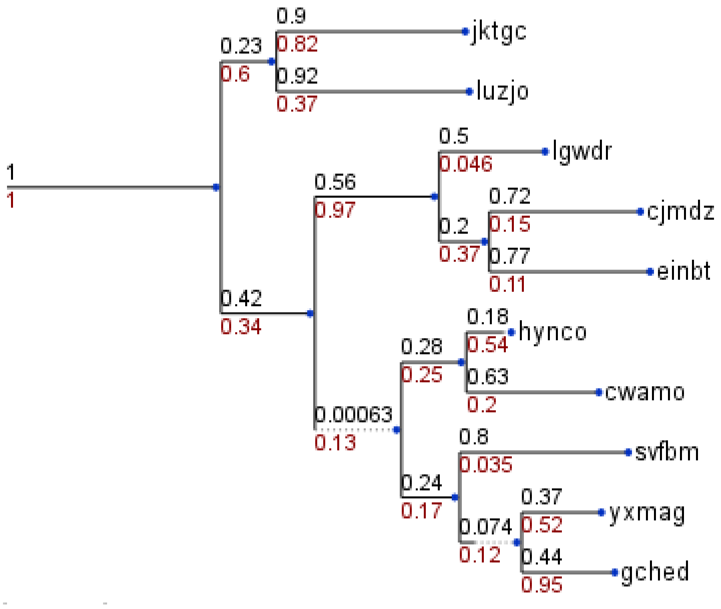


Figure 3.1: Automatically adds node names and branch information to the tree image:

```
from ete2 import Tree, TreeStyle
t = Tree()
t.populate(10, random_dist=True)
ts = TreeStyle()
ts.show_leaf_name = True
ts.show_branch_length = True
ts.show_branch_support = True
t.show(tree_style=ts)
```

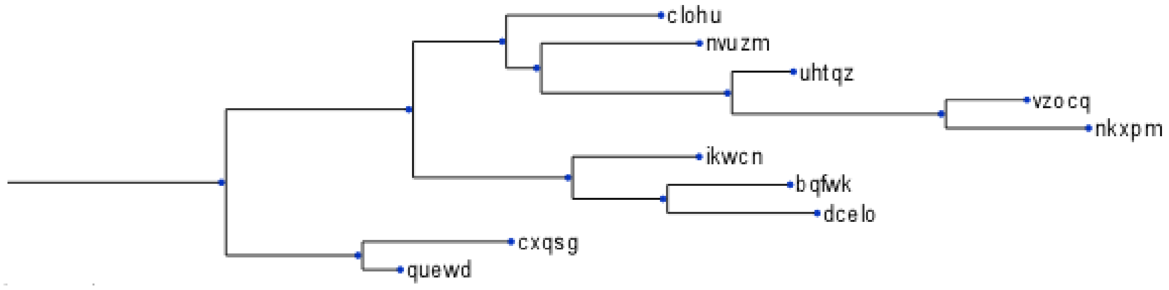


Figure 3.2: Increases the length of the tree by changing the scale:

```
from ete2 import Tree, TreeStyle
t = Tree()
t.populate(10, random_dist=True)
ts = TreeStyle()
ts.show_leaf_name = True
ts.scale =  120 # 120 pixels per branch length unit
t.show(tree_style=ts)
```

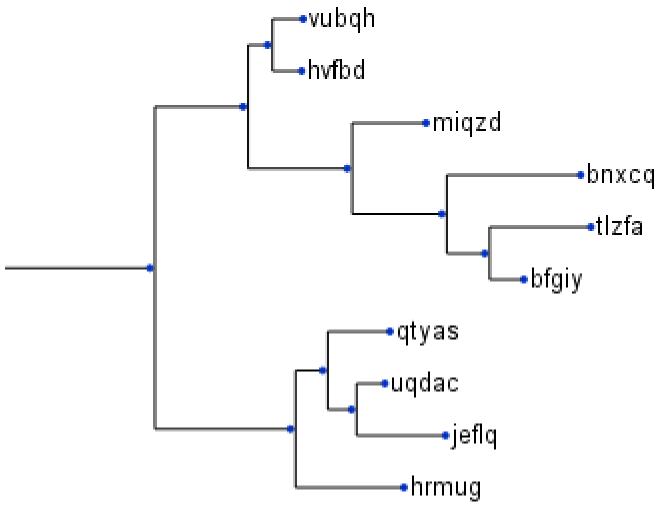


Figure 3.3: Increases the separation between leaf branches:

```
from ete2 import Tree, TreeStyle
t = Tree()
t.populate(10, random_dist=True)
ts = TreeStyle()
ts.show_leaf_name = True
ts.branch_vertical_margin = 10 # 10 pixels between adjacent branches
t.show(tree_style=ts)
```

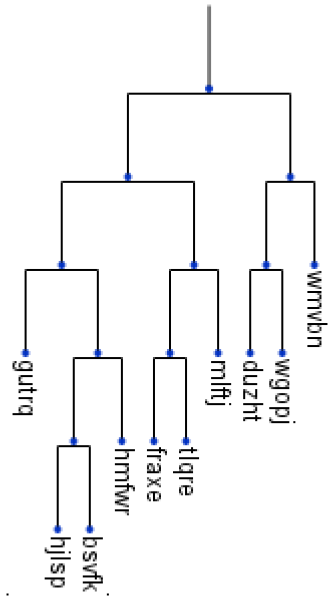


Figure 3.4: Draws a rectangular tree from top to bottom:

```
from ete2 import Tree, TreeStyle
t = Tree()
t.populate(10)
ts = TreeStyle()
ts.show_leaf_name = True
ts.rotation = 90
t.show(tree_style=ts)
```

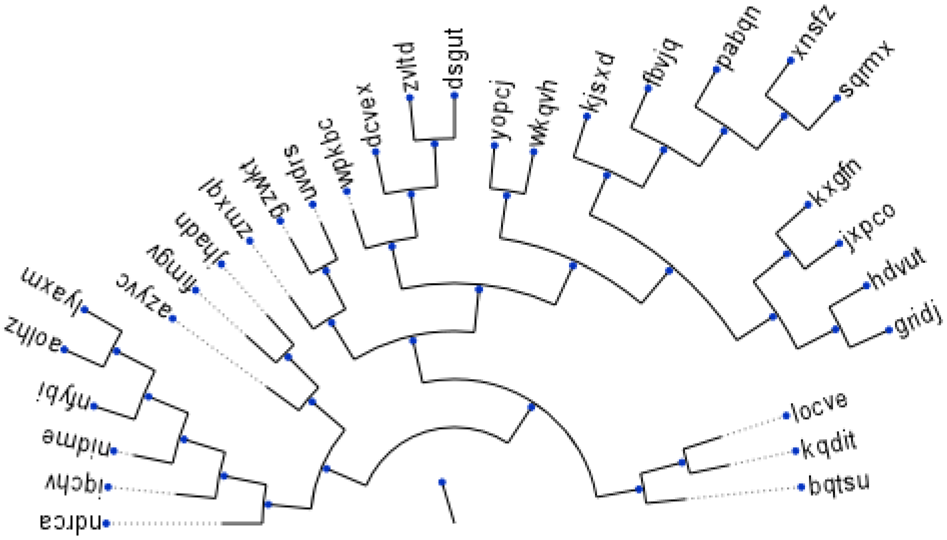


Figure 3.5: Draws a circular tree using a semi-circumference:

```
from ete2 import Tree, TreeStyle
t = Tree()
t.populate(30)
ts = TreeStyle()
ts.show_leaf_name = True
ts.mode = "c"
ts.arc_start = -180 # 0 degrees = 3 o'clock
ts.arc_span = 180
t.show(tree_style=ts)
```

Node style

Through the NodeStyle class the aspect of each single node can be controlled, including its size, color, background and branch type.

A node style can be defined statically and attached to several nodes:

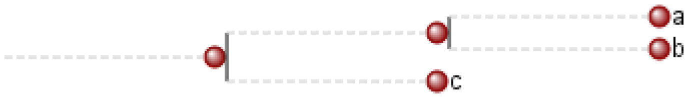


Figure 3.6: Simple tree in which the same style is applied to all nodes:

```
from ete2 import Tree, NodeStyle, TreeStyle
t = Tree( "((a,b),c);" )

# Basic tree style
ts = TreeStyle()
ts.show_leaf_name = True

# Draws nodes as small red spheres of diameter equal to 10 pixels
nstyle = NodeStyle()
nstyle["shape"] = "sphere"
nstyle["size"] = 10
nstyle["fgcolor"] = "darkred"

# Gray dashed branch lines
nstyle["hz_line_type"] = 1
nstyle["hz_line_color"] = "#cccccc"

# Applies the same static style to all nodes in the tree. Note that,
# if "nstyle" is modified, changes will affect to all nodes
for n in t.traverse():
   n.set_style(nstyle)

t.show(tree_style=ts)
```

If you want to draw nodes with different styles, an independent NodeStyle instance must be created for each node. Note that node styles can be modified at any moment by accessing the TreeNode.img_style attribute.

Static node styles, set through the set_style() method, will be attached to the nodes and exported as part of their information. For instance, TreeNode.copy() will replicate all node styles in the replicate tree. Note that node styles can be also modified on the fly through a layout function (see *layout functions*)

Node faces

Node faces are small pieces of graphical information that can be linked to nodes. For instance, text labels or external images could be linked to nodes and they will be plotted within the tree image.

Several types of node faces are provided by the main ete2 module, ranging from simple text (TextFace) and geometric shapes (CircleFace), to molecular sequence representations (SequenceFace), heatmaps and profile plots (ProfileFace). A complete list of available faces can be found at the ete2.treeview reference page..

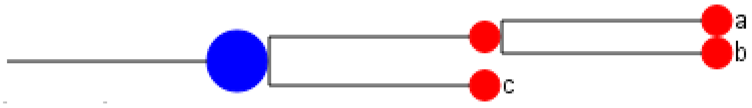


Figure 3.7: Simple tree in which the different styles are applied to each node:

```
from ete2 import Tree, NodeStyle, TreeStyle
t = Tree( "((a,b),c);" )

# Basic tree style
ts = TreeStyle()
ts.show_leaf_name = True

# Creates an independent node style for each node, which is
# initialized with a red foreground color.
for n in t.traverse():
   nstyle = NodeStyle()
   nstyle["fgcolor"] = "red"
   nstyle["size"] = 15
   n.set_style(nstyle)

# Let's now modify the aspect of the root node
t.img_style["size"] = 30
t.img_style["fgcolor"] = "blue"

t.show(tree_style=ts)
```

Faces position

Faces can be added to different areas around the node, namely **branch-right**, **branch-top**, **branch-bottom** or **aligned**. Each area represents a table in which faces can be added through the `TreeNode.add_face()` method. For instance, if two text labels want to be drawn bellow the branch line of a given node, a pair of `TextFace` faces can be created and added to the columns 0 and 1 of the **branch-bottom** area:

```
from ete2 import Tree, TreeStyle, TextFace
t = Tree( "((a,b),c);" )

# Basic tree style
ts = TreeStyle()
ts.show_leaf_name = True

# Add two text faces to different columns
t.add_face(TextFace("hola "), column=0, position = "branch-right")
t.add_face(TextFace("mundo!"), column=1, position = "branch-right")
t.show(tree_style=ts)
```

If you add more than one face to the same area and column, they will be piled up. See the following image as an example of face positions:

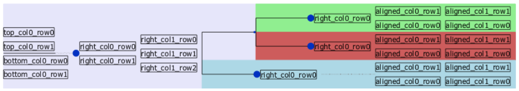


Figure 3.8: `Source code` used to generate the above image.

Note: Once a face object is created, it can be linked to one or more nodes. For instance, the same text label can be recycled and added to several nodes.

Face properties

Apart from the specific config values of each face type, all face instances contain same basic attributes that permit to modify general aspects such as margins, background colors, border, etc. A complete list of face attributes can be found in the general `Face` class documentation. Here is a very simple example:

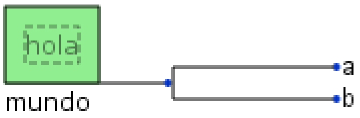


Figure 3.9: Basic use of face general attributes

```
from ete2 import Tree, TreeStyle, TextFace

t = Tree( "(a,b);" )

# Basic tree style
ts = TreeStyle()
ts.show_leaf_name = True

# Creates two faces
hola = TextFace("hola")
mundo = TextFace("mundo")

# Set some attributes
hola.margin_top = 10
hola.margin_right = 10
hola.margin_left = 10
hola.margin_bottom = 10
hola.opacity = 0.5 # from 0 to 1
hola.inner_border.width = 1 # 1 pixel border
hola.inner_border.type = 1  # dashed line
hola.border.width = 1
hola.background.color = "LightGreen"

t.add_face(hola, column=0, position = "branch-top")
t.add_face(mundo, column=1, position = "branch-bottom")

t.show(tree_style=ts)
```

layout functions

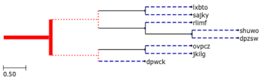
Layout functions act as pre-drawing hooking functions. This means, when a node is about to be drawn, it is first sent to a layout function. Node properties, style and faces can be then modified on the fly and return it to the drawer engine. Thus, layout functions can be understood as a collection of rules controlling how different nodes should be drawn.

```
from ete2 import Tree
t = Tree( "((((a,b),c), d), e);" )

def abc_layout(node):
    vowels = set(["a", "e", "i", "o", "u"])
    if node.name in vowels:

       # Note that node style are already initialized with the
       # default values

       node.img_style["size"] = 15
       node.img_style["color"] = "red"

# Basic tree style
ts = TreeStyle()
ts.show_leaf_name = True

# Add two text faces to different columns
t.add_face(TextFace("hola "), column=0, position = "branch-right")
t.add_face(TextFace("mundo!"), column=1, position = "branch-right")
t.show(tree_style=ts)
```

3.2.5 Combining styles, faces and layouts

Examples are probably the best way to show how ETE works:

Fixed node styles

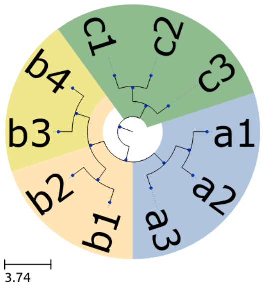


```
from ete2 import Tree, faces, AttrFace, TreeStyle, NodeStyle

def layout(node):
    # If node is a leaf, add the nodes name and a its scientific name
    if node.is_leaf():
        faces.add_face_to_node(AttrFace("name"), node, column=0)

def get_example_tree():

    t = Tree()
    t.populate(8)

    # Node style handling is no longer limited to layout functions. You
    # can now create fixed node styles and use them many times, save them
    # or even add them to nodes before drawing (this allows to save and
    # reproduce an tree image design)

    # Set bold red branch to the root node
    style = NodeStyle()
    style["fgcolor"] = "#0f0f0f"
```

```
        style["size"] = 0
        style["vt_line_color"] = "#ff0000"
        style["hz_line_color"] = "#ff0000"
        style["vt_line_width"] = 8
        style["hz_line_width"] = 8
        style["vt_line_type"] = 0 # 0 solid, 1 dashed, 2 dotted
        style["hz_line_type"] = 0
        t.set_style(style)

        #Set dotted red lines to the first two branches
        style1 = NodeStyle()
        style1["fgcolor"] = "#0f0f0f"
        style1["size"] = 0
        style1["vt_line_color"] = "#ff0000"
        style1["hz_line_color"] = "#ff0000"
        style1["vt_line_width"] = 2
        style1["hz_line_width"] = 2
        style1["vt_line_type"] = 2 # 0 solid, 1 dashed, 2 dotted
        style1["hz_line_type"] = 2
        t.children[0].img_style = style1
        t.children[1].img_style = style1

        # Set dashed blue lines in all leaves
        style2 = NodeStyle()
        style2["fgcolor"] = "#000000"
        style2["shape"] = "circle"
        style2["vt_line_color"] = "#0000aa"
        style2["hz_line_color"] = "#0000aa"
        style2["vt_line_width"] = 2
        style2["hz_line_width"] = 2
        style2["vt_line_type"] = 1 # 0 solid, 1 dashed, 2 dotted
        style2["hz_line_type"] = 1
        for l in t.iter_leaves():
            l.img_style = style2

        ts = TreeStyle()
        ts.layout_fn = layout
        ts.show_leaf_name = False

        return t, ts

    if __name__ == "__main__":
        t, ts = get_example_tree()
        t.render("node_style.png", w=400, tree_style=ts)
```

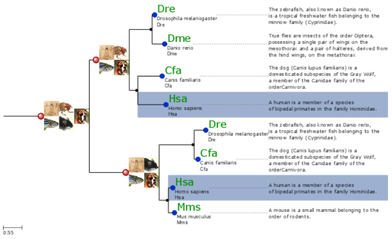
Node backgrounds

```
from ete2 import Tree, faces, AttrFace, TreeStyle, NodeStyle

def layout(node):
    if node.is_leaf():
        N = AttrFace("name", fsize=30)
        faces.add_face_to_node(N, node, 0, position="aligned")

def get_example_tree():
```



```
    # Set dashed blue lines in all leaves
    nst1 = NodeStyle()
    nst1["bgcolor"] = "LightSteelBlue"
    nst2 = NodeStyle()
    nst2["bgcolor"] = "Moccasin"
    nst3 = NodeStyle()
    nst3["bgcolor"] = "DarkSeaGreen"
    nst4 = NodeStyle()
    nst4["bgcolor"] = "Khaki"


    t = Tree("((((a1,a2),a3), ((b1,b2),(b3,b4))), ((c1,c2),c3));")

    n1 = t.get_common_ancestor("a1", "a2", "a3")
    n1.set_style(nst1)
    n2 = t.get_common_ancestor("b1", "b2", "b3", "b4")
    n2.set_style(nst2)
    n3 = t.get_common_ancestor("c1", "c2", "c3")
    n3.set_style(nst3)
    n4 = t.get_common_ancestor("b3", "b4")
    n4.set_style(nst4)
    ts = TreeStyle()
    ts.layout_fn = layout
    ts.show_leaf_name = False

    ts.mode = "c"
    return t, ts

if __name__ == "__main__":
    t, ts = get_example_tree()
    #t.render("node_background.png", w=400, tree_style=ts)
    t.show(tree_style=ts)
```

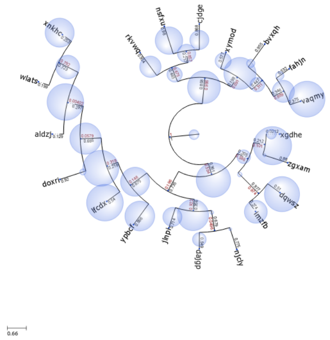
Img Faces


Dre
Drosophila melanogaster
Dre
The zebrafish, also known as Danio rerio, is a tropical freshwater fish belonging to the minnow family (Cyprinidae).
Dme
Danio rerio
Dme
True flies are insects of the order Diptera, possessing a single pair of wings on the mesothorax and a pair of halteres, derived from the hind wings, on the metathorax
Cfa
Canis familiaris
Cfa
The dog (Canis lupus familiaris) is a domesticated subspecies of the Gray Wolf, a member of the Canidae family of the orderCarnivora.
Hsa
Homo sapiens
Hsa
A human is a member of a species of bipedal primates in the family Hominidae.
Dre
Drosophila melanogaster
Dre
The zebrafish, also known as Danio rerio, is a tropical freshwater fish belonging to the minnow family (Cyprinidae).
Cfa
Canis familiaris
Cfa
The dog (Canis lupus familiaris) is a domesticated subspecies of the Gray Wolf, a member of the Canidae family of the orderCarnivora.
Hsa
Homo sapiens
Hsa
A human is a member of a species of bipedal primates in the family Hominidae.
Mms
Mus musculus
Mms
A mouse is a small mammal belonging to the order of rodents.
0.55

Note that images are attached to terminal and internal nodes.

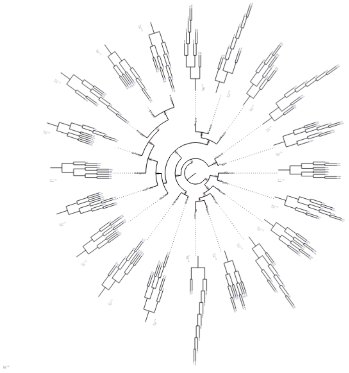
```
# Import Tree instance and faces module
from ete2 import Tree, faces, TreeStyle

# Loads an example tree
nw = """
(((Dre:0.008339,Dme:0.300613)1.000000:0.596401,
(Cfa:0.640858,Hsa:0.753230)1.000000:0.182035)1.000000:0.106234,
((Dre:0.271621,Cfa:0.046042)1.000000:0.953250,
(Hsa:0.061813,Mms:0.110769)1.000000:0.204419)1.000000:0.973467);
"""
t = Tree(nw)

# You can create any random tree containing the same leaf names, and
# layout will work equally
#
# t = Tree()
# Creates a random tree with 8 leaves using a given set of names
# t.populate(8, ["Dme", "Dre", "Hsa", "Ptr", "Cfa", "Mms"])

# Set the path in which images are located
img_path = "./"
# Create faces based on external images
humanFace = faces.ImgFace(img_path+"human.png")
mouseFace = faces.ImgFace(img_path+"mouse.png")
dogFace = faces.ImgFace(img_path+"dog.png")
chimpFace = faces.ImgFace(img_path+"chimp.png")
fishFace = faces.ImgFace(img_path+"fish.png")
flyFace = faces.ImgFace(img_path+"fly.png")

# Create a faces ready to read the name attribute of nodes
#nameFace = faces.TextFace(open("text").readline().strip(), fsize=20, fgcolor="#009
nameFace = faces.AttrFace("name", fsize=20, fgcolor="#009000")

# Create a conversion between leaf names and real names
code2name = {
        "Dre":"Drosophila melanogaster",
        "Dme":"Danio rerio",
        "Hsa":"Homo sapiens",
        "Ptr":"Pan troglodytes",
        "Mms":"Mus musculus",
        "Cfa":"Canis familiaris"
        }

# Creates a dictionary with the descriptions of each leaf name
code2desc = {
        "Dre":"""The zebrafish, also known as Danio rerio,
is a tropical freshwater fish belonging to the
minnow family (Cyprinidae).""",
        "Dme":"""True flies are insects of the order Diptera,
possessing a single pair of wings on the
mesothorax and a pair of halteres, derived from
the hind wings, on the metathorax""",
        "Hsa":"""A human is a member of a species
of bipedal primates in the family Hominidae.""",
        "Ptr":"""Chimpanzee, sometimes colloquially
chimp, is the common name for the
two extant species of ape in the genus Pan.""",
        "Mms":"""A mouse is a small mammal belonging to the
order of rodents.""",
        "Cfa": """The dog (Canis lupus familiaris) is a
domesticated subspecies of the Gray Wolf,
a member of the Canidae family of the
orderCarnivora."""
        }

# Creates my own layout function. I will use all previously created
# faces and will set different node styles depending on the type of
# node.
def mylayout(node):
    # If node is a leaf, add the nodes name and a its scientific
    # name
    if node.is_leaf():
        # Add an static face that handles the node name
        faces.add_face_to_node(nameFace, node, column=0)
        # We can also create faces on the fly
        longNameFace = faces.TextFace(code2name[node.name])
        faces.add_face_to_node(longNameFace, node, column=0)

        # text faces support multiline. We add a text face
        # with the whole description of each leaf.
        descFace = faces.TextFace(code2desc[node.name], fsize=10)
        descFace.margin_top = 10
        descFace.margin_bottom = 10
        descFace.border.margin = 1

        # Note that this faces is added in "aligned" mode
        faces.add_face_to_node(descFace, node, column=0, aligned=True)

        # Sets the style of leaf nodes
        node.img_style["size"] = 12
        node.img_style["shape"] = "circle"
    #If node is an internal node
    else:
        # Sets the style of internal nodes
        node.img_style["size"] = 6
        node.img_style["shape"] = "circle"
        node.img_style["fgcolor"] = "#000000"

    # If an internal node contains more than 4 leaves, add the
    # images of the represented species sorted in columns of 2
    # images max.
    if len(node)>=4:
        col = 0
        for i, name in enumerate(set(node.get_leaf_names())):
            if i>0 and i%2 == 0:
                col += 1
            # Add the corresponding face to the node
            if name.startswith("Dme"):
                faces.add_face_to_node(flyFace, node, column=col)
            elif name.startswith("Dre"):
                faces.add_face_to_node(fishFace, node, column=col)
            elif name.startswith("Mms"):
                faces.add_face_to_node(mouseFace, node, column=col)
            elif name.startswith("Ptr"):
                faces.add_face_to_node(chimpFace, node, column=col)
            elif name.startswith("Hsa"):
                faces.add_face_to_node(humanFace, node, column=col)
            elif name.startswith("Cfa"):
                faces.add_face_to_node(dogFace, node, column=col)

            # Modifies this node's style
            node.img_style["size"] = 16
            node.img_style["shape"] = "sphere"
            node.img_style["fgcolor"] = "#AA0000"

    # If leaf is "Hsa" (homo sapiens), highlight it using a
    # different background.
    if node.is_leaf() and node.name.startswith("Hsa"):
        node.img_style["bgcolor"] = "#9db0cf"

# And, finally, Visualize the tree using my own layout function
ts = TreeStyle()
ts.layout_fn = mylayout
t.render("img_faces.png", w=600, tree_style = ts)
```

Bubble tree maps



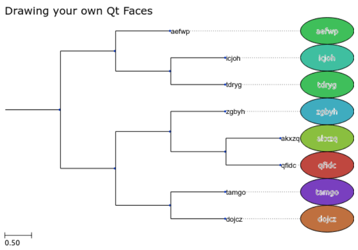
```
import random
from ete2 import Tree, TreeStyle, NodeStyle, faces, AttrFace, CircleFace

def layout(node):
    if node.is_leaf():
        # Add node name to laef nodes
        N = AttrFace("name", fsize=14, fgcolor="black")
        faces.add_face_to_node(N, node, 0)
    if "weight" in node.features:
        # Creates a sphere face whose size is proportional to node's
        # feature "weight"
        C = CircleFace(radius=node.weight, color="RoyalBlue", style="sphere")
        # Let's make the sphere transparent
        C.opacity = 0.3
        # And place as a float face over the tree
        faces.add_face_to_node(C, node, 0, position="float")

def get_example_tree():
    # Random tree
    t = Tree()
    t.populate(20, random_branches=True)

    # Some random features in all nodes
    for n in t.traverse():
        n.add_features(weight=random.randint(0, 50))

    # Create an empty TreeStyle
    ts = TreeStyle()

    # Set our custom layout function
    ts.layout_fn = layout

    # Draw a tree
    ts.mode = "c"

    # We will add node names manually
    ts.show_leaf_name = False
    # Show branch data
    ts.show_branch_length = True
    ts.show_branch_support = True

    return t, ts

if __name__ == "__main__":
    t, ts = get_example_tree()

    #t.render("bubble_map.png", w=600, dpi=300, tree_style=ts)
    t.show(tree_style=ts)
```

Trees within trees



```
    import random
    from ete2 import Tree, TreeStyle, NodeStyle, faces, AttrFace, TreeFace

    small_ts = TreeStyle()
    small_ts.show_leaf_name = True

    def layout(node):
        if node.is_leaf():
            # Add node name to laef nodes
            N = AttrFace("name", fsize=14, fgcolor="black")
            faces.add_face_to_node(N, node, 0)

            t = Tree()
            t.populate(10)

            T = TreeFace(t, small_ts)
            # Let's make the sphere transparent
            T.opacity = 0.8
            # And place as a float face over the tree
            faces.add_face_to_node(T, node, 1, position="aligned")

    # Random tree
    t = Tree()
    t.populate(20, random_branches=True)

    # Some random features in all nodes
    for n in t.traverse():
        n.add_features(weight=random.randint(0, 50))

    # Create an empty TreeStyle
    ts = TreeStyle()

    # Set our custom layout function
    ts.layout_fn = layout

    # Draw a tree
    ts.mode = "c"

    # We will add node names manually
    ts.show_leaf_name = False
    # Show branch data
    ts.show_branch_length = True
    ts.show_branch_support = True

    t.render("tree_faces.png", w=600, dpi=300, tree_style=ts)
    t.show(tree_style=ts)
```

Creating your custom interactive Item faces



Note that item faces shown in this image are not static. When the tree is view using the tree.show() method, you can interact with items.

```
# We will need to create Qt4 items
from PyQt4 import QtCore
from PyQt4.QtGui import QGraphicsRectItem, QGraphicsSimpleTextItem, \
    QGraphicsEllipseItem, QColor, QPen, QBrush

from ete2 import Tree, faces, TreeStyle, NodeStyle

# To play with random colors
import colorsys
import random

class InteractiveItem(QGraphicsRectItem):
    def __init__(self, *arg, **karg):
        QGraphicsRectItem.__init__(self, *arg, **karg)
        self.node = None
        self.label = None
        self.setCursor(QtCore.Qt.PointingHandCursor)
        self.setAcceptsHoverEvents(True)

    def hoverEnterEvent (self, e):
        # There are many ways of adding interactive elements. With the
        # following code, I show/hide a text item over my custom
        # DynamicItemFace
        if not self.label:
            self.label = QGraphicsRectItem()
            self.label.setParentItem(self)
            # This is to ensure that the label is rendered over the
            # rest of item children (default ZValue for items is 0)
            self.label.setZValue(1)
            self.label.setBrush(QBrush(QColor("white")))
            self.label.text = QGraphicsSimpleTextItem()
            self.label.text.setParentItem(self.label)

        self.label.text.setText(self.node.name)
        self.label.setRect(self.label.text.boundingRect())
        self.label.setVisible(True)

    def hoverLeaveEvent(self, e):
        if self.label:
            self.label.setVisible(False)

def random_color(h=None):
    """Generates a random color in RGB format."""
    if not h:
        h = random.random()
    s = 0.5
    l = 0.5
    return _hls2hex(h, l, s)

def _hls2hex(h, l, s):
    return '#%02x%02x%02x' %tuple(map(lambda x: int(x*255),
                                  colorsys.hls_to_rgb(h, l, s)))

def ugly_name_face(node, *args, **kargs):
    """ This is my item generator. It must receive a node object, and
    returns a Qt4 graphics item that can be used as a node face.
    """

    # receive an arbitrary number of arguments, in this case width and
    # height of the faces
    width = args[0][0]
    height = args[0][1]

    ## Creates a main master Item that will contain all other elements
    ## Items can be standard QGraphicsItem
    # masterItem = QGraphicsRectItem(0, 0, width, height)

    # Or your custom Items, in which you can re-implement interactive
    # functions, etc. Check QGraphicsItem doc for details.
    masterItem = InteractiveItem(0, 0, width, height)

    # Keep a link within the item to access node info
    masterItem.node = node

    # I dont want a border around the masterItem
    masterItem.setPen(QPen(QtCore.Qt.NoPen))

    # Add ellipse around text
    ellipse = QGraphicsEllipseItem(masterItem.rect())
    ellipse.setParentItem(masterItem)
    # Change ellipse color
    ellipse.setBrush(QBrush(QColor( random_color())))

    # Add node name within the ellipse
    text = QGraphicsSimpleTextItem(node.name)
    text.setParentItem(ellipse)
    text.setPen(QPen(QPen(QColor("white"))))

    # Center text according to masterItem size
    tw = text.boundingRect().width()
    th = text.boundingRect().height()
    center = masterItem.boundingRect().center()
    text.setPos(center.x()-tw/2, center.y()-th/2)

    return masterItem

def master_ly(node):
    if node.is_leaf():
        # Create an ItemFAce. First argument must be the pointer to
        # the constructor function that returns a QGraphicsItem. It
        # will be used to draw the Face. Next arguments are arbitrary,
        # and they will be forwarded to the constructor Face function.
        F = faces.DynamicItemFace(ugly_name_face, 100, 50)
        faces.add_face_to_node(F, node, 0, position="aligned")

def get_example_tree():

    t = Tree()
    t.populate(8, reuse_names=False)

    ts = TreeStyle()
    ts.layout_fn = master_ly
    ts.title.add_face(faces.TextFace("Drawing your own Qt Faces", fsize=15), 0)
    return t, ts

if __name__ == "__main__":
    t, ts = get_example_tree()

    #t.render("item_faces.png", h=400, tree_style=ts)
    # The interactive features are only available using the GUI
    t.show(tree_style=ts)
```

3.3 Phylogenetic Trees

Contents

3.3.1 Overview

Phylogenetic trees are the result of most evolutionary analyses. They represent the evolutionary relationships among a set of species or, in molecular biology, a set of homologous sequences.

The `PhyloTree` class is an extension of the base `Tree` object, providing a appropriate way to deal with phylogenetic trees. Thus, while leaves are considered to represent species (or sequences from a given species genome), internal nodes are considered ancestral nodes. A direct consequence of this is, for instance, that every split in the tree will represent a speciation or duplication event.

3.3.2 Linking Phylogenetic Trees with Multiple Sequence Alignments

`PhyloTree` instances allow molecular phylogenies to be linked to the Multiple Sequence Alignments (MSA). To associate a MSA with a phylogenetic tree you can use the `PhyloNode.link_to_alignment()` method. You can use the `alg_format` argument to specify its format (See `SeqGroup` documentation for available formats)

Given that Fasta format are not only applicable for MSA but also for **Unaligned Sequences**, you may also associate sequences of different lengths with tree nodes.

```
from ete2 import PhyloTree
fasta_txt = """
>seqA
```

```
MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEALNSYYASQTDDIKDRREEAH
>seqB
MAEIPDATIQQFMALTNVSHNIAVQY--EFGDLNEALNSYYAYQTDDQKDRREEAH
>seqC
MAEIPDATIQ---ALTNVSHNIAVQYLSEFGDLNEALNSYYASQTDDQPDRREEAH
>seqD
MAEAPDETIQQFMALTNVSHNIAVQYLSEFGDLNEAL--------------REEAH
"""

# Load a tree and link it to an alignment.
t = PhyloTree("(((seqA,seqB),seqC),seqD);")
t.link_to_alignment(alignment=fasta_txt, alg_format="fasta")
```

The same could be done at the same time the tree is being loaded, by using the `alignment` and `alg_format` arguments of `PhyloTree`.

```
# Load a tree and link it to an alignment.
t = PhyloTree("(((seqA,seqB),seqC),seqD);", alignment=fasta_txt, alg_format="fasta")
```

As currently implemented, sequence linking process is not strict, which means that a perfect match between all node names and sequences names **is not required**. Thus, if only one match is found between sequences names within the MSA file and tree node names, only one tree node will contain an associated sequence. Also, it is important to note that sequence linking is not limited to terminal nodes. If internal nodes are named, and such names find a match within the provided MSA file, their corresponding sequences will be also loaded into the tree structure. Once a MSA is linked, sequences will be available for every tree node through its `node.sequence` attribute.

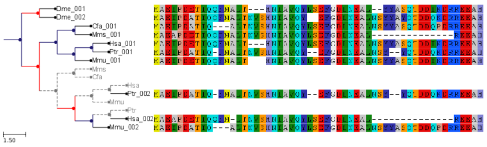
```
from ete2 import PhyloTree
fasta_txt = """
 >seqA
 MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEALNSYYASQTDDIKDRREEAH
 >seqB
 MAEIPDATIQQFMALTNVSHNIAVQY--EFGDLNEALNSYYAYQTDDQKDRREEAH
 >seqC
 MAEIPDATIQ---ALTNVSHNIAVQYLSEFGDLNEALNSYYASQTDDQPDRREEAH
 >seqD
 MAEAPDETIQQFMALTNVSHNIAVQYLSEFGDLNEAL--------------REEAH
"""
iphylip_txt = """
 4 76
      seqA   MAEIPDETIQ QFMALT---H NIAVQYLSEF GDLNEALNSY YASQTDDIKD RREEAHQFMA
      seqB   MAEIPDATIQ QFMALTNVSH NIAVQY--EF GDLNEALNSY YAYQTDDQKD RREEAHQFMA
      seqC   MAEIPDATIQ ---ALTNVSH NIAVQYLSEF GDLNEALNSY YASQTDDQPD RREEAHQFMA
      seqD   MAEAPDETIQ QFMALTNVSH NIAVQYLSEF GDLNEAL--- ---------- -REEAHQ---
             LTNVSHQFMA LTNVSH
             LTNVSH---- ------
             LTNVSH---- ------
             -------FMA LTNVSH
"""
# Load a tree and link it to an alignment. As usual, 'alignment' can
# be the path to a file or data in text format.
t = PhyloTree("(((seqA,seqB),seqC),seqD);", alignment=fasta_txt, alg_format="fasta")

#We can now access the sequence of every leaf node
print "These are the nodes and its sequences:"
for leaf in t.iter_leaves():
    print leaf.name, leaf.sequence
```

```
#seqD MAEAPDETIQQFMALTNVSHNIAVQYLSEFGDLNEAL--------------REEAH
#seqC MAEIPDATIQ---ALTNVSHNIAVQYLSEFGDLNEALNSYYASQTDDQPDRREEAH
#seqA MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEALNSYYASQTDDIKDRREEAH
#seqB MAEIPDATIQQFMALTNVSHNIAVQY--EFGDLNEALNSYYAYQTDDQKDRREEAH
#
# The associated alignment can be changed at any time
t.link_to_alignment(alignment=iphylip_txt, alg_format="iphylip")
# Let's check that sequences have changed
print "These are the nodes and its re-linked sequences:"
for leaf in t.iter_leaves():
    print leaf.name, leaf.sequence
#seqD MAEAPDETIQQFMALTNVSHNIAVQYLSEFGDLNEAL--------------REEAHQ----------FMALTNVSH
#seqC MAEIPDATIQ---ALTNVSHNIAVQYLSEFGDLNEALNSYYASQTDDQPDRREEAHQFMALTNVSH----------
#seqA MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEALNSYYASQTDDIKDRREEAHQFMALTNVSHQFMALTNVSH
#seqB MAEIPDATIQQFMALTNVSHNIAVQY--EFGDLNEALNSYYAYQTDDQKDRREEAHQFMALTNVSH----------
#
# The sequence attribute is considered as node feature, so you can
# even include sequences in your extended newick format!
print t.write(features=["sequence"], format=9)
#
#
# (((seqA[&&NHX:sequence=MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEALNSYYASQTDDIKDRREEAHQF
# MALTNVSHQFMALTNVSH],seqB[&&NHX:sequence=MAEIPDATIQQFMALTNVSHNIAVQY--EFGDLNEALNSY
# YAYQTDDQKDRREEAHQFMALTNVSH----------]),seqC[&&NHX:sequence=MAEIPDATIQ---ALTNVSHNIA
# VQYLSEFGDLNEALNSYYASQTDDQPDRREEAHQFMALTNVSH----------]),seqD[&&NHX:sequence=MAEAPD
# ETIQQFMALTNVSHNIAVQYLSEFGDLNEAL--------------REEAHQ----------FMALTNVSH]);
#
# And yes, you can save this newick text and reload it into a PhyloTree instance.
sametree = PhyloTree(t.write(features=["sequence"]))
print "Recovered tree with sequence features:"
print sametree
#
#                              /-seqA
#                    /--------|
#          /--------|          \-seqB
#         |         |
#---------|          \-seqC
#         |
#          \-seqD
#
print "seqA sequence:", (t&"seqA").sequence
# MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEALNSYYASQTDDIKDRREEAHQFMALTNVSHQFMALTNVSH
```

3.3.3 Visualization of phylogenetic trees

PhyloTree instances can benefit from all the features of the programmable drawing engine. However, a built-in phylogenetic layout is provided for convenience.

All PhyloTree instances are, by default, attached to such layout for tree visualization, thus allowing for in-place alignment visualization and evolutionary events labeling.



```
from ete2 import PhyloTree

alg = """
 >Dme_001
 MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEAL--YYASQTDDIKDRREEAH
 >Dme_002
 MAEIPDATIQQFMALTNVSHNIAVQY--EFGDLNEALNSYYAYQTDDQKDRREEAH
 >Cfa_001
 MAEIPDATIQ---ALTNVSHNIAVQYLSEFGDLNEALNSYYASQTDDQPDRREEAH
 >Mms_001
 MAEAPDETIQQFMALTNVSHNIAVQYLSEFGDLNEAL--------------REEAH
 >Hsa_001
 MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEALNSYYASQTDDIKDRREEAH
 >Ptr_002
 MAEIPDATIQ-FMALTNVSHNIAVQY--EFGDLNEALNSY--YQTDDQKDRREEAH
 >Mmu_002
 MAEIPDATIQ---ALTNVSHNIAVQYLSEFGDLNEALNSYYASQTDDQPDRREEAH
 >Hsa_002
 MAEAPDETIQQFM-LTNVSHNIAVQYLSEFGDLNEAL--------------REEAH
 >Mmu_001
 MAEIPDETIQQFMALT---HNIAVQYLSEFGDLNEALNSYYASQTDDIKDRREEAH
 >Ptr_001
 MAEIPDATIQ-FMALTNVSHNIAVQY--EFGDLNEALNSY--YQTDDQKDRREEAH
 >Mmu_001
 MAEIPDATIQ---ALTNVSHNIAVQYLSEFGDLNEALNSYYASQTDDQPDRREEAH
"""

# Performs a tree reconciliation analysis
gene_tree_nw = '((Dme_001,Dme_002),(((Cfa_001,Mms_001),((Hsa_001,Ptr_001),Mmu_001)),(Ptr
species_tree_nw = "((((Hsa, Ptr), Mmu), (Mms, Cfa)), Dme);"
genetree = PhyloTree(gene_tree_nw)
sptree = PhyloTree(species_tree_nw)
recon_tree, events = genetree.reconcile(sptree)
recon_tree.link_to_alignment(alg)

# Visualize the reconciled tree
recon_tree.render("phylotree.png", w=750)
```

3.3.4 Adding taxonomic information

PhyloTree instances allow to deal with leaf names and species names separately. This is useful when working with molecular phylogenies, in which node names usually represent sequence identifiers. Species names will be stored in the PhyloNode.species attribute of each leaf node. The method PhyloNode.get_species() can be used obtain the set of species names found under a given internal node (speciation or duplication event). Often, sequence names do contain species information as a part of the name, and ETE can parse this information automatically.

There are three ways to establish the species of the different tree nodes:

- Default: The three first letters of node's name represent the species
- The species code of each node is dynamically created based on node's name
- The species code of each node is manually set.

Automatic control of species info

```
from ete2 import PhyloTree
# Reads a phylogenetic tree (using default species name encoding)
t = PhyloTree("(((Hsa_001,Ptr_001),(Cfa_001,Mms_001)),(Dme_001,Dme_002));")
#                              /-Hsa_001
#                    /--------|
#                   |          \-Ptr_001
#          /--------|
#         |         |          /-Cfa_001
#         |          \--------|
#---------|                    \-Mms_001
#         |
#         |          /-Dme_001
#          \--------|
#                    \-Dme_002
#
# Prints current leaf names and species codes
print "Deafult mode:"
for n in t.get_leaves():
    print "node:", n.name, "Species name:", n.species
# node: Dme_001 Species name: Dme
# node: Dme_002 Species name: Dme
# node: Hsa_001 Species name: Hsa
# node: Ptr_001 Species name: Ptr
# node: Cfa_001 Species name: Cfa
# node: Mms_001 Species name: Mms
```

Automatic (and custom) control of the species info

The default behavior can be changed by using the `PhyloNode.set_species_naming_function()` method or by using the `sp_naming_function` argument of the `PhyloTree` class. Note that, using the `sp_naming_function` argument, the whole tree structure will be initialized to use the provided parsing function to obtain species name information. `PhyloNode.set_species_naming_function()` (present in all tree nodes) can be used to change the behavior in a previously loaded tree, or to set different parsing function to different parts of the tree.

```
from ete2 import PhyloTree
# Reads a phylogenetic tree
t = PhyloTree("(((Hsa_001,Ptr_001),(Cfa_001,Mms_001)),(Dme_001,Dme_002));")

# Let's use our own leaf name parsing function to obtain species
# names. All we need to do is create a python function that takes
# node's name as argument and return its corresponding species name.
def get_species_name(node_name_string):
    # Species code is the first part of leaf name (separated by an
    #  underscore character)
    spcode = node_name_string.split("_")[0]
    # We could even translate the code to complete names
    code2name = {
      "Dme":"Drosophila melanogaster",
      "Hsa":"Homo sapiens",
      "Ptr":"Pan troglodytes",
      "Mms":"Mus musculus",
      "Cfa":"Canis familiaris"
      }
    return code2name[spcode]

# Now, let’s ask the tree to use our custom species naming function
t.set_species_naming_function(get_species_name)
print "Custom mode:"
for n in t.get_leaves():
    print "node:", n.name, "Species name:", n.species

# node: Dme_001 Species name: Drosophila melanogaster
# node: Dme_002 Species name: Drosophila melanogaster
# node: Hsa_001 Species name: Homo sapiens
# node: Ptr_001 Species name: Pan troglodytes
# node: Cfa_001 Species name: Canis familiaris
# node: Mms_001 Species name: Mus musculus
```

Manual control of the species info

To disable the automatic generation of species names based on node names, a `None` value can be passed to the `PhyloNode.set_species_naming_function()` function. From then on, species attribute will not be automatically updated based on the name of nodes and it could be controlled manually.

```
from ete2 import PhyloTree
# Reads a phylogenetic tree
t = PhyloTree("(((Hsa_001,Ptr_001),(Cfa_001,Mms_001)),(Dme_001,Dme_002));")

# Of course, you can disable the automatic generation of species
# names. To do so, you can set the species naming function to
# None. This is useful to set the species names manually or for
# reading them from a newick file. Other wise, species attribute would
# be overwriten
mynewick = """
(((Hsa_001[&&NHX:species=Human],Ptr_001[&&NHX:species=Chimp]),
(Cfa_001[&&NHX:species=Dog],Mms_001[&&NHX:species=Mouse])),
(Dme_001[&&NHX:species=Fly],Dme_002[&&NHX:species=Fly]));
"""
t = PhyloTree(mynewick, sp_naming_function=None)
print "Disabled mode (manual set)"
for n in t.get_leaves():
    print "node:", n.name, "Species name:", n.species

# node: Dme_001 Species name: Fly
# node: Dme_002 Species name: Fly
# node: Hsa_001 Species name: Human
# node: Ptr_001 Species name: Chimp
# node: Cfa_001 Species name: Dog
# node: Mms_001 Species name: Mouse
```

Full Example: `Species aware trees.`

3.3.5 Detecting evolutionary events

There are several ways to automatically detect duplication and speciation nodes. ETE provides two methodologies: One implements the algorithm described in Huerta-Cepas (2007) and is based on the species overlap (SO) between partitions and thus does not depend on the availability of a species tree. The second, which requires the comparison between the gene tree and a previously defined species tree, implements a strict tree reconciliation algorithm (Page and Charleston, 1997). By detecting evolutionary events, orthology and paralogy relationships among sequences can also be inferred. Find a comparison of both methods in Marcet-Houben and Gabaldon (2009).

Species Overlap (SO) algorithm

In order to apply the SO algorithm, you can use the `PhyloNode.get_descendant_evol_events()` method (it will detect all evolutionary events under the current node) or the `PhyloNode.get_my_evol_events()` method (it will detect only the evolutionary events in which current node, a leaf, is involved).

By default the **species overlap score (SOS) threshold** is set to 0.0, which means that a single species in common between two node branches will rise a duplication event. This has been shown to perform the best with real data, however you can adjust the threshold using the `sos_thr` argument present in both methods.

```
from ete2 import PhyloTree
# Loads an example tree
nw = """
((Dme_001,Dme_002),(((Cfa_001,Mms_001),((Hsa_001,Ptr_001),Mmu_001)),
(Ptr_002,(Hsa_002,Mmu_002))));
"""
t = PhyloTree(nw)
print t
#                    /-Dme_001
#          /--------|
#         |          \-Dme_002
#         |
#         |                              /-Cfa_001
#         |                    /--------|
#---------|                   |          \-Mms_001
#         |          /--------|
#         |         |         |                    /-Hsa_001
#         |         |         |          /--------|
#         |         |          \--------|          \-Ptr_001
#          \--------|                   |
#                   |                    \-Mmu_001
#                   |
#                   |          /-Ptr_002
#                    \--------|
#                             |          /-Hsa_002
#                              \--------|
#                                        \-Mmu_002
#
# To obtain all the evolutionary events involving a given leaf node we
# use get_my_evol_events method
matches = t.search_nodes(name="Hsa_001")
human_seq = matches[0]
# Obtains its evolutionary events
```

```
events = human_seq.get_my_evol_events()
# Print its orthology and paralogy relationships
print "Events detected that involve Hsa_001:"
for ev in events:
    if ev.etype == "S":
        print '   ORTHOLOGY RELATIONSHIP:', ','.join(ev.in_seqs), "<====>", ','.join(ev.
    elif ev.etype == "D":
        print '   PARALOGY RELATIONSHIP:', ','.join(ev.in_seqs), "<====>", ','.join(ev.

# Alternatively, you can scan the whole tree topology
events = t.get_descendant_evol_events()
# Print its orthology and paralogy relationships
print "Events detected from the root of the tree"
for ev in events:
    if ev.etype == "S":
        print '   ORTHOLOGY RELATIONSHIP:', ','.join(ev.in_seqs), "<====>", ','.join(ev.
    elif ev.etype == "D":
        print '   PARALOGY RELATIONSHIP:', ','.join(ev.in_seqs), "<====>", ','.join(ev.

# If we are only interested in the orthology and paralogy relationship
# among a given set of species, we can filter the list of sequences
#
# fseqs is a function that, given a list of sequences, returns only
# those from human and mouse
fseqs = lambda slist: [s for s in slist if s.startswith("Hsa") or s.startswith("Mms")]
print "Paralogy relationships among human and mouse"
for ev in events:
    if ev.etype == "D":
        # Prints paralogy relationships considering only human and
        # mouse. Some duplication event may not involve such species,
        # so they will be empty
        print '   PARALOGY RELATIONSHIP:', \
            ','.join(fseqs(ev.in_seqs)), \
            "<====>",\
            ','.join(fseqs(ev.out_seqs))

# Note that besides the list of events returned, the detection
# algorithm has labeled the tree nodes according with the
# predictions. We can use such lables as normal node features.
dups = t.search_nodes(evoltype="D") # Return all duplication nodes
```

Tree reconciliation algorithm

Tree reconciliation algorithm uses a predefined species tree to infer all the necessary genes losses that explain a given gene tree topology. Consequently, duplication and separation nodes will strictly follow the species tree topology.

To perform a tree reconciliation analysis over a given node in a molecular phylogeny you can use the `PhyloNode.reconcile()` method, which requires a species `PhyloTree` as its first argument. Leaf node names in the the species are expected to be the same species codes in the gene tree (see taxonomic_info). All species codes present in the gene tree should appear in the species tree.

As a result, the `PhyloNode.reconcile()` method will label the original gene tree nodes as duplication or speciation, will return the list of inferred events, and will return a new **reconcilied tree** (`PhyloTree` instance), in which inferred gene losses are present and labeled.

```
from ete2 import PhyloTree

# Loads a gene tree and its corresponding species tree. Note that
# species names in sptree are the 3 firs letters of leaf nodes in
# genetree.
gene_tree_nw = ’((Dme_001,Dme_002),(((Cfa_001,Mms_001),((Hsa_001,Ptr_001),Mmu_001)),(Ptr
species_tree_nw = "((((Hsa, Ptr), Mmu), (Mms, Cfa)), Dme);"
genetree = PhyloTree(gene_tree_nw)
sptree = PhyloTree(species_tree_nw)
print genetree
#                    /-Dme_001
#          /--------|
#         |          \-Dme_002
#         |
#         |                              /-Cfa_001
#         |                    /--------|
#---------|                   |          \-Mms_001
#         |          /--------|
#         |         |         |                    /-Hsa_001
#         |         |         |          /--------|
#         |         |          \--------|          \-Ptr_001
#          \--------|                   |
#                   |                    \-Mmu_001
#                   |
#                   |          /-Ptr_002
#                    \--------|
#                             |          /-Hsa_002
#                              \--------|
#                                        \-Mmu_002
#
# Let’s reconcile our genetree with the species tree
recon_tree, events = genetree.reconcile(sptree)
# a new "reconcilied tree" is returned. As well as the list of
# inferred events.
print "Orthology and Paralogy relationships:"
for ev in events:
    if ev.etype == "S":
        print ’ORTHOLOGY RELATIONSHIP:’, ’,’.join(ev.inparalogs), "<====>", ’,’.join(ev.
    elif ev.etype == "D":
        print ’PARALOGY RELATIONSHIP:’, ’,’.join(ev.inparalogs), "<====>", ’,’.join(ev.o
# And we can explore the resulting reconciled tree
print recon_tree
# You will notice how the reconcilied tree is the same as the gene
# tree with some added branches. They are inferred gene losses.
#
#
#                    /-Dme_001
#          /--------|
#         |          \-Dme_002
#         |
#         |                              /-Cfa_001
#         |                    /--------|
#         |                   |          \-Mms_001
#---------|          /--------|
#         |         |         |                    /-Hsa_001
#         |         |         |          /--------|
#         |         |          \--------|          \-Ptr_001
#         |         |                   |
#         |         |                    \-Mmu_001
#          \--------|
#                   |                    /-Mms
#                   |          /--------|
#                   |         |          \-Cfa
#                   |         |
#                   |         |                              /-Hsa
#                    \--------|                    /--------|
#                             |          /--------|          \-Ptr_002
#                             |         |         |
#                             |         |          \-Mmu
#                              \--------|
#                                       |                    /-Ptr
#                                       |          /--------|
#                                        \--------|          \-Hsa_002
#                                                 |
#                                                  \-Mmu_002
#
# And we can visualize the trees using the default phylogeny
# visualization layout
genetree.show()
recon_tree.show()
```

A closer look to the evolutionary event object

Both methods, species overlap and tree reconciliation, can be used to label each tree node as a duplication or speciation event. Thus, the `PhyloNode.evoltype` attribute of every node will be set to one of the following states: `D` (Duplication), `S` (Speciation) or `L` gene loss.

Additionally, a list of all the detected events is returned. Each event is a python object of type `phylo.EvolEvent`, containing some basic information about each event (`etype`, `in_seqs`, `out_seqs`, `node`):

If an event represents a duplication, `in_seqs` **are all paralogous** to `out_seqs`. Similarly, if an event represents a speciation, `in_seqs` **are all orthologous** to `out_seqs`.

3.3.6 Relative dating phylogenetic nodes

In molecular phylogeny, nodes can be interpreted as evolutionary events. Therefore, they represent duplication or speciation events. In the case of gene duplication events, nodes can also be assigned to a certain point in a relative temporal scale. In other words, you can obtain a relative dating of all the duplication events detected.

Although **absolute dating is always preferred and more precise**, topological dating provides a faster approach to compare the relative age of paralogous sequences (read this for a comparison with other methods, such as the use of synonymous substitution rates as a proxy to the divergence time).

Some applications of topological dating can be found in Huerta-Cepas et al, 2007 or, more recently, in Huerta-Cepas et al, 2011 or Kalinka et al, 2001.

Implementation

The aim of relative dating is to establish a gradient of ages among sequences. For this, a reference species needs to be fixed, so the gradient of ages will be referred to that referent point.

Thus, if our reference species is *Human*, we could establish the following gradient of species:

- (1) Human -> (2) Other Primates -> (3) Mammals -> (4) Vertebrates

So, nodes in a tree can be assigned to one of the above categories depending on the sequences grouped. For instance:

- A node with only human sequences will be mapped to (1).
- A node with human and orangutan sequences will be mapped to (2)
- A node with human a fish sequences will be mapped to (4)

This simple calculation can be done automatically by encoding the gradient of species ages as Python dictionary.

```
relative_dist = {
    "human": 0, # human
    "chimp": 1, # Primates non human
    "rat":   2, # Mammals non primates
    "mouse": 2, # Mammals non primates
    "fish":  3  # Vertebrates non mammals
    }
```

Once done, ETE can check the relative age of any tree node. The `PhyloNode.get_age()` method can be used to that purpose.

For example, let's consider the following gene tree:

```
#                         /-humanA
#                    /---|
#                   |     \-chimpA
#               /Dup1
#              |    |     /-humanB
#          /---|     \---|
#         |    |          \-chimpB
#     /---|    |
#    |    |     \-mouseA
#    |    |
#    |     \-fish
#-Dup3
#    |               /-humanC
#    |          /---|
#    |     /---|     \-chimpC
#    |    |    |
#     \Dup2     \-humanD
#         |
#         |     /-ratC
#          \---|
#               \-mouseC
```

the expected node dating would be:

- Dup1 will be assigned to primates (most distant species is chimp). `Dup1.get_age(relative_distances)` will return 1

- Dup2 will be assigned to mammals [2] (most distant species are rat and mouse). `Dup2.get_age(relative_distances)` will return 2
- Dup3 will be assigned to mammals [3] (most distant species is fish). `Dup3.get_age(relative_distances)` will return 3

```
from ete2 import PhyloTree
# Creates a gene phylogeny with several duplication events at
# different levels. Note that we are using the default method for
# detecting the species code of leaves (three first lettes in the node
# name are considered the species code).
nw = """
((Dme_001,Dme_002),(((Cfa_001,Mms_001),((((Hsa_001,Hsa_003),Ptr_001)
,Mmu_001),((Hsa_004,Ptr_004),Mmu_004))),(Ptr_002,(Hsa_002,Mmu_002))));
"""
t = PhyloTree(nw)
print "Original tree:",
print t
#
#             /-Dme_001
#   /--------|
#  |          \-Dme_002
#  |
#  |                              /-Cfa_001
#  |                    /--------|
#  |                   |          \-Mms_001
#  |                   |
#--|                   |                                        /-Hsa_001
#  |                   |                              /--------|
#  |          /--------|                    /--------|          \-Hsa_003
#  |         |         |                   |         |
#  |         |         |          /--------|          \-Ptr_001
#  |         |         |         |         |
#  |         |         |         |          \-Mmu_001
#  |         |          \--------|
#   \--------|                   |                    /-Hsa_004
#            |                   |          /--------|
#            |                    \--------|          \-Ptr_004
#            |                             |
#            |                              \-Mmu_004
#            |
#            |          /-Ptr_002
#             \--------|
#                      |          /-Hsa_002
#                       \--------|
#                                 \-Mmu_002
# Create a dictionary with relative ages for the species present in
# the phylogenetic tree.  Note that ages are only relative numbers to
# define which species are older, and that different species can
# belong to the same age.
species2age = {
  'Hsa': 1, # Homo sapiens (Hominids)
  'Ptr': 2, # P. troglodytes (primates)
  'Mmu': 2, # Macaca mulata (primates)
  'Mms': 3, # Mus musculus (mammals)
  'Cfa': 3, # Canis familiaris (mammals)
  'Dme': 4  # Drosophila melanogaster (metazoa)
}
```

```
# We can translate each number to its correspondig taxonomic number
age2name = {
  1:"hominids",
  2:"primates",
  3:"mammals",
  4:"metazoa"
}
event1= t.get_common_ancestor("Hsa_001", "Hsa_004")
event2=t.get_common_ancestor("Hsa_001", "Hsa_002")
print
print "The duplication event leading to the human sequences Hsa_001 and "+\
    "Hsa_004 is dated at: ", age2name[event1.get_age(species2age)]
print "The duplication event leading to the human sequences Hsa_001 and "+\
    "Hsa_002 is dated at: ", age2name[event2.get_age(species2age)]
# The duplication event leading to the human sequences Hsa_001 and Hsa_004
# is dated at:  primates
#
# The duplication event leading to the human sequences Hsa_001 and Hsa_002
# is dated at:  mammals
```

Warning: Note that relative distances will vary depending on your reference species.

3.3.7 Automatic rooting (outgroup detection)

Two methods are provided to assist in the automatic rooting of phylogenetic trees. Since tree nodes contain relative age information (based on the species code autodetection), the same relative age dictionaries can be used to detect the farthest and oldest node in a tree to given sequences.

`PhyloNode.get_farthest_oldest_node()` and `PhyloNode.get_farthest_oldest_leaf()` can be used for that purpose.

3.4 Clustering Trees

Contents

3.4.1 Overview

Cluster analysis is the assignment of a set of observations into subsets (called clusters) so that observations in the same cluster are similar in some sense. Clustering is a method of unsupervised learning, and a common technique for statistical data analysis used in many fields, including machine learning, data mining, pattern recognition, image analysis and bioinformatics. Hierarchical clustering creates a hierarchy of clusters which may be represented in a tree structure called a dendrogram. The root of the tree consists of a single cluster containing all observations, and the leaves correspond to individual observations. [The Wikipedia project Jun-2009].

ETE provides special ClusterNode (alias ClusterTree) instances to deal with trees associated to a clustering analysis. The basic difference between Tree and ClusterTree is that leaf nodes in a cluster-tree are linked to numerical profiles. Such profiles are expected to represent the data used to generate the clustering tree. In other words, trees are bound to numerical arrays.

```
#              /-A
#          ---|
#              \-B
#
# #Names   col1    col2    col3
# A        1.1     0.1     1.33
# B        2.0     1.0     2.0
```

Based on this, ClusterTree instances provide several several clustering validation techniques that help in the analysis of cluster quality. Currently, inter and intra-cluster distances, cluster std.deviation, Silhouette analysis and Dunn indexes are supported. In addition, ClusterTree nodes can be visualized using the ProfileFace face type, which can represent cluster profiles in different ways, such as line plots, heatmaps or bar plots.

Although this type of trees are intended to be used for clustering results, any tree that can be linked to a table (i.e. phylogenetic profiles) could be loaded using this data type, thus taking advantage of the profile visualization modes, etc.

3.4.2 Loading ClusterTrees

A ClusterTree can be linked to a numerical matrix by using the text_array argument.

```
from ete2 import ClusterTree

# Example of a minimalistic numerical matrix. It is encoded as a text
# string for convenience, but it usally be loaded from a text file.
matrix = """
#Names\tcol1\tcol2\tcol3\tcol4\tcol5\tcol6\tcol7
A\t-1.23\t-0.81\t1.79\t0.78\t-0.42\t-0.69\t0.58
B\t-1.76\t-0.94\t1.16\t0.36\t0.41\t-0.35\t1.12
C\t-2.19\t0.13\t0.65\t-0.51\t0.52\t1.04\t0.36
D\t-1.22\t-0.98\t0.79\t-0.76\t-0.29\t1.54\t0.93
E\t-1.47\t-0.83\t0.85\t0.07\t-0.81\t1.53\t0.65
F\t-1.04\t-1.11\t0.87\t-0.14\t-0.80\t1.74\t0.48
G\t-1.57\t-1.17\t1.29\t0.23\t-0.20\t1.17\t0.26
H\t-1.53\t-1.25\t0.59\t-0.30\t0.32\t1.41\t0.77
"""
print "Example numerical matrix"
print matrix
# #Names   col1    col2    col3    col4    col5    col6    col7
# A        -1.23   -0.81   1.79    0.78    -0.42   -0.69   0.58
# B        -1.76   -0.94   1.16    0.36    0.41    -0.35   1.12
# C        -2.19   0.13    0.65    -0.51   0.52    1.04    0.36
# D        -1.22   -0.98   0.79    -0.76   -0.29   1.54    0.93
# E        -1.47   -0.83   0.85    0.07    -0.81   1.53    0.65
# F        -1.04   -1.11   0.87    -0.14   -0.80   1.74    0.48
# G        -1.57   -1.17   1.29    0.23    -0.20   1.17    0.26
# H        -1.53   -1.25   0.59    -0.30   0.32    1.41    0.77
```

```
#
#
# We load a tree structure whose leaf nodes correspond to rows in the
# numerical matrix. We use the text_array argument to link the tree
# with numerical matrix.
t = ClusterTree("(((A,B),(C,(D,E))),(F,(G,H)));", text_array=matrix)
```

Alternatively, you can re-link the tree (or a sub-part of it) to a new matrix using the `ClusterNode.link_to_arraytable()` method.

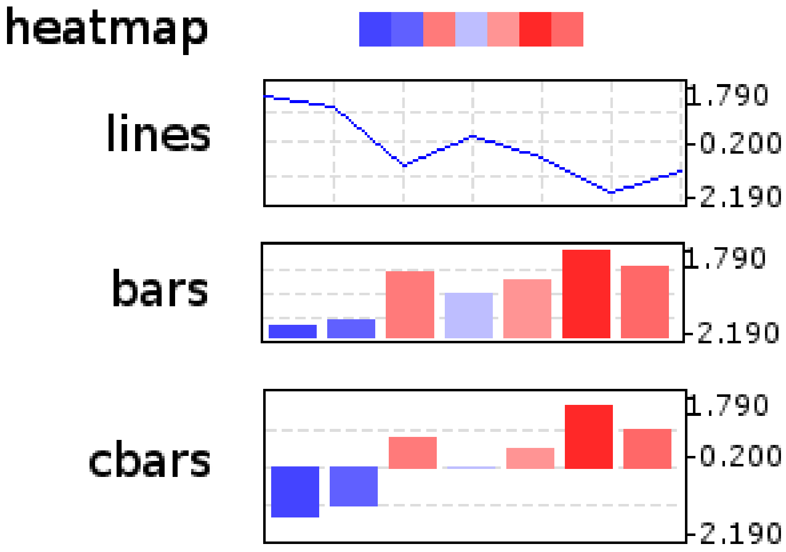
```
t = ClusterTree("(((A,B),(C,(D,E))),(F,(G,H)));")
t.children[0].link_to_arraytable(matrix1)
t.children[1].link_to_arraytable(matrix2)
```

Once the tree is linked to a table of profiles, the following node properties will be available: `PhyloNode.profile`, `PhyloNode.deviation`, `PhyloNode.silhoutte`, `PhyloNode.intercluster_dist`, `PhyloNode.intracluster_dist`, `PhyloNode.dunn`.

Similarly, the following methods are provide for convenience `PhyloNode.iter_leaf_profiles()`, `PhyloNode.get_leaf_profiles()`, `PhyloNode.get_silhouette()` and `PhyloNode.get_dunn()` methods.

3.4.3 Visualization of matrix associated Trees

Clustering or not, any ClusterTree instance, associated to a numerical matrix, can be visualized together with the graphical representation of its node's numeric profiles. To this end, the `ProfileFace` class is provided by the `treeview` module. This face type can represent a node's numeric profile in four different ways:



Additionally, three basic layouts are provided that use different styles of ProfileFace instances: **heatmap**, **line_profiles**, **bar_profiles**, **cbar_profiles**.

```
# Import Tree instance and faces module
from ete2 import ClusterTree
```

```
# Example of a minimalistic numerical matrix. It is encoded as a text
# string for convenience, but it usally be loaded from a text file.
matrix = """
#Names\tcol1\tcol2\tcol3\tcol4\tcol5\tcol6\tcol7
A\t-1.23\t-0.81\t1.79\t0.78\t-0.42\t-0.69\t0.58
B\t-1.76\t-0.94\t1.16\t0.36\t0.41\t-0.35\t1.12
C\t-2.19\t0.13\t0.65\t-0.51\t0.52\t1.04\t0.36
D\t-1.22\t-0.98\t0.79\t-0.76\t-0.29\t1.54\t0.93
E\t-1.47\t-0.83\t0.85\t0.07\t-0.81\t1.53\t0.65
F\t-1.04\t-1.11\t0.87\t-0.14\t-0.80\t1.74\t0.48
G\t-1.57\t-1.17\t1.29\t0.23\t-0.20\t1.17\t0.26
H\t-1.53\t-1.25\t0.59\t-0.30\t0.32\t1.41\t0.77
"""
print "Example numerical matrix"
print matrix
# #Names    col1    col2    col3    col4    col5    col6    col7
# A         -1.23   -0.81   1.79    0.78    -0.42   -0.69   0.58
# B         -1.76   -0.94   1.16    0.36    0.41    -0.35   1.12
# C         -2.19   0.13    0.65    -0.51   0.52    1.04    0.36
# D         -1.22   -0.98   0.79    -0.76   -0.29   1.54    0.93
# E         -1.47   -0.83   0.85    0.07    -0.81   1.53    0.65
# F         -1.04   -1.11   0.87    -0.14   -0.80   1.74    0.48
# G         -1.57   -1.17   1.29    0.23    -0.20   1.17    0.26
# H         -1.53   -1.25   0.59    -0.30   0.32    1.41    0.77
#
#
# We load a tree structure whose leaf nodes correspond to rows in the
# numerical matrix. We use the text_array argument to link the tree
# with numerical matrix.
t = ClusterTree("(((A,B),(C,(D,E))),(F,(G,H)));", text_array=matrix)
# Try the default layout using ProfileFaces
t.show("heatmap")
t.show("cluster_cbars")
t.show("cluster_bars")
t.show("cluster_lines")
```

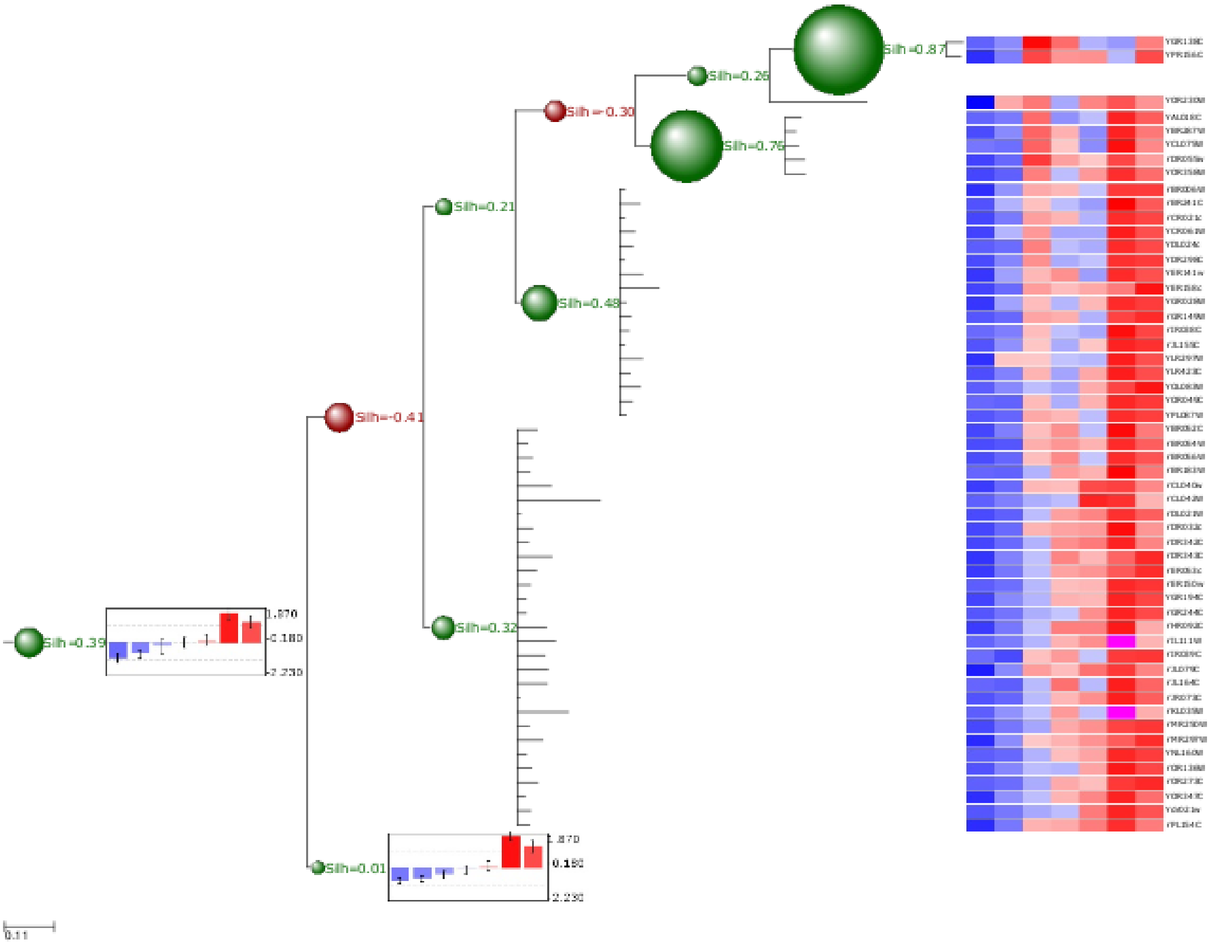
3.4.4 Cluster Validation Example

If associated matrix represents the dataset used to produce a given tree, clustering validation values can be used to assess the quality of partitions. To do so, you will need to set the distance function that was used to calculate distances among items (leaf nodes). ETE implements three common distance methods in bioinformatics : **euclidean**, **pearson** correlation and **spearman** correlation distances.

In the following example, a microarray clustering result is visualized and validated using ETE.

Image resulting from a microarray clustering validation analysis. Red bubbles represent a bad silhouette index (S<0), while green represents good silhouette index (S>0). Size of bubbles is proportional to the Silhouette index. Internal nodes are drawn with the average expression profile grouped by their partitions. Leaf node profiles are shown as a heatmap.

```
from ete2 import ClusterTree, TreeStyle, AttrFace, ProfileFace, TextFace
from ete2.treeview.faces import add_face_to_node

# To operate with numbers efficiently
import numpy
```


Silh=0.87
Silh=0.26
Silh=0.76
Silh=-0.30
Silh=0.21
Silh=0.48
Silh=-0.41
Silh=0.32
Silh=0.39
1.870
0.180
-2.230
Silh=0.01
1.870
0.180
-2.230
0.11

```
PATH = "./"
# Loads tree and array
t = ClusterTree(PATH+"diauxic.nw", PATH+"diauxic.array")

# nodes are linked to the array table
array =  t.arraytable

# Calculates some stats on the matrix. Needed to establish the color
# gradients.
matrix_dist = [i for r in xrange(len(array.matrix))\
               for i in array.matrix[r] if numpy.isfinite(i)]
matrix_max = numpy.max(matrix_dist)
matrix_min = numpy.min(matrix_dist)
matrix_avg = matrix_min+((matrix_max-matrix_min)/2)

# Creates a profile face that will represent node’s profile as a
# heatmap
profileFace  = ProfileFace(matrix_max, matrix_min, matrix_avg, \
                                          200, 14, "heatmap")
cbarsFace = ProfileFace(matrix_max,matrix_min,matrix_avg,200,70,"cbars")
nameFace = AttrFace("name", fsize=8)
# Creates my own layout function that uses previous faces
def mylayout(node):
    # If node is a leaf
    if node.is_leaf():
        # And a line profile
        add_face_to_node(profileFace, node, 0, aligned=True)
        node.img_style["size"]=0
        add_face_to_node(nameFace, node, 1, aligned=True)

    # If node is internal
    else:
        # If silhouette is good, creates a green bubble
        if node.silhouette>0:
            validationFace = TextFace("Silh=%0.2f" %node.silhouette,
                                      "Verdana", 10, "#056600")
            node.img_style["fgcolor"]="#056600"
        # Otherwise, use red bubbles
        else:
            validationFace = TextFace("Silh=%0.2f" %node.silhouette,
                                      "Verdana", 10, "#940000")
            node.img_style["fgcolor"]="#940000"

        # Sets node size proportional to the silhouette value.
        node.img_style["shape"]="sphere"
        if node.silhouette<=1 and node.silhouette>=-1:
            node.img_style["size"]= 15+int((abs(node.silhouette)*10)**2)

        # If node is very internal, draw also a bar diagram
        # with the average expression of the partition
        add_face_to_node(validationFace, node, 0)
        if len(node)>100:
            add_face_to_node(cbarsFace, node, 1)

# Use my layout to visualize the tree
ts = TreeStyle()
ts.layout_fn = mylayout
```

```
t.show(tree_style=ts)
```

3.5 The PhylomeDB API

PhylomeDB is a public database for complete collections of gene phylogenies (phylomes). It allows users to interactively explore the evolutionary history of genes through the visualization of phylogenetic trees and multiple sequence alignments. Moreover, phylomeDB provides genome-wide orthology and paralogy predictions which are based on the analysis of the phylogenetic trees. The automated pipeline used to reconstruct trees aims at providing a high-quality phylogenetic analysis of different genomes , including Maximum Likelihood or Bayesian tree inference, alignment trimming and evolutionary model testing. PhylomeDB includes also a public download section with the complete set of trees, alignments and orthology predictions.

ETE's phylomeDB extension provides an access API to the main PhylomeDB database, thus allowing to search for and fetch precomputed gene phylogenies.

3.5.1 Basis of the phylomeDB API usage

In order to explore the database resources, you have to create a connector to the database, which will be used to query it. To do so, you must use the **PhylomeDBConnector** class and specify the parameters of the DB connection.

The PhylomeDBConnector constructor will return a pointer to the DB that you can use to perform queries. All methods starting by **get_** can be used to retrieve information from the database. A complete list of available methods can be found in the ETE's programming guide (available at http://ete.cgenomics.orgete.cgenomics.org) or explored by executing **dir(PhylomeDBConnector)** in a python console.

3.5.2 PhylomeDB structure

A phylome includes thousands of gene trees associated to the different genes/proteins of a given species. Thus, for example, the human phylome includes more than 20.000 phylogenetic trees; on per human gene. Moreover, the same gene may be associated to different trees within the same phylome differing only in the evolutionary model that assumed to reconstruct the phylogeny.

Given that each phylogenetic tree was reconstructed using a a single gene as the seed sequence to find homologous in other species, the tree takes the name from the seed sequence.

You can obtain a full list of phylomes through the **get_phylomes()** and a full list of seed sequence in a phylome using the **get_seed_ids()** method. Phylogenetic trees within a given phylome were reconstructed in a context of a fixed set of species. In order to obtain the list of proteomes included in a phylome, use the** get_proteomes_in_phylome()** method. PhylomeDB uses its own sequence identifiers, but you can use the **search_id()** to find a match from an external sequence ID.

Each phylome is the collection of all trees associated to a given species. Thus, the human phylome will contain thousands of phylogenetic trees. Each gene/protein in a phylome may be associated to different trees, testing, for example, different evolutionary models. Thus when you query the database for a gene phylogeny you have to specify from which phylome and which specific tree. Alternatively, you can query for the best tree in a given phylomes, which will basically return the best likelihood tree for the queried gene/protein. The get_tree and get_best_tree methods carry out such operations. When trees are fetched from the phylomeDB database, the are automatically converted to the PhyloTree class, thus allowing to operate with them as phylogenetic trees.

3.5.3 Going phylogenomic scale

Just to show you how to explore a complete phylome:

3.6 Phylogenetic XML standards

New in version 2.1. From version 2.1, ETE provides support for NeXML and PhyloXML phylogenetic XML standards, both reading and writing. These standards allow to encode complex phylogenetic data, and therefore they are not limited to trees. Although ETE is mainly focused on allowing transparent interaction with trees, it also provides basic I/O methods to data of different type.

Essentially, NexML and PhyloXML files are intended to encode collections of phylogenetic data. Such information can be converted to a collection Python objects sorted in a hierarchical way. A specific Python class exists for every element encoded documented by the NeXML and PhyloXML formats. This is possible thanks to the the general purpose Python drivers available for both formats (http://ete.cgenomics.org/phyloxml-and-nexml-python-parsers). ETE will use such drivers to access XML data, and it will also convert tree data into PhyloTree objects. In practice, conversions will occur transparently. NeXML and PhyloXML files are loaded using their specific root classes, provided by the main ETE module, and all the information will become available as a collection of Python objects internally sorted according to the original XML hierarchy. New in version 2.1.

3.6.1 NeXML

NeXML(http://nexml.org) is an exchange standard for representing phyloinformatic data inspired by the commonly used NEXUS format, but more robust and easier to process.

Reading NeXML projects

Nexml projects are handled through the `Nexml` base class. To load a NexML file, the `Nexml.build_from_file()` method can be used.

```
from ete2 import Nexml

nexml_prj = Nexml()
nexml_prj.build_from_file("/path/to/nexml_example.xml")
```

Note that the ETE parser will read the provided XML file and convert all elements into python instances, which will be hierarchically connected to the Nexml root instance.

Every NeXML XML element has its own python class. Content and attributes can be handled through the "**set_**" and "**get_**" methods existing in all objects. Nexml classes can be imported from the `ete2.nexml` module.

```
from ete2 import Nexml, nexml
nexml_prj = Nexml()
nexml_meta = nexml.LiteralMeta(datatype="double", property="branch_support", content=1.0
nexml_prj.add_meta(nexml_meta)
nexml_prj.export()
```

```
# Will produce:
#
# <Nexml xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Nexml">
#    <meta datatype="double" content="1.0" property="branch_support" xmlns:xsi="http://w
# </Nexml>
```

NeXML trees

NeXML tree elements are automatically converted into PhyloTree instances, containing all ETE functionality (traversing, drawing, etc) plus normal NeXML attributes.

In the Nexml standard, trees are represented as plain lists of nodes and edges. ETE will convert such lists into tree topologies, in which every node will contain a nexml_node and nexml_edge attribute. In addition, each tree node will have a nexml_tree attribute (i.e. NEXML->FloatTree) , which can be used to set the nexml properties of the subtree represented by each node. Note also that node.dist and node.name features will be linked to node.nexml_edge.length and node.nexml_node.label, respectively.

```
from ete2 import Nexml
# Create an empty Nexml project
nexml_project = Nexml()

# Load content from NeXML file
nexml_project.build_from_file("trees.xml")

# All XML elements are within the project instance.
# exist in each element to access their attributes.
print "Loaded Taxa:"
for taxa in  nexml_project.get_otus():
    for otu in taxa.get_otu():
        print "OTU:", otu.id

# Extracts all the collection of trees in the project
tree_collections = nexml_project.get_trees()
# Select the first collection
collection_1 = tree_collections[0]

# print the topology of every tree
for tree in  collection_1.get_tree():
    # trees contain all the nexml information in their "nexml_node",
    # "nexml_tree", and "nexml_edge" attributes.
    print "Tree id", tree.nexml_tree.id
    print tree
    for node in tree.traverse():
        print "node", node.nexml_node.id, "is associated with", node.nexml_node.otu, "OT


# Output:
# ==========
# Loaded Taxa:
# OTU: t1
# OTU: t2
# OTU: t3
# OTU: t4
# OTU: t5
# Tree id tree1
#
#                /-n5(n5)
#           /---|
#          |     \-n6(n6)
#      /---|
#     |    |     /-n8(n8)
# ----|     \---|
#     |          \-n9(n9)
#     |
#      \-n2(n2)
# node n1 is associated with None OTU
# node n3 is associated with None OTU
# node n2 is associated with t1 OTU
# node n4 is associated with None OTU
# node n7 is associated with None OTU
# node n5 is associated with t3 OTU
# node n6 is associated with t2 OTU
# node n8 is associated with t5 OTU
# node n9 is associated with t4 OTU
# Tree id tree2
#
#                /-tree2n5(n5)
#           /---|
#          |     \-tree2n6(n6)
#      /---|
#     |    |     /-tree2n8(n8)
# ----|     \---|
#     |          \-tree2n9(n9)
#     |
#      \-tree2n2(n2)
# node tree2n1 is associated with None OTU
# node tree2n3 is associated with None OTU
# node tree2n2 is associated with t1 OTU
# node tree2n4 is associated with None OTU
# node tree2n7 is associated with None OTU
# node tree2n5 is associated with t3 OTU
# node tree2n6 is associated with t2 OTU
# node tree2n8 is associated with t5 OTU
# node tree2n9 is associated with t4 OTU
```

[Download tolweb.xml example] || [Download script]

Node meta information is also available:

```
from ete2 import Nexml

# Creates and empty NeXML project
p = Nexml()
# Fill it with the tolweb example
p.build_from_file("tolweb.xml")

# extract the first collection of trees
tree_collection = p.trees[0]
# and all the tree instances in it
trees = tree_collection.tree

# For each loaded tree, prints its structure and some of its
# meta-properties
for t in trees:
    print t
    print
    print "Leaf node meta information:\n"
    print
    for meta in  t.children[0].nexml_node.meta:
        print  meta.property, ":", (meta.content)


# Output
# ==========
#
# ---- /-node3(Eurysphindus)
#
# Leaf node meta information:
#
#
# dc:description :
# tbe:AUTHORITY : Leconte
# tbe:AUTHDATE : 1878
# tba:ANCESTORWITHPAGE : 117851
# tba:CHILDCOUNT : 0
# tba:COMBINATION_DATE : null
# tba:CONFIDENCE : 0
# tba:EXTINCT : 0
# tba:HASPAGE : 1
# tba:ID : 117855
# tba:INCOMPLETESUBGROUPS : 0
# tba:IS_NEW_COMBINATION : 0
# tba:ITALICIZENAME : 1
# tba:LEAF : 0
# tba:PHYLESIS : 0
# tba:SHOWAUTHORITY : 0
# tba:SHOWAUTHORITYCONTAINING : 1
```

[Download tolweb.xml example] || [Download script]

Creating Nexml project from scratch Nexml base class can also be used to create projects from scratch in a programmatic way. Using the collection of NeXML classes provided by the:mod:*ete2.nexml* module, you can populate an empty project and export it as XML.

```
import sys
# Note that we import the nexml module rather than the root Nexml
#  class.  This module contains a python object for each of the
#  nexml elements declared in its XML schema.
from ete2 import nexml

# Create an empty Nexml project
nexml_project = nexml.Nexml()
tree_collection = nexml.Trees()

# NexmlTree is a special PhyloTree instance that is prepared to be
# added to NeXML projects. So lets populate a random tree
nexml_tree = nexml.NexmlTree()
# Random tree with 10 leaves
nexml_tree.populate(10, random_branches=True)
```

```
# We add the tree to the collection
tree_collection.add_tree(nexml_tree)

# Create another tree from a newick string
nexml_tree2 = nexml.NexmlTree("((hello, nexml):1.51, project):0.6;")
tree_collection.add_tree(nexml_tree2)

# Tree can be handled as normal ETE objects
nexml_tree2.show()

# Add the collection of trees to the NexML project object
nexml_project.add_trees(tree_collection)

# Now we can export the project containing our two trees
nexml_project.export()
```

[Download script]

Writing NeXML objects Every NexML object has its own `export()` method. By calling it, you can obtain the XML representation of any instance contained in the Nexml project structure. Usually, all you will need is to export the whole project, but individual elements can be exported.

```
import sys
from ete2 import Nexml
# Create an empty Nexml project
nexml_project = Nexml()

# Upload content from file
nexml_project.build_from_file("nexml_example.xml")

# Extract first collection of trees
tree_collection =  nexml.get_trees()[0]

# And export it
tree_collection.export(output=sys.stdout, level=0)
```

NeXML tree manipulation and visualization

NeXML trees contain all ETE PhyloTree functionality: orthology prediction, topology manipulation and traversing methods, visualization, etc.

For instance, tree changes performed through the visualization GUI are kept in the NeXML format.

```
from ete2 import nexml
nexml_tree = nexml.NexMLTree("((hello, nexml):1.51, project):0.6;")
tree_collection.add_tree(nexml_tree)
nexml_tree.show()
```

New in version 2.1.

3.6.2 PhyloXML

PhyloXML (http://www.phyloxml.org/) is a novel standard used to encode phylogenetic information. In particular, phyloXML is designed to describe phylogenetic trees (or networks) and associated data, such as taxonomic information, gene names and identifiers, branch lengths, support values, and gene duplication and speciation events.

Loading PhyloXML projects from files

ETE provides full support for phyloXML projects through the Phyloxml object. Phylogenies are integrated as ETE's tree data structures as PhyloxmlTree instances, while the rest of features are represented as simple classes (ete2.phyloxml) providing basic reading and writing operations.

```
from ete2 import Phyloxml
project = Phyloxml()
project.build_from_file("apaf.xml")

# Each tree contains the same methods as a PhyloTree object
for tree in project.get_phylogeny():
    print tree
    # you can even use rendering options
    tree.show()
    # PhyloXML features are stored in the phyloxml_clade attribute
    for node in tree:
        print "Node name:", node.name
        for seq in node.phyloxml_clade.get_sequence():
            for domain in seq.domain_architecture.get_domain():
                domain_data = [domain.valueOf_, domain.get_from(), domain.get_to()]
                print "  Domain:", '\t'.join(map(str, domain_data))
```

[Download script] [Download example]

Each tree node contains two phyloxml elements, phyloxml_clade and phyloxml_phylogeny. The first attribute contains clade information referred to the node, while phyloxml_phylogeny contains general data about the subtree defined by each node. This way, you can split, or copy any part of a tree and it will be exported as a separate phyloxml phylogeny instance.

Note that node.dist, node.support and node.name features are linked to node.phyloxml_clade.branch_length, node.phyloxml_clade.confidence and node.phyloxml_clade.name, respectively.

Creating PhyloXML projects from scratch

In order to create new PhyloXML projects, a set of classes is available in the ete2.phyloxml module.

```
from ete2 import Phyloxml, phyloxml
import random
project = Phyloxml()

# Creates a random tree
phylo = phyloxml.PhyloxmlTree()
phylo.populate(5, random_branches=True)
phylo.phyloxml_phylogeny.set_name("test_tree")
# Add the tree to the phyloxml project
project.add_phylogeny(phylo)

print project.get_phylogeny()[0]

#          /-iajom
#     /---|
#    |     \-wiszh
#----|
#    |     /-xrygw
#     \---|
#          |     /-gjlwx
#           \---|
#                \-ijvnk

# Trees can be operated as normal ETE trees
phylo.show()


# Export the project as phyloXML format
project.export()

# <phy:Phyloxml xmlns:phy="http://www.phyloxml.org/1.10/phyloxml.xsd">
#     <phy:phylogeny>
#         <phy:name>test_tree</phy:name>
#         <phy:clade>
#             <phy:name>NoName</phy:name>
#             <phy:branch_length>0.000000e+00</phy:branch_length>
#             <phy:confidence type="branch_support">1.0</phy:confidence>
#             <phy:clade>
#                 <phy:name>NoName</phy:name>
#                 <phy:branch_length>1.665083e-01</phy:branch_length>
#                 <phy:confidence type="branch_support">0.938507980435</phy:confidence>
#                 <phy:clade>
#                     <phy:name>NoName</phy:name>
#                     <phy:branch_length>1.366655e-01</phy:branch_length>
#                     <phy:confidence type="branch_support">0.791888248212</phy:confiden
#                     <phy:clade>
#                         <phy:name>ojnfg</phy:name>
#                         <phy:branch_length>2.194209e-01</phy:branch_length>
#                         <phy:confidence type="branch_support">0.304705977822</phy:con
#                     </phy:clade>
#                     <phy:clade>
#                         <phy:name>qrfnz</phy:name>
#                         <phy:branch_length>5.235437e-02</phy:branch_length>
#                         <phy:confidence type="branch_support">0.508533765418</phy:con
#                     </phy:clade>
#                 </phy:clade>
#                 <phy:clade>
#                     <phy:name>shngq</phy:name>
#                     <phy:branch_length>9.740958e-01</phy:branch_length>
#                     <phy:confidence type="branch_support">0.642187390965</phy:confiden
#                 </phy:clade>
#             </phy:clade>
#             <phy:clade>
#                 <phy:name>NoName</phy:name>
#                 <phy:branch_length>3.806412e-01</phy:branch_length>
#                 <phy:confidence type="branch_support">0.383619811911</phy:confidence>
#                 <phy:clade>
#                     <phy:name>vfmnk</phy:name>
#                     <phy:branch_length>6.495163e-01</phy:branch_length>
#                     <phy:confidence type="branch_support">0.141298879514</phy:confiden
#                 </phy:clade>
#                 <phy:clade>
#                     <phy:name>btexi</phy:name>
#                     <phy:branch_length>5.704955e-01</phy:branch_length>
#                     <phy:confidence type="branch_support">0.951876078012</phy:confiden
#                 </phy:clade>
#             </phy:clade>
#         </phy:clade>
#     </phy:phylogeny>
# </phy:Phyloxml>
```

[Download script]

Note: NeXML and PhyloXML python parsers are possible thanks to Dave Kulhman and his work on the generateDS.py application. Thanks Dave! ;-)

New in version 2.1.

3.7 Overview

Starting at version 2.1, ETE provides a module to interactively display trees within web pages. This task is not straightforward, but ETE tries to simplify it by providing a basic `WebTreeApplication` class that can be imported in your python web applications.

`WebTreeApplication` implements a transparent connector between ETE's functionality and web application. For this, a pre-built WSGI application is provided.

Through this application, you will be able to create custom web implementations to visualize and manipulate trees interactively. Some examples can be found at the PhylomeDB tree browser or in the ETE's online treeviewer.

3.7.1 Installing a X server

All modern linux desktop installations include a graphical interface (called X server). However web servers (in which the ETE plugin is expected to run) may not count with a X server.

3.7.2 Servers

In order to render tree images with ETE, you will need to install, at least, a basic X server. Note that the X server does not require a desktop interface, such as Gnome or KDE.

In Ubuntu, for instance, a basic X server called xdm can be installed as follows:

```
apt-get install xserver-xorg xdm xfonts-base xfonts-100dpi
xfonts-75dpi
```

Once the X server is installed, you will need to configure it to accept connections from the web-server.

In our example, edit the `/etc/X11/xdm/xdm-config` file and set following values:

```
DisplayManager*authorize:       false
!
DisplayManager*authComplain:    false
```

Do not forget to restart your xdm server.

```
/etc/init.d/xdm restart
```

3.7.3 Desktops

If you plan to use web tree application in a linux desktop computer, then the X server is already installed. You will only need to give permissions to the web-server (i.e. apache) to connect your display. Usually, as simple as running the following command in a terminal:

```
xhost +
```

3.7.4 Configuring the web sever

You will need to add support for WSGI application to your web server. In the following steps, an Apache2 web server will be assumed.

- Install and activate the `modwsgi` module in Apache.
- Configure your site to support WSGI.

Configuration will depend a lot on your specific system, but this is an example configuration file for the default site of your Apache server (usually at `/ete/apache2/sites-available/default`):

```
<VirtualHost *:80>
 ServerAdmin webmaster@localhost

 DocumentRoot /var/www
 <Directory />
        Options +FollowSymLinks
        AllowOverride None
 </Directory>

 ErrorLog /var/log/apache2/error.log

 # Possible values include: debug, info, notice, warn, error, crit,
 # alert, emerg.
 LogLevel warn

 CustomLog /var/log/apache2/access.log combined

 # ########################### #
 # WSGI SPECIFIC CONFIG        #

 WSGIDaemonProcess eteApp user=www-data group=www-data processes=1 threads=1
 WSGIProcessGroup eteApp
 WSGIApplicationGroup %{GLOBAL}

 <Directory /var/www/webplugin/>
        Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
        SetHandler wsgi-script
        Order allow,deny
        Allow from all
        AddHandler wsgi-script .py
 </Directory>
 # END OF WSGI SPECIFIC CONFIG #
 # ########################### #

</VirtualHost>
```

Note: */var/www/webplugin/wsgi/* is the folder in which python web application will be located. Make sure that the apache WSGI config enables this folder to run wsgi-scripts.

Warning: Important notes:
/var/www/webplugin/ is assumed to be the directory in which your application will run.
/var/www/webplugin/tmp/ should be writable by the web-server (i.e. chmod 777)

3.7.5 Implementation of WebTreeApplications

ETE's `WebTreeApplication` uses WSGI in the backend, and a several javascript files in the frontend. Basic files are included as an example in the ETE installation package `examples/webplugin`. The whole example folder is necessary, and it contains a commented copy of a web-tree implementation `examples/webplugin/wsgi/webplugin_example.py`.

CHAPTER 4

ETE's Reference Guide

Current modules:

4.1 Master Tree class

class TreeNode (*newick=None*, *format=0*)

TreeNode (Tree) class is used to store a tree structure. A tree consists of a collection of TreeNode instances connected in a hierarchical way. Trees can be loaded from the New Hampshire Newick format (newick).

Parameters

- **newick** – Path to the file containing the tree or, alternatively, the text string containing the same information.
- **format** (*0*) – subnewick format

FORMAT	DESCRIPTION
0	flexible with support values
1	flexible with internal node names
2	all branches + leaf names + internal supports
3	all branches + all names
4	leaf branches + leaf names
5	internal and leaf branches + leaf names
6	internal branches + leaf names
7	leaf branches + all names
8	all names
9	leaf names
100	topology only

Returns a tree node object which represents the base of the tree.

** Examples: **

```
t1 = Tree() # creates an empty tree
t2 = Tree('(A:1,(B:1,(C:1,D:1):0.5):0.5);')
t3 = Tree('/home/user/myNewickFile.txt')
```

add_child (*child=None*, *name=None*, *dist=None*, *support=None*)

Adds a new child to this node. If child node is not suplied as an argument, a new node instance will be created.

Parameters

- **child** (*None*) – the node instance to be added as a child.
- **name** (*None*) – the name that will be given to the child.
- **dist** (*None*) – the distance from the node to the child.
- **support'** (*None*) – the support value of child partition.

Returns The child node instance

add_face (*face*, *column*, *position='branch-right'*)

Add a fixed face to the node. This type of faces will be always attached to nodes, independently of the layout function.

Parameters

- **face** – a Face or inherited instance
- **column** – An integer number starting from 0
- **position** (*"branch-right"*) – Posible values are: "branch-right", "branch-top", "branch-bottom", "float", "aligned"

add_feature (*pr_name*, *pr_value*)

Add or update a node's feature.

add_features (***features*)

Add or update several features.

add_sister (*sister=None*, *name=None*, *dist=None*)

Adds a sister to this node. If sister node is not supplied as an argument, a new TreeNode instance will be created and returned.

children

A list of children nodes

convert_to_ultrametric (*tree_length*, *strategy='balanced'*)

Converts a tree to ultrametric topology (all leaves must have the same distance to root). Note that, for visual inspection of ultrametric trees, node.img_style["size"] should be set to 0.

copy ()

Returns an exact and complete copy of current node.

del_feature (*pr_name*)

Permanently deletes a node's feature.

delete (*prevent_nondicotomic=True*)

Deletes node from the tree structure. Notice that this method makes 'disapear' the node from the tree structure. This means that children from the deleted node are transferred to the next available parent.

Example:

```
       / C
root-|
      |        / B
      \--- H |
              \ A

> root.delete(H) will produce this structure:

      / C
     |
root-|--B
     |
      \ A
```

describe()

Prints general information about this node and its connections.

detach()

Detachs this node (and all its descendants) from its parent and returns the referent to itself.

Detached node conserves all its structure of descendants, and can be attached to another node through the 'add_child' function. This mechanism can be seen as a cut and paste.

dist

Branch length distance to parent node. Default = 0.0

get_ascii(*show_internal=True*, *compact=False*)

Returns a string containing an ascii drawing of the tree.

Parameters

- **show_internal** – includes internal edge names.
- **compact** – use exactly one line per tip.

get_children()

Returns an independent list of node's children.

get_closest_leaf(*topology_only=False*)

Returns node's closest descendant leaf and the distance to it.

Parameters **topology_only** (*False*) – If set to True, distance between nodes will be referred to the number of nodes between them. In other words, topological distance will be used instead of branch length distances.

Returns A tuple containing the closest leaf referred to the current node and the distance to it.

get_common_ancestor(**target_nodes*, ***kargs*)

Returns the first common ancestor between this node and a given list of 'target_nodes'.

Examples:

```
t = tree.Tree("(((A:0.1, B:0.01):0.001, C:0.0001):1.0[&&NHX:name=common], (D:0.
A = t.get_descendants_by_name("A")[0]
C = t.get_descendants_by_name("C")[0]
common =  A.get_common_ancestor(C)
print common.name
```

get_common_ancestor_OLD(**target_nodes*)

Returns the first common ancestor between this node and a given list of 'target_nodes'.

Examples:

```
t = tree.Tree("(((A:0.1, B:0.01):0.001, C:0.0001):1.0[&&NHX:name=common], (D:0.
A = t.get_descendants_by_name("A")[0]
C = t.get_descendants_by_name("C")[0]
common =  A.get_common_ancestor(C)
print common.name
```

get_descendants(*strategy='levelorder'*, *is_leaf_fn=None*)
Returns a list of all (leaves and internal) descendant nodes.

Parameters **is_leaf_fn** (*None*) – See `TreeNode.traverse()` for documentation.

get_distance(*target*, *target2=None*, *topology_only=False*)
Returns the distance between two nodes. If only one target is specified, it returns the distance bewtween the target and the current node.

Parameters

- **target** – a node within the same tree structure.
- **target2** – a node within the same tree structure. If not specified, current node is used as target2.
- **topology_only** (*False*) – If set to True, distance will refer to the number of nodes between target and target2.

Returns branch length distance between target and target2. If topology_only flag is True, returns the number of nodes between target and target2.

get_farthest_leaf(*topology_only=False*)
Returns node's farthest descendant node (which is always a leaf), and the distance to it.

Parameters **topology_only** (*False*) – If set to True, distance between nodes will be referred to the number of nodes between them. In other words, topological distance will be used instead of branch length distances.

Returns A tuple containing the farthest leaf referred to the current node and the distance to it.

get_farthest_node(*topology_only=False*)
Returns the node's farthest descendant or ancestor node, and the distance to it.

Parameters **topology_only** (*False*) – If set to True, distance between nodes will be referred to the number of nodes between them. In other words, topological distance will be used instead of branch length distances.

Returns A tuple containing the farthest node referred to the current node and the distance to it.

get_leaf_names(*is_leaf_fn=None*)
Returns the list of terminal node names under the current node.

Parameters **is_leaf_fn** (*None*) – See `TreeNode.traverse()` for documentation.

get_leaves(*is_leaf_fn=None*)
Returns the list of terminal nodes (leaves) under this node.

Parameters **is_leaf_fn** (*None*) – See `TreeNode.traverse()` for documentation.

get_leaves_by_name (*name*)
Returns a list of leaf nodes matching a given name.

get_midpoint_outgroup ()
Returns the node that divides the current tree into two distance-balanced partitions.

get_partitions ()
It returns the set of all possible partitions under a node. Note that current implementation is quite inefficient when used in very large trees.

t = Tree("((a, b), e);") partitions = t.get_partitions()

# Will return: # a,b,e # a,e # b,e # a,b # e # b # a

get_sisters ()
Returns an indepent list of sister nodes.

get_tree_root ()
Returns the absolute root node of current tree structure.

is_leaf ()
Return True if current node is a leaf.

is_root ()
Returns True if current node has no parent

iter_descendants (*strategy='levelorder'*, *is_leaf_fn=None*)
Returns an iterator over all descendant nodes.

Parameters is_leaf_fn (*None*) – See `TreeNode.traverse()` for documentation.

iter_leaf_names (*is_leaf_fn=None*)
Returns an iterator over the leaf names under this node.

Parameters is_leaf_fn (*None*) – See `TreeNode.traverse()` for documentation.

iter_leaves (*is_leaf_fn=None*)
Returns an iterator over the leaves under this node.

Parameters is_leaf_fn (*None*) – See `TreeNode.traverse()` for documentation.

iter_search_nodes (***conditions*)
Search nodes in an interative way. Matches are being yield as they are being found. This avoids to scan the full tree topology before returning the first matches. Useful when dealing with huge trees.

ladderize (*direction=0*)
Sort the branches of a given tree (swapping children nodes) according to the size of each partition.

```
t =  Tree("(f,((d, ((a,b),c)),e));")

print t

#
#      /-f
#     |
#     |          /-d
# ----|         |
#     |     /---|          /-a
#     |    |    |     /---|
#     |    |     \---|     \-b
#      \---|         |
#          |          \-c
#          |
#           \-e

t.ladderize()
print t

#      /-f
# ----|
#     |     /-e
#      \---|
#          |     /-d
#           \---|
#               |     /-c
#                \---|
#                    |     /-a
#                     \---|
#                          \-b
```

populate(*size*, *names_library=None*, *reuse_names=False*, *random_branches=False*, *branch_range=(0, 1)*, *support_range=(0, 1)*)
Generates a random topology by populating current node.

Parameters

- **names_library** (*None*) – If provided, names library (list, set, dict, etc.) will be used to name nodes.
- **reuse_names** (*False*) – If True, node names will not be necessarily unique, which makes the process a bit more efficient.
- **random_branches** (*False*) – If True, branch distances and support values will be randomized.
- **branch_range** (*(0,1)*) – If random_branches is True, this

range of values will be used to generate random distances.

Parameters **support_range** (*(0,1)*) – If random_branches is True,

this range of values will be used to generate random branch support values.

prune(*nodes*)
Prunes the topology of a node in order to conserve only a selected list of leaf or internal nodes. The minimum number of internal nodes (the deepest as possible) are kept to conserve the topological relationship among the provided list of nodes.

Variables **nodes** – a list of node names or node objects that must be kept

Examples:

```
t = Tree("(((A:0.1, B:0.01):0.001, C:0.0001):1.0[&&NHX:name=I], (D:0.00001):0.0
node_C = t.search_nodes(name="C")[0]
```

```
t.prune(["A","D", node_C])
print t
```

remove_child(*child*)

Removes a child from this node (parent and child nodes still exit but are no longer connected).

remove_sister(*sister=None*)

Removes a node's sister node. It has the same effect as **'TreeNode.up.remove_child(sister)'**

If a sister node is not supplied, the first sister will be deleted and returned.

Parameters **sister** – A node instance

Returns The node removed

render(*file_name*, *layout=None*, *w=None*, *h=None*, *tree_style=None*, *units='px'*, *dpi=300*)

Renders the node structure as an image.

Variables

- **file_name** – path to the output image file. valid extensions are .SVG, .PDF, .PNG
- **layout** – a layout function or a valid layout function name
- **tree_style** – a *TreeStyle* instance containing the image properties
- **units** (*px*) – "px": pixels, "mm": millimeters, "in": inches
- **h** (*None*) – height of the image in `units`
- **w** (*None*) – weight of the image in `units`
- **dpi** (*300*) – dots per inches.

search_nodes(***conditions*)

Returns the list of nodes matching a given set of conditions.

Example:

```
tree.search_nodes(dist=0.0, name="human")
```

set_outgroup(*outgroup*)

Sets a descendant node as the outgroup of a tree. This function can be used to root a tree or even an internal node.

Parameters **outgroup** – a node instance within the same tree structure that will be used as a basal node.

set_style(*node_style*)

Set 'node_style' as the fixed style for the current node.

show(*layout=None*, *tree_style=None*)

Starts an interative session to visualize current node structure using provided layout and TreeStyle.

sort_descendants()

This function sort the branches of a given tree by considerening node names. After the tree is sorted, nodes are labeled using ascendent numbers. This can be used to ensure that nodes in a tree with the same node names are always labeled in the same way. Note that if duplicated names are present, extra criteria should be added to sort nodes. unique id is stored in _nid

support
Branch support for current node

swap_children()
Swaps current children order.

traverse(*strategy='levelorder'*, *is_leaf_fn=None*)
Returns an iterator to traverse the tree structure under this node.

Parameters

- **strategy** (*"levelorder"*) – set the way in which tree will be traversed. Possible values are: "preorder" (first parent and then children) 'postorder' (first children and the parent) and "levelorder" (nodes are visited in order from root to leaves)
- **is_leaf_fn** (*None*) – If supplied, `is_leaf_fn` function will be used to interrogate nodes about if they are terminal or internal. `is_leaf_fn` function should receive a node instance as first argument and return True or False. Use this argument to traverse a tree dynamically collapsing internal nodes.

unroot()
Unroots current node. This function is expected to be used on the absolute tree root node, but it can be also be applied to any other internal node. It will convert a split into a multifurcation.

up
Pointer to parent node

write(*features=None*, *outfile=None*, *format=0*)
Returns the newick representation of current node. Several arguments control the way in which extra data is shown for every node:

Parameters

- **features** – a list of feature names to be exported using the Extended Newick Format (i.e. features=["name", "dist"]). Use an empty list to export all available features in each node (features=[])
- **outfile** – writes the output to a given file
- **format** – defines the newick standard used to encode the tree. See tutorial for details.

Example:

```
t.get_newick(features=["species","name"], format=1)
```

Tree
alias of `TreeNode`

4.2 Treeview module

Contents

4.2.1 TreeStyle

class `TreeStyle`

New in version 2.1. Contains all the general image properties used to render a tree

– About tree design –

Variables **layout_fn** (*None*) – Layout function used to dynamically control the aspect of nodes. Valid values are: None or a pointer to a method, function, etc.

– About tree shape –

Variables

- **mode** (*"r"*) – Valid modes are 'c'(ircular) or 'r'(ectangular).
- **orientation** (*0*) – If 0, tree is drawn from left-to-right. If 1, tree is drawn from right-to-left. This property only makes sense when "r" mode is used.
- **rotation** (*0*) – Tree figure will be rotate X degrees (clock-wise rotation).
- **min_leaf_separation** (*1*) – Min separation, in pixels, between two adjacent branches
- **branch_vertical_margin** (*0*) – Leaf branch separation margin, in pixels. This will add a separation of X pixels between adjacent leaf branches. In practice, increasing this value work as increasing Y axis scale.
- **arc_start** (*0*) – When circular trees are drawn, this defines the starting angle (in degrees) from which leaves are distributed (clock-wise) around the total arc span (0 = 3 o'clock).
- **arc_span** (*360*) – Total arc used to draw circular trees (in degrees).
- **margin_left** (*0*) – Left tree image margin, in pixels.
- **margin_right** (*0*) – Right tree image margin, in pixels.
- **margin_top** (*0*) – Top tree image margin, in pixels.
- **margin_bottom** (*0*) – Bottom tree image margin, in pixels.

– About Tree branches –

Variables

- **scale** (*None*) – Scale used to draw branch lengths. If None, it will be automatically calculated.
- **optimal_scale_level** (*"mid"*) – Two levels of automatic branch scale detection are available: `"mid"` and `"full"`. In `full` mode, branch scale will

me adjusted to fully avoid dotted lines in the tree image. In other words, scale will be increased until the extra space necessary to allocated all branch-top/bottom faces and branch-right faces (in circular mode) is covered by real branches. Note, however, that the optimal scale in trees with very unbalanced branch lengths might be huge. If `"mid"` mode is selected (as it is by default), optimal scale will only satisfy the space necessary to allocate branch-right faces in circular trees. Some dotted lines (artificial branch offsets) will still appear when branch-top/bottom faces are larger than branch length. Note that both options apply only when `scale` is set to None (automatic).

- **root_opening_factor** (*0.25*) – (from 0 to 1). It defines how much the center of a circular tree could be opened when adjusting optimal scale, referred to the total tree length. By default (0.25), a blank space up to 4 times smaller than the tree width could be used to calculate the optimal tree scale. A 0 value would mean that root node should always be tightly adjusted to the center of the tree.
- **complete_branch_lines_when_necessary** (*True*) – True or False. Draws an extra line (dotted by default) to complete branch lengths when the space to cover is larger than the branch itself.
- **extra_branch_line_type** (*2*) – 0=solid, 1=dashed, 2=dotted
- **extra_branch_line_color”** (*“gray”*) – RGB code or name in `SVG_COLORS`
- **force_topology** (*False*) – Convert tree branches to a fixed length, thus allowing to observe the topology of tight nodes
- **draw_guiding_lines** (*True*) – Draw guidelines from leaf nodes to aligned faces
- **guiding_lines_type** (*2*) – 0=solid, 1=dashed, 2=dotted.
- **guiding_lines_color** (*“gray”*) – RGB code or name in `SVG_COLORS`

– About node faces –

Variables

- **allow_face_overlap** (*False*) – If True, node faces are not taken into account to scale circular tree images, just like many other visualization programs. Overlapping among branch elements (such as node labels) will be therefore ignored, and tree size will be a lot smaller. Note that in most cases, manual setting of tree scale will be also necessary.
- **draw_aligned_faces_as_table** (*True*) – Aligned faces will be drawn as a table, considering all columns in all node faces.
- **children_faces_on_top** (*True*) – When floating faces from different nodes overlap, children faces are drawn on top of parent faces. This can be reversed by setting this attribute to false.

– Addons –

Variables

- **show_border** (*False*) – Draw a border around the whole tree
- **show_scale** (*True*) – Include the scale legend in the tree image

- **show_leaf_name** (*False*) – Automatically adds a text Face to leaf nodes showing their names
- **show_branch_length** (*False*) – Automatically adds branch length information on top of branches
- **show_branch_support** (*False*) – Automatically adds branch support text in the bottom of tree branches

– Tree surroundings –

The following options are actually Face containers, so graphical elements can be added just as it is done with nodes. In example, to add tree legend:

```
TreeStyle.legend.add_face(CircleFace(10, "red"), column=0)
TreeStyle.legend.add_face(TextFace("0.5 support"), column=1)
```

Variables

- **aligned_header** – a `FaceContainer` aligned to the end of the tree and placed at the top part.
- **aligned_foot** – a `FaceContainer` aligned to the end of the tree and placed at the bottom part.
- **legend** – a `FaceContainer` with an arbitrary number of faces representing the legend of the figure.
- **legend_position=4** (*4*) – TopLeft corner if 1, TopRight if 2, BottomLeft if 3, BottomRight if 4
- **title** – A text string that will be draw as the Tree title

class **`FaceContainer`**

New in version 2.1. Use this object to create a grid of faces. You can add faces to different columns.

`add_face` (*face*, *column*)

add the face **face** to the specified **column**

4.2.2 NodeStyle

class **`NodeStyle`** (**args*, ***kargs*)

New in version 2.1. A dictionary with all valid node graphical attributes.

Parameters

- **fgcolor** (*#0030c1*) – RGB code or name in `SVG_COLORS`
- **bgcolor** (*#FFFFFF*) – RGB code or name in `SVG_COLORS`
- **node_bgcolor** (*#FFFFFF*) – RGB code or name in `SVG_COLORS`
- **partition_bgcolor** (*#FFFFFF*) – RGB code or name in `SVG_COLORS`
- **faces_bgcolor** (*#FFFFFF*) – RGB code or name in `SVG_COLORS`
- **vt_line_color** (*#000000*) – RGB code or name in `SVG_COLORS`
- **hz_line_color** (*#000000*) – RGB code or name in `SVG_COLORS`

- **hz_line_type** (*0*) – integer number
- **vt_line_type** (*0*) – integer number
- **size** (*3*) – integer number
- **shape** (*"circle"*) – "circle", "square" or "sphere"
- **draw_descendants** (*True*) – Mark an internal node as a leaf.
- **hz_line_width** (*0*) – integer number representing the width of the line in pixels. A line width of zero indicates a cosmetic pen. This means that the pen width is always drawn one pixel wide, independent of the transformation set on the painter.
- **vt_line_width** (*0*) – integer number representing the width of the line in pixels. A line width of zero indicates a cosmetic pen. This means that the pen width is always drawn one pixel wide, independent of the transformation set on the painter.

4.2.3 Faces

`add_face_to_node` (*face*, *node*, *column*, *aligned=False*, *position='branch-right'*)

Adds a Face to a given node.

Parameters **face** – A `Face` instance

Parameters

- **node** – a tree node instance (`Tree`, `PhyloTree`, etc.)
- **column** – An integer number starting from 0
- **position** (*"branch-right"*) – Possible values are "branch-right", "branch-top", "branch-bottom", "float", "aligned"

class **`Face`**

Standard definition of a Face node object.

This class is not functional and it should only be used to create other face objects. By inheriting this class, you set all the essential attributes, however the update_pixmap() function is required to be reimplemented for convenience.

User adjustable properties:

Parameters

- **margin_left** (*0*) – in pixels
- **margin_right** (*0*) – in pixels
- **margin_top** (*0*) – in pixels
- **margin_bottom** (*0*) – in pixels
- **opacity** (*1.0*) – a float number in the (0,1) range
- **rotable** (*True*) – If True, face will be rotated when necessary (i.e. when circular mode is enabled and face occupies an inverted position.)
- **hz_align** (*0*) – 0 left, 1 center, 2 right

- **vt_align** (*1*) – 0 top, 1 center, 2 bottom
- **background** – background of face plus its margins
- **inner_background** – background of the face
- **border** (*None*) – Border around face margins. Integer number representing the width of the face border line in pixels. A line width of zero indicates a cosmetic pen. This means that the pen width is always drawn one pixel wide, independent of the transformation set on the painter. A "None" value means invisible border.
- **inner_border** (*None*) – Border around face . Integer number representing the width of the face border line in pixels. A line width of zero indicates a cosmetic pen. This means that the pen width is always drawn one pixel wide, independent of the transformation set on the painter. A "None" value means invisible border.

class `TextFace` (*text*, *ftype='Verdana'*, *fsize=10*, *fgcolor='#000000'*, *penwidth=0*, *fstyle='normal'*)

Static text Face object

Parameters

- **text** – Text to be drawn
- **ftype** – Font type, e.g. Arial, Verdana, Courier
- **fsize** – Font size, e.g. 10,12,6, (default=10)
- **fgcolor** – Foreground font color. RGB code or name in `SVG_COLORS`
- **penwidth** – Penwdith used to draw the text.
- **fstyle** – "normal" or "italic"

class `AttrFace` (*attr*, *ftype='Verdana'*, *fsize=10*, *fgcolor='#000000'*, *penwidth=0*, *text_prefix=' '*, *text_suffix=' '*, *formatter=None*, *fstyle='normal'*)

Dynamic text Face. Text rendered is taken from the value of a given node attribute.

Parameters

- **attr** – Node's attribute that will be drawn as text
- **ftype** – Font type, e.g. Arial, Verdana, Courier, (default="Verdana")
- **fsize** – Font size, e.g. 10,12,6, (default=10)
- **fgcolor** – Foreground font color. RGB code or name in `SVG_COLORS`
- **penwidth** – Penwdith used to draw the text. (default is 0)
- **text_prefix** – text_rendered before attribute value
- **text_suffix** – text_rendered after attribute value
- **formatter** – a text string defining a python formater to process the attribute value before renderer. e.g. "%0.2f"
- **fstyle** – "normal" or "italic"

class `ImgFace` (*img_file*)

Creates a new image face object.

Parameters img_file – Image file in png,jpg,bmp format

class CircleFace (*radius*, *color*, *style='circle'*)

New in version 2.1. Creates a Circle or Sphere Face.

Arguments radius integer number defining the radius of the face

Arguments color Color used to fill the circle. RGB code or name in SVG_COLORS

Arguments "circle" style Valid values are "circle" or "sphere"

class SequenceFace (*seq*, *seqtype*, *fsize=10*, *aafg=None*, *aabg=None*, *ntfg=None*, *ntbg=None*)

Creates a new molecular sequence face object.

Parameters

- **seq** – Sequence string to be drawn
- **seqtype** – Type of sequence: "nt" or "aa"
- **fsize** – Font size, (default=10)

You can set custom colors for aminoacids or nucleotides:

Parameters

- **aafg** – a dictionary in which keys are aa codes and values are foreground RGB colors
- **aabg** – a dictionary in which keys are aa codes and values are background RGB colors
- **ntfg** – a dictionary in which keys are nucleotides codes and values are foreground RGB colors
- **ntbg** – a dictionary in which keys are nucleotides codes and values are background RGB colors

class ProfileFace (*max_v*, *min_v*, *center_v*, *width=200*, *height=40*, *style='lines'*, *colorscheme=2*)

A profile Face for ClusterNodes

Parameters

- **max_v** – maximum value used to build the build the plot scale.
- **max_v** – minimum value used to build the build the plot scale.
- **center_v** – Center value used to scale plot and heatmap.
- **width** (*200*) – Plot width in pixels.
- **height** (*40*) – Plot width in pixels.
- **style** (*lines*) – Plot style: "lines", "bars", "cbars" or "heatmap".
- **colorscheme** (*2*) – colors used to create the gradient from min values to max values. 0=green & blue; 1=green & red; 2=red & blue. In all three cases, missing values are rendered in black and transition color (values=center) is white.

class TreeFace (*tree*, *tree_style*)

New in version 2.1. Creates a Face containing a Tree object. Yes, a tree within a tree :)

Parameters

- **tree** – An ETE Tree instance (Tree, PhyloTree, etc...)
- **tree_style** – A TreeStyle instance defining how tree show be drawn

class `StaticItemFace` (*item*)

New in version 2.1. Creates a face based on an external QtGraphicsItem object. QGraphicsItem object is expected to be independent from tree node properties, so its content is assumed to be static (drawn only once, no updates when tree changes).

Arguments item an object based on QGraphicsItem

class `DynamicItemFace` (*constructor*, **args*, ***kargs*)

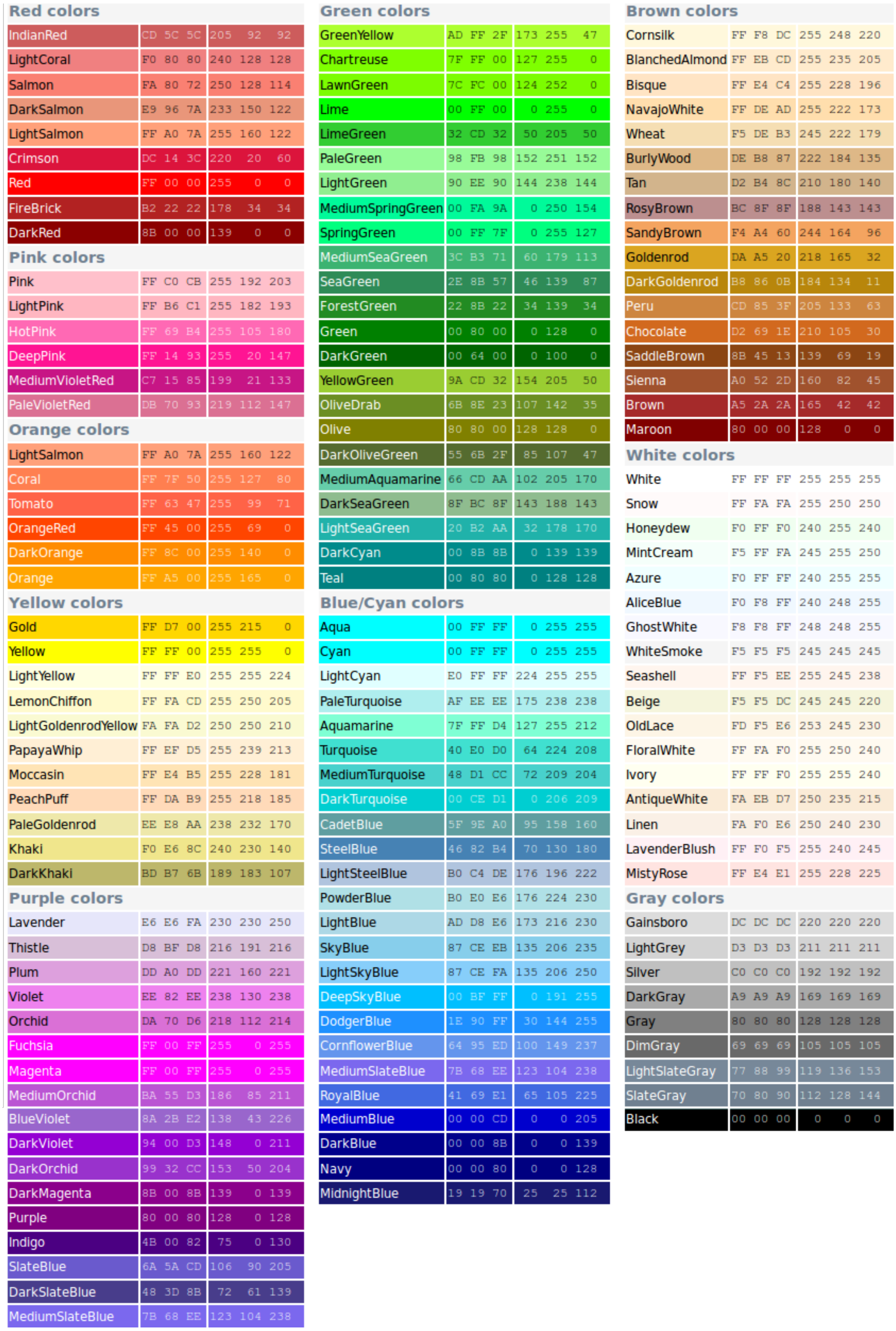
New in version 2.1. Creates a face based on an external QGraphicsItem object whose content depends on the node that is linked to.

Arguments constructor A pointer to a method (function or class constructor) returning a QGraphicsItem based object. “constructor” method is expected to receive a node instance as the first argument. The rest of arguments passed to ItemFace are optional and will passed also to the constructor function.

4.2.4 Color names

`SVG_COLORS`

Apart from RGB color codes, the following SVG color names are supported:



Red colors						
IndianRed	CD	5C	5C	205	92	92
LightCoral	F0	80	80	240	128	128
Salmon	FA	80	72	250	128	114
DarkSalmon	E9	96	7A	233	150	122
LightSalmon	FF	A0	7A	255	160	122
Crimson	DC	14	3C	220	20	60
Red	FF	00	00	255	0	0
FireBrick	B2	22	22	178	34	34
DarkRed	8B	00	00	139	0	0

Pink colors						
Pink	FF	C0	CB	255	192	203
LightPink	FF	B6	C1	255	182	193
HotPink	FF	69	B4	255	105	180
DeepPink	FF	14	93	255	20	147
MediumVioletRed	C7	15	85	199	21	133
PaleVioletRed	DB	70	93	219	112	147

Orange colors						
LightSalmon	FF	A0	7A	255	160	122
Coral	FF	7F	50	255	127	80
Tomato	FF	63	47	255	99	71
OrangeRed	FF	45	00	255	69	0
DarkOrange	FF	8C	00	255	140	0
Orange	FF	A5	00	255	165	0

Yellow colors						
Gold	FF	D7	00	255	215	0
Yellow	FF	FF	00	255	255	0
LightYellow	FF	FF	E0	255	255	224
LemonChiffon	FF	FA	CD	255	250	205
LightGoldenrodYellow	FA	FA	D2	250	250	210
PapayaWhip	FF	EF	D5	255	239	213
Moccasin	FF	E4	B5	255	228	181
PeachPuff	FF	DA	B9	255	218	185
PaleGoldenrod	EE	E8	AA	238	232	170
Khaki	F0	E6	8C	240	230	140
DarkKhaki	BD	B7	6B	189	183	107

Purple colors						
Lavender	E6	E6	FA	230	230	250
Thistle	D8	BF	D8	216	191	216
Plum	DD	A0	DD	221	160	221
Violet	EE	82	EE	238	130	238
Orchid	DA	70	D6	218	112	214
Fuchsia	FF	00	FF	255	0	255
Magenta	FF	00	FF	255	0	255
MediumOrchid	BA	55	D3	186	85	211
BlueViolet	8A	2B	E2	138	43	226
DarkViolet	94	00	D3	148	0	211
DarkOrchid	99	32	CC	153	50	204
DarkMagenta	8B	00	8B	139	0	139
Purple	80	00	80	128	0	128
Indigo	4B	00	82	75	0	130
SlateBlue	6A	5A	CD	106	90	205
DarkSlateBlue	48	3D	8B	72	61	139
MediumSlateBlue	7B	68	EE	123	104	238

Green colors						
GreenYellow	AD	FF	2F	173	255	47
Chartreuse	7F	FF	00	127	255	0
LawnGreen	7C	FC	00	124	252	0
Lime	00	FF	00	0	255	0
LimeGreen	32	CD	32	50	205	50
PaleGreen	98	FB	98	152	251	152
LightGreen	90	EE	90	144	238	144
MediumSpringGreen	00	FA	9A	0	250	154
SpringGreen	00	FF	7F	0	255	127
MediumSeaGreen	3C	B3	71	60	179	113
SeaGreen	2E	8B	57	46	139	87
ForestGreen	22	8B	22	34	139	34
Green	00	80	00	0	128	0
DarkGreen	00	64	00	0	100	0
YellowGreen	9A	CD	32	154	205	50
OliveDrab	6B	8E	23	107	142	35
Olive	80	80	00	128	128	0
DarkOliveGreen	55	6B	2F	85	107	47
MediumAquamarine	66	CD	AA	102	205	170
DarkSeaGreen	8F	BC	8F	143	188	143
LightSeaGreen	20	B2	AA	32	178	170
DarkCyan	00	8B	8B	0	139	139
Teal	00	80	80	0	128	128

Blue/Cyan colors						
Aqua	00	FF	FF	0	255	255
Cyan	00	FF	FF	0	255	255
LightCyan	E0	FF	FF	224	255	255
PaleTurquoise	AF	EE	EE	175	238	238
Aquamarine	7F	FF	D4	127	255	212
Turquoise	40	E0	D0	64	224	208
MediumTurquoise	48	D1	CC	72	209	204
DarkTurquoise	00	CE	D1	0	206	209
CadetBlue	5F	9E	A0	95	158	160
SteelBlue	46	82	B4	70	130	180
LightSteelBlue	B0	C4	DE	176	196	222
PowderBlue	B0	E0	E6	176	224	230
LightBlue	AD	D8	E6	173	216	230
SkyBlue	87	CE	EB	135	206	235
LightSkyBlue	87	CE	FA	135	206	250
DeepSkyBlue	00	BF	FF	0	191	255
DodgerBlue	1E	90	FF	30	144	255
CornflowerBlue	64	95	ED	100	149	237
MediumSlateBlue	7B	68	EE	123	104	238
RoyalBlue	41	69	E1	65	105	225
MediumBlue	00	00	CD	0	0	205
DarkBlue	00	00	8B	0	0	139
Navy	00	00	80	0	0	128
MidnightBlue	19	19	70	25	25	112

Brown colors						
Cornsilk	FF	F8	DC	255	248	220
BlanchedAlmond	FF	EB	CD	255	235	205
Bisque	FF	E4	C4	255	228	196
NavajoWhite	FF	DE	AD	255	222	173
Wheat	F5	DE	B3	245	222	179
BurlyWood	DE	B8	87	222	184	135
Tan	D2	B4	8C	210	180	140
RosyBrown	BC	8F	8F	188	143	143
SandyBrown	F4	A4	60	244	164	96
Goldenrod	DA	A5	20	218	165	32
DarkGoldenrod	B8	86	0B	184	134	11
Peru	CD	85	3F	205	133	63
Chocolate	D2	69	1E	210	105	30
SaddleBrown	8B	45	13	139	69	19
Sienna	A0	52	2D	160	82	45
Brown	A5	2A	2A	165	42	42
Maroon	80	00	00	128	0	0

White colors						
White	FF	FF	FF	255	255	255
Snow	FF	FA	FA	255	250	250
Honeydew	F0	FF	F0	240	255	240
MintCream	F5	FF	FA	245	255	250
Azure	F0	FF	FF	240	255	255
AliceBlue	F0	F8	FF	240	248	255
GhostWhite	F8	F8	FF	248	248	255
WhiteSmoke	F5	F5	F5	245	245	245
Seashell	FF	F5	EE	255	245	238
Beige	F5	F5	DC	245	245	220
OldLace	FD	F5	E6	253	245	230
FloralWhite	FF	FA	F0	255	250	240
Ivory	FF	FF	F0	255	255	240
AntiqueWhite	FA	EB	D7	250	235	215
Linen	FA	F0	E6	250	240	230
LavenderBlush	FF	F0	F5	255	240	245
MistyRose	FF	E4	E1	255	228	225

Gray colors						
Gainsboro	DC	DC	DC	220	220	220
LightGrey	D3	D3	D3	211	211	211
Silver	C0	C0	C0	192	192	192
DarkGray	A9	A9	A9	169	169	169
Gray	80	80	80	128	128	128
DimGray	69	69	69	105	105	105
LightSlateGray	77	88	99	119	136	153
SlateGray	70	80	90	112	128	144
Black	00	00	00	0	0	0

4.3 PhyloTree class

class **PhyloNode** (*newick=None*, *alignment=None*, *alg_format='fasta'*, *sp_naming_function=<function _parse_species at 0x280d410>*, *format=0*)
Bases: `ete2.coretype.tree.TreeNode`

Extends the standard `TreeNode` instance. It adds specific attributes and methods to work with phylogentic trees.

Parameters

- **newick** – Path to the file containing the tree or, alternatively, the text string containing the same information.
- **alignment** – file containing a multiple sequence alignment.
- **alg_format** – "fasta", "phylip" or "iphylip" (interleaved)
- **format** – sub-newick format

FORMAT	DESCRIPTION
0	flexible with support values
1	flexible with internal node names
2	all branches + leaf names + internal supports
3	all branches + all names
4	leaf branches + leaf names
5	internal and leaf branches + leaf names
6	internal branches + leaf names
7	leaf branches + all names
8	all names
9	leaf names
100	topology only

- **sp_naming_function** – Pointer to a parsing python function that receives nodename as first argument and returns the species name (see `PhyloNode.set_species_naming_function()`. By default, the 3 first letter of nodes will be used as species identifiers.

Returns a tree node object which represents the base of the tree.

get_age (*species2age*)

get_age_balanced_outgroup (*species2age*)

New in version 2.x. Returns the best outgroup according to topological ages and node sizes.

Currently Experimental !!

get_descendant_evol_events (*sos_thr=0.0*)

Returns a list of **all** duplication and speciation events detected after this node. Nodes are assumed to be duplications when a species overlap is found between its child linages. Method is described more detail in:

"The Human Phylome." Huerta-Cepas J, Dopazo H, Dopazo J, Gabaldon T. Genome Biol. 2007;8(6):R109.

get_farthest_oldest_leaf (*species2age*, *is_leaf_fn=None*)

Returns the farthest oldest leaf to the current one. It requires an species2age dictionary with the age estimation for all species.

Parameters is_leaf_fn (*None*) – A pointer to a function that receives a node instance as unique argument and returns True or False. It can be used to dynamically collapse nodes, so they are seen as leaves.

get_farthest_oldest_node(*species2age*)

New in version 2.1. Returns the farthest oldest node (leaf or internal). The difference with get_farthest_oldest_leaf() is that in this function internal nodes grouping seqs from the same species are collapsed.

get_my_evol_events(*sos_thr=0.0*)

Returns a list of duplication and speciation events in which the current node has been involved. Scanned nodes are also labeled internally as dup=True|False. You can access this labels using the 'node.dup' sintaxis.

Method: the algorithm scans all nodes from the given leafName to the root. Nodes are assumed to be duplications when a species overlap is found between its child linages. Method is described more detail in:

"The Human Phylome." Huerta-Cepas J, Dopazo H, Dopazo J, Gabaldon T. Genome Biol. 2007;8(6):R109.

get_species()

Returns the set of species covered by its partition.

is_monophyletic(*species*)

Returns True id species names under this node are all included in a given list or set of species names.

iter_species()

Returns an iterator over the species grouped by this node.

link_to_alignment(*alignment, alg_format='fasta'*)

reconcile(*species_tree*)

Returns the reconciled topology with the provided species tree, and a list of evolutionary events inferred from such reconciliation.

set_species_naming_function(*fn*)

Sets the parsing function used to extract species name from a node's name.

Parameters fn – Pointer to a parsing python function that receives nodename as first argument and returns the species name.

```
# Example of a parsing function to extract species names for
# all nodes in a given tree.
def parse_sp_name(node_name):
    return node_name.split("_")[1]
tree.set_species_naming_function(parse_sp_name)
```

species

PhyloTree

alias of `PhyloNode`

class EvolEvent

Basic evolutionary event. It stores all the information about an event(node) ocurred in a phylogenetic tree.

`etype` : `D` (Duplication), `S` (Speciation), `L` (gene loss), `in_seqs` : the list of sequences in one side of the event.

`out_seqs` : the list of sequences in the other side of the event

`node` : link to the event node in the tree

Contents

4.4 Clustering module

class **`ClusterNode`** (*newick=None*, *text_array=None*, *fdist=<function spearman_dist at 0x30360c8>*)

Bases: `ete2.coretype.tree.TreeNode`

Creates a new Cluster Tree object, which is a collection of ClusterNode instances connected in a hierarchical way, and representing a clustering result.

a newick file or string can be passed as the first argument. An ArrayTable file or instance can be passed as a second argument.

Examples: t1 = Tree() # creates an empty tree t2 = Tree('(A:1,(B:1,(C:1,D:1):0.5):0.5);') t3 = Tree('/home/user/myNewickFile.txt')

`get_dunn` (*clusters*, *fdist=None*)

Calculates the Dunn index for the given set of descendant nodes.

`get_leaf_profiles` ()

Returns the list of all the profiles associated to the leaves under this node.

`get_silhouette` (*fdist=None*)

Calculates the node's silhouette value by using a given distance function. By default, euclidean distance is used. It also calculates the deviation profile, mean profile, and inter/intra-cluster distances.

It sets the following features into the analyzed node:

- node.intracluster
- node.intercluster
- node.silhouete

intracluster distances a(i) are calculated as the Centroid Diameter

intercluster distances b(i) are calculated as the Centroid linkage distance

** Rousseeuw, P.J. (1987) Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. J. Comput. Appl. Math., 20, 53-65.

`iter_leaf_profiles` ()

Returns an iterator over all the profiles associated to the leaves under this node.

`link_to_arraytable` (*arraytbl*)

Allows to link a given arraytable object to the tree structure under this node. Row names in the arraytable object are expected to match leaf names.

Returns a list of nodes for with profiles could not been found in arraytable.

set_distance_function (*fn*)

Sets the distance function used to calculate cluster distances and silouette index.

ARGUMENTS:

fn: a pointer to python function acepting two arrays (numpy) as arguments.

EXAMPLE:

# A simple euclidean distance my_dist_fn = lambda x,y: abs(x-y) tree.set_distance_function(my_dist_fn)

ClusterTree

alias of `ClusterNode`

New in version 2.1.

4.5 Nexml module

4.5.1 Nexml classes linked to ETE

class **Nexml** (**args*, ***kargs*)

Creates a new nexml project.

build_from_file (*fname*, *index_otus=True*)

Populate Nexml project with data in a nexml file.

class **NexmlTree** (*newick=None*, *alignment=None*, *alg_format='fasta'*, *sp_naming_function=<function _parse_species at 0x280d410>*, *format=0*)

Special PhyloTree object with nexml support

4.5.2 Generic Nexml classes

class **AAChar** (*about=None*, *meta=None*, *label=None*, *id=None*, *tokens=None*, *states=None*, *codon=None*, *valueOf_=None*)

A concrete implementation of the AbstractChar element.

AAChar.**superclass**

alias of `AbstractChar`

class **AAFormat** (*about=None*, *meta=None*, *states=None*, *char=None*, *set=None*, *valueOf_=None*)

The AAFormat class is the container of amino acid column definitions.

AAFormat.**superclass**

alias of `AbstractFormat`

class **AAMapping** (*state=None*, *valueOf_=None*)

An IUPAC ambiguity mapping.

AAMapping.**superclass**

alias of `AbstractMapping`

class **AAMatrixObsRow**(*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *cell=None*, *set=None*, *valueOf_=None*)

This is a row in a matrix of amino acid data containing granular observations.

AAMatrixObsRow.**superclass**

alias of AbstractObsRow

class **AAMatrixSeqRow**(*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *seq=None*, *valueOf_=None*)

This is a row in a matrix of amino acid data containing raw sequence data.

AAMatrixSeqRow.**superclass**

alias of AbstractSeqRow

class **AAObs**(*about=None*, *meta=None*, *label=None*, *char=None*, *state=None*, *valueOf_=None*)

This is a single cell in a matrix containing an amino acid observation.

AAObs.**superclass**

alias of AbstractObs

class **AAObsMatrix**(*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)

A matrix of rows with single character observations.

AAObsMatrix.**superclass**

alias of AbstractObsMatrix

class **AAPolymorphicStateSet**(*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *member=None*, *uncertain_state_set=None*, *valueOf_=None*)

The AAPolymorphicStateSet defines a polymorphic ambiguity mapping.

AAPolymorphicStateSet.**superclass**

alias of AbstractPolymorphicStateSet

class **AASeqMatrix**(*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)

A matrix of rows with amino acid data as sequence strings.

AASeqMatrix.**superclass**

alias of AbstractSeqMatrix

class **AAState**(*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *valueOf_=None*)

This is a concrete implementation of the state element, which requires a symbol element, in this case restricted to AAToken, i.e. a single IUPAC amino acid symbol, and optional mapping elements to refer to other states.

AAState.**superclass**

alias of AbstractState

class **AAStates**(*about=None*, *meta=None*, *label=None*, *id=None*, *state=None*, *polymorphic_state_set=None*, *uncertain_state_set=None*, *set=None*, *valueOf_=None*)

A container for a set of states.

AAStates.**superclass**

alias of AbstractStates

class **AAUncertainStateSet**(*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *member=None*, *valueOf_=None*)

The AAUncertainStateSet defines an uncertain ambiguity mapping.

`AAUncertainStateSet.`**`superclass`**
alias of `AbstractUncertainStateSet`

class **`AbstractBlock`**(*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *valueOf_=None*)
The AbstractBlock is the superclass for blocks that contain an element structure of type AbstractFormat.

`AbstractBlock.`**`superclass`**
alias of `TaxaLinked`

class **`AbstractCells`**(*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *matrix=None*, *valueOf_=None*)
The AbstractSeqBlock type is the superclass for character blocks that consist of granular character state observations.

`AbstractCells.`**`superclass`**
alias of `AbstractBlock`

class **`AbstractChar`**(*about=None*, *meta=None*, *label=None*, *id=None*, *tokens=None*, *states=None*, *codon=None*, *valueOf_=None*)
The AbstractChar type is the superclass for a column definition, which may have a "states" attribute that refers to an AbstractStates element, a codon attribute of type CodonPosition and an id attribute that may be an actual id (e.g. for categorical matrices where observations explicitly refer to a column definition) or an integer for sequence matrices.

`AbstractChar.`**`superclass`**
alias of `IDTagged`

class **`AbstractEdge`**(*about=None*, *meta=None*, *label=None*, *id=None*, *source=None*, *length=None*, *target=None*, *valueOf_=None*)
The AbstractEdge superclass is what concrete edges inherit from by restriction. It represents an edge element much like that of GraphML, i.e. an element that connects node elements.

`AbstractEdge.`**`superclass`**
alias of `IDTagged`

class **`AbstractFormat`**(*about=None*, *meta=None*, *states=None*, *char=None*, *set=None*, *valueOf_=None*)
The AbstractFormat type is the superclass for the element that defines the allowed characters and states in a matrix, and their ambiguity mapping. It may enclose AbstractStates elements that define states and their mappings, and AbstractChar elements that specify which AbstractStates apply to which matrix columns.

`AbstractFormat.`**`superclass`**
alias of `Annotated`

class **`AbstractMapping`**(*state=None*, *valueOf_=None*)
The AbstractMapping type is the superclass for an ambiguity mapping. In an instance document, a subclass of this type will look like <member state="st1"/>, i.e. an element called "member" with an attribute called "state" whose value is an id reference that refers to an element that subclasses AbstractState. The purpose of AbstractMapping is to specify which other states may be implied, e.g. a nucleotide symbol "N" would have mappings to "A", "C", "G" and "T".

`AbstractMapping.`**`superclass`**
alias of `Base`

class **`AbstractNetwork`**(*about=None*, *meta=None*, *label=None*, *id=None*, *node=None*, *edge=None*, *set=None*, *valueOf_=None*)

The AbstractNetwork superclass is what a concrete network inherits from.

AbstractNetwork.**superclass**
alias of IDTagged

class AbstractNode(*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *root=False*, *valueOf_=None*)
The AbstractNode superclass is what concrete nodes inherit from by restriction. It represents a node element much like that of GraphML, i.e. an element that is connected into a tree by edge elements.

AbstractNode.**superclass**
alias of OptionalTaxonLinked

class AbstractObs(*about=None*, *meta=None*, *label=None*, *char=None*, *state=None*, *valueOf_=None*)
The AbstractObs type is the superclass for single observations, i.e. cells in a matrix. A concrete instance of AbstractObs has a "char" attribute that refers to an explicitly defined character (e.g. in categorical matrices), and a "state" attribute that either holds a reference to an explicitly defined state, or a raw state value (a continuous value).

AbstractObs.**superclass**
alias of Labelled

class AbstractObsMatrix(*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)
The AbstractObsMatrix super class is the abstract type for a <matrix> element that contains rows which hold granular state observations.

AbstractObsMatrix.**superclass**
alias of Annotated

class AbstractObsRow(*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *cell=None*, *set=None*, *valueOf_=None*)
The AbstractObsRow represents a single row in a matrix. The row must refer to a previously declared otu element by its id attribute (and must have an id itself, may have a label, and may have meta attachments). The row contains multiple cell elements.

AbstractObsRow.**superclass**
alias of TaxonLinked

class AbstractPolymorphicStateSet(*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *member=None*, *uncertain_state_set=None*, *valueOf_=None*)
The AbstractPolymorphicStateSet type is the super-class for a polymorphic state set definition. The element has a required AbstractSymbol attribute that in restricted concrete subclasses must be of a sensible type such as a single IUPAC character. It may enclose zero or more AbstractMapping elements to resolve ambiguities.

AbstractPolymorphicStateSet.**superclass**
alias of AbstractUncertainStateSet

class AbstractRootEdge(*about=None*, *meta=None*, *label=None*, *id=None*, *length=None*, *target=None*, *valueOf_=None*)
The AbstractRootEdge complex type is a superclass for the edge that leads into a root, i.e. an edge with only a target attribute, but no source attribute. This type of edge is used for coalescent trees, where the initial lineage has a certain length before things start splitting up.

AbstractRootEdge.**superclass**
alias of IDTagged

class AbstractSeqMatrix (*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)
The AbstractSeqMatrix super class is the abstract type for a <matrix> element that contains rows which hold raw character sequences.

AbstractSeqMatrix.**superclass**
alias of Annotated

class AbstractSeqRow (*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *seq=None*, *valueOf_=None*)
The AbstractSeqRow represents a single row in a matrix. The row must refer to a previously declared otu element by its id attribute (and must have an id itself, may have a label, and may have meta attachments). The row contains a single seq element with raw character data.

AbstractSeqRow.**superclass**
alias of TaxonLinked

class AbstractSeqs (*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *matrix=None*, *valueOf_=None*)
The AbstractSeqBlock type is the superclass for character blocks that consist of raw character sequences.

AbstractSeqs.**superclass**
alias of AbstractBlock

class AbstractState (*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *valueOf_=None*)
The AbstractState type is the super-class for a state definition. The element has a required symbol attribute that in restricted concrete subclasses must be of a sensible type such as a single IUPAC character. It may enclose zero or more AbstractMapping elements to resolve ambiguities.

AbstractState.**superclass**
alias of IDTagged

class AbstractStates (*about=None*, *meta=None*, *label=None*, *id=None*, *state=None*, *polymorphic_state_set=None*, *uncertain_state_set=None*, *set=None*, *valueOf_=None*)
A container for a set of AbstractState elements.

AbstractStates.**superclass**
alias of IDTagged

class AbstractTree (*about=None*, *meta=None*, *label=None*, *id=None*, *node=None*, *rootedge=None*, *edge=None*, *set=None*, *valueOf_=None*)
The AbstractTree superclass is what a concrete tree inherits from.

AbstractTree.**superclass**
alias of IDTagged

class AbstractTrees (*about=None*, *meta=None*, *label=None*, *id=None*, *network=None*, *tree=None*, *set=None*, *valueOf_=None*)
The AbstractTrees superclass is what concrete trees inherit from.

AbstractTrees.**superclass**
alias of IDTagged

class AbstractUncertainStateSet(*about=None, meta=None, label=None, id=None, symbol=None, member=None, valueOf_=None*)

The AbstractUncertainStateSet type is the super-class for an uncertain state set definition. The element has a required AbstractSymbol attribute that in restricted concrete subclasses must be of a sensible type such as a single IUPAC character. It may enclose zero or more AbstractMapping elements to resolve ambiguities.

`AbstractUncertainStateSet.`**superclass**

alias of `AbstractState`

class Annotated(*about=None, meta=None, valueOf_=None*)

The Annotated complexType is a super class for objects that optionally have metadata annotations of type Meta.

`Annotated.`**superclass**

alias of `Base`

class Base(*valueOf_=None*)

The base type for all complexType definitions in the nexml schema. This type allows a number of special attributes: xml:lang - for languages codes xml:base - see http://www.w3.org/TR/xmlbase/ xml:id - see http://www.w3.org/TR/xml-id/ xml:space - for whitespace handling xlink:href - for links Also see http://www.w3.org/2001/xml.xsd for more information on the xml and xlink attributes.

class ContinuousCells(*about=None, meta=None, label=None, id=None, otus=None, format=None, matrix=None, valueOf_=None*)

A continuous characters block consisting of granular cells preceded by metadata.

`ContinuousCells.`**superclass**

alias of `AbstractCells`

class ContinuousChar(*about=None, meta=None, label=None, id=None, tokens=None, states=None, codon=None, valueOf_=None*)

A concrete implementation of the char element, which requires only an id attribute.

`ContinuousChar.`**superclass**

alias of `AbstractChar`

class ContinuousFormat(*about=None, meta=None, states=None, char=None, set=None, valueOf_=None*)

The ContinuousFormat class is the container of continuous column definitions.

`ContinuousFormat.`**superclass**

alias of `AbstractFormat`

class ContinuousMatrixObsRow(*about=None, meta=None, label=None, id=None, otu=None, cell=None, set=None, valueOf_=None*)

This is a row in a matrix of continuous data as granular obervations.

`ContinuousMatrixObsRow.`**superclass**

alias of `AbstractObsRow`

class ContinuousMatrixSeqRow(*about=None, meta=None, label=None, id=None, otu=None, seq=None, valueOf_=None*)

This is a row in a matrix of continuous data as character sequences.

`ContinuousMatrixSeqRow.`**superclass**

alias of `AbstractSeqRow`

class ContinuousObs (*about=None*, *meta=None*, *label=None*, *char=None*, *state=None*, *valueOf_=None*)

This is a single cell in a matrix containing a continuous observation.

ContinuousObs.**superclass**

alias of AbstractObs

class ContinuousObsMatrix (*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)

A matrix of rows with single character observations.

ContinuousObsMatrix.**superclass**

alias of AbstractObsMatrix

class ContinuousSeqMatrix (*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)

A matrix of rows with seq strings of type continuous.

ContinuousSeqMatrix.**superclass**

alias of AbstractSeqMatrix

class ContinuousSeqs (*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *matrix=None*, *valueOf_=None*)

A continuous characters block consisting of float sequences preceded by metadata.

ContinuousSeqs.**superclass**

alias of AbstractSeqs

class DNAChar (*about=None*, *meta=None*, *label=None*, *id=None*, *tokens=None*, *states=None*, *codon=None*, *valueOf_=None*)

A concrete implementation of the AbstractChar element.

DNAChar.**superclass**

alias of AbstractChar

class DNAFormat (*about=None*, *meta=None*, *states=None*, *char=None*, *set=None*, *valueOf_=None*)

The DNAFormat class is the container of DNA column definitions.

DNAFormat.**superclass**

alias of AbstractFormat

class DNAMapping (*state=None*, *valueOf_=None*)

An IUPAC ambiguity mapping.

DNAMapping.**superclass**

alias of AbstractMapping

class DNAMatrixObsRow (*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *cell=None*, *set=None*, *valueOf_=None*)

This is a row in a matrix of DNA data containing granular observations.

DNAMatrixObsRow.**superclass**

alias of AbstractObsRow

class DNAMatrixSeqRow (*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *seq=None*, *valueOf_=None*)

This is a row in a matrix of DNA data containing raw sequence data.

DNAMatrixSeqRow.**superclass**

alias of AbstractSeqRow

class DNAObs (*about=None*, *meta=None*, *label=None*, *char=None*, *state=None*, *valueOf_=None*)

This is a single cell in a matrix containing a nucleotide observation.

`DNAObs.`**superclass**

alias of `AbstractObs`

class DNAObsMatrix (*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)

A matrix of rows with single character observations.

`DNAObsMatrix.`**superclass**

alias of `AbstractObsMatrix`

class DNAPolymorphicStateSet (*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *member=None*, *uncertain_state_set=None*, *valueOf_=None*)

The DNAPolymorphicStateSet type defines an IUPAC ambiguity mapping. It may enclose zero or more AbstractMapping elements to resolve ambiguities.

`DNAPolymorphicStateSet.`**superclass**

alias of `AbstractPolymorphicStateSet`

class DNASeqMatrix (*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)

A matrix of rows with seq strings of type DNA.

`DNASeqMatrix.`**superclass**

alias of `AbstractSeqMatrix`

class DNAState (*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *valueOf_=None*)

This is a concrete implementation of the state element, which requires a symbol element, in this case restricted to DNAToken, i.e. a single IUPAC nucleotide symbol, and optional mapping elements to refer to other states.

`DNAState.`**superclass**

alias of `AbstractState`

class DNAStates (*about=None*, *meta=None*, *label=None*, *id=None*, *state=None*, *polymorphic_state_set=None*, *uncertain_state_set=None*, *set=None*, *valueOf_=None*)

A container for a set of states.

`DNAStates.`**superclass**

alias of `AbstractStates`

class DNAUncertainStateSet (*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *member=None*, *valueOf_=None*)

The DNAUncertainStateSet type defines an IUPAC ambiguity mapping. It may enclose zero or more AbstractMapping elements to resolve ambiguities.

`DNAUncertainStateSet.`**superclass**

alias of `AbstractUncertainStateSet`

class DnaCells (*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *matrix=None*, *valueOf_=None*)

A DNA characters block consisting of granular cells preceded by metadata.

`DnaCells.`**superclass**

alias of `AbstractCells`

class **DnaSeqs** (*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *matrix=None*, *valueOf_=None*)
A DNA characters block consisting of sequences preceded by metadata.

`DnaSeqs`.**superclass**
alias of `AbstractSeqs`

class **FloatNetwork** (*about=None*, *meta=None*, *label=None*, *id=None*, *node=None*, *edge=None*, *set=None*, *valueOf_=None*)
A concrete network implementation, with floating point edge lengths.

`FloatNetwork`.**superclass**
alias of `AbstractNetwork`

class **FloatTree** (*about=None*, *meta=None*, *label=None*, *id=None*, *node=None*, *rootedge=None*, *edge=None*, *set=None*, *valueOf_=None*)
A concrete tree implementation, with floating point edge lengths.

`FloatTree`.**subclass**
alias of `NexmlTree`

`FloatTree`.**superclass**
alias of `AbstractTree`

class **IDTagged** (*about=None*, *meta=None*, *label=None*, *id=None*, *valueOf_=None*)
The IDTagged complexType is a super class for objects that require unique id attributes of type xs:ID. The id must be unique within the XML document.

`IDTagged`.**superclass**
alias of `Labelled`

class **IntNetwork** (*about=None*, *meta=None*, *label=None*, *id=None*, *node=None*, *edge=None*, *set=None*, *valueOf_=None*)
A concrete network implementation, with integer edge lengths.

`IntNetwork`.**superclass**
alias of `AbstractNetwork`

class **IntTree** (*about=None*, *meta=None*, *label=None*, *id=None*, *node=None*, *rootedge=None*, *edge=None*, *set=None*, *valueOf_=None*)
A concrete tree implementation, with integer edge lengths.

`IntTree`.**subclass**
alias of `NexmlTree`

`IntTree`.**superclass**
alias of `AbstractTree`

class **Labelled** (*about=None*, *meta=None*, *label=None*, *valueOf_=None*)
The Labelled complexType is a super class for objects that optionally have label attributes to use as a (non-unique) name of type xs:string.

`Labelled`.**superclass**
alias of `Annotated`

class **LiteralMeta** (*datatype=None*, *content=None*, *property=None*, *valueOf_=None*)
Metadata annotations in which the object is a literal value. If the @content attribute is used, then the element should contain no children.

class **NetworkFloatEdge** (*about=None*, *meta=None*, *label=None*, *id=None*, *source=None*, *length=None*, *target=None*, *valueOf_=None*)
A concrete network edge implementation, with float edge.

NetworkFloatEdge.**superclass**
alias of AbstractEdge

class **NetworkIntEdge** (*about=None*, *meta=None*, *label=None*, *id=None*, *source=None*, *length=None*, *target=None*, *valueOf_=None*)
A concrete network edge implementation, with int edge.

NetworkIntEdge.**superclass**
alias of AbstractEdge

class **NetworkNode** (*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *root=False*, *valueOf_=None*)
A concrete network node implementation.

NetworkNode.**superclass**
alias of AbstractNode

class **Nexml** (**args*, ***kargs*)
Creates a new nexml project.

Nexml.**build_from_file** (*fname*, *index_otus=True*)
Populate Nexml project with data in a nexml file.

class **OptionalTaxonLinked** (*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *valueOf_=None*)
The OptionalOTULinked complexType is a super class for objects that that optionally have an otu id reference.

OptionalTaxonLinked.**superclass**
alias of IDTagged

class **ProteinCells** (*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *matrix=None*, *valueOf_=None*)
An amino acid characters block consisting of granular cells preceded by metadata.

ProteinCells.**superclass**
alias of AbstractCells

class **ProteinSeqs** (*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *matrix=None*, *valueOf_=None*)
An amino acid characters block consisting of sequences preceded by metadata.

ProteinSeqs.**superclass**
alias of AbstractSeqs

class **RNAChar** (*about=None*, *meta=None*, *label=None*, *id=None*, *tokens=None*, *states=None*, *codon=None*, *valueOf_=None*)
A concrete implementation of the AbstractChar element, i.e. a single column in an alignment.

RNAChar.**superclass**
alias of AbstractChar

class **RNAFormat** (*about=None*, *meta=None*, *states=None*, *char=None*, *set=None*, *valueOf_=None*)
The RNAFormat class is the container of RNA column definitions.

RNAFormat.**superclass**
alias of AbstractFormat

class **RNAMapping** (*state=None*, *valueOf_=None*)
An IUPAC RNA ambiguity mapping.

RNAMapping.**superclass**
alias of AbstractMapping

class **RNAMatrixObsRow** (*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *cell=None*, *set=None*, *valueOf_=None*)
This is a row in a matrix of RNA data containing granular observations.

RNAMatrixObsRow.**superclass**
alias of AbstractObsRow

class **RNAMatrixSeqRow** (*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *seq=None*, *valueOf_=None*)
This is a row in a matrix of RNA data containing raw sequence data.

RNAMatrixSeqRow.**superclass**
alias of AbstractSeqRow

class **RNAObs** (*about=None*, *meta=None*, *label=None*, *char=None*, *state=None*, *valueOf_=None*)
This is a single cell in a matrix containing an RNA nucleotide observation.

RNAObs.**superclass**
alias of AbstractObs

class **RNAObsMatrix** (*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)
A matrix of rows with single character observations.

RNAObsMatrix.**superclass**
alias of AbstractObsMatrix

class **RNAPolymorphicStateSet** (*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *member=None*, *uncertain_state_set=None*, *valueOf_=None*)
The RNAPolymorphicStateSet describes a single polymorphic IUPAC ambiguity mapping.

RNAPolymorphicStateSet.**superclass**
alias of AbstractPolymorphicStateSet

class **RNASeqMatrix** (*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)
A matrix of rows with seq strings of type RNA.

RNASeqMatrix.**superclass**
alias of AbstractSeqMatrix

class **RNAState** (*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *valueOf_=None*)
This is a concrete implementation of the state element, which requires a symbol attribute, in this case restricted to RNAToken, i.e. a single IUPAC nucleotide symbol, and optional mapping elements to refer to other states.

RNAState.**superclass**
alias of AbstractState

class RNAStates(*about=None*, *meta=None*, *label=None*, *id=None*, *state=None*, *polymorphic_state_set=None*, *uncertain_state_set=None*, *set=None*, *valueOf_=None*)

A container for a set of states.

`RNAStates.`**superclass**

alias of `AbstractStates`

class RNAUncertainStateSet(*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *member=None*, *valueOf_=None*)

The RNAUncertainStateSet describes a single uncertain IUPAC ambiguity mapping.

`RNAUncertainStateSet.`**superclass**

alias of `AbstractUncertainStateSet`

class ResourceMeta(*href=None*, *rel=None*, *meta=None*, *valueOf_=None*)

Metadata annotations in which the object is a resource. If this element contains meta elements as children, then the object of this annotation is a "blank node".

class RestrictionCells(*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *matrix=None*, *valueOf_=None*)

A standard characters block consisting of granular cells preceded by metadata.

`RestrictionCells.`**superclass**

alias of `AbstractCells`

class RestrictionChar(*about=None*, *meta=None*, *label=None*, *id=None*, *tokens=None*, *states=None*, *codon=None*, *valueOf_=None*)

A concrete implementation of the char element, which requires a unique identifier and a state set reference.

`RestrictionChar.`**superclass**

alias of `AbstractChar`

class RestrictionFormat(*about=None*, *meta=None*, *states=None*, *char=None*, *set=None*, *valueOf_=None*)

The RestrictionFormat class is the container of restriction column definitions.

`RestrictionFormat.`**superclass**

alias of `AbstractFormat`

class RestrictionMatrixObsRow(*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *cell=None*, *set=None*, *valueOf_=None*)

This is a row in a matrix of restriction site data as granular obervations.

`RestrictionMatrixObsRow.`**superclass**

alias of `AbstractObsRow`

class RestrictionMatrixSeqRow(*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *seq=None*, *valueOf_=None*)

This is a row in a matrix of restriction site data as character sequences.

`RestrictionMatrixSeqRow.`**superclass**

alias of `AbstractSeqRow`

class RestrictionObs(*about=None*, *meta=None*, *label=None*, *char=None*, *state=None*, *valueOf_=None*)

This is a single cell in a matrix containing a restriction site observation.

`RestrictionObs.`**superclass**

alias of `AbstractObs`

class `RestrictionObsMatrix`(*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)

A matrix of rows with single character observations.

`RestrictionObsMatrix.`**`superclass`**

alias of `AbstractObsMatrix`

class `RestrictionSeqMatrix`(*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)

A matrix of rows with seq strings of type restriction.

`RestrictionSeqMatrix.`**`superclass`**

alias of `AbstractSeqMatrix`

class `RestrictionSeqs`(*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *matrix=None*, *valueOf_=None*)

A restriction site characters block consisting of sequences preceded by metadata.

`RestrictionSeqs.`**`superclass`**

alias of `AbstractSeqs`

class `RestrictionState`(*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *valueOf_=None*)

This is a concrete implementation of the state element, which requires a symbol element, in this case restricted to 0/1.

`RestrictionState.`**`superclass`**

alias of `AbstractState`

class `RestrictionStates`(*about=None*, *meta=None*, *label=None*, *id=None*, *state=None*, *polymorphic_state_set=None*, *uncertain_state_set=None*, *set=None*, *valueOf_=None*)

A container for a set of states.

`RestrictionStates.`**`superclass`**

alias of `AbstractStates`

class `RnaCells`(*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *matrix=None*, *valueOf_=None*)

A RNA characters block consisting of granular cells preceded by metadata.

`RnaCells.`**`superclass`**

alias of `AbstractCells`

class `RnaSeqs`(*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *matrix=None*, *valueOf_=None*)

A RNA characters block consisting of sequences preceded by metadata.

`RnaSeqs.`**`superclass`**

alias of `AbstractSeqs`

class `StandardCells`(*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *matrix=None*, *valueOf_=None*)

A standard characters block consisting of granular cells preceded by metadata.

`StandardCells.`**`superclass`**

alias of `AbstractCells`

class `StandardChar`(*about=None*, *meta=None*, *label=None*, *id=None*, *tokens=None*, *states=None*, *codon=None*, *valueOf_=None*)

A concrete implementation of the char element, which requires a states attribute to refer to a set of defined states

`StandardChar.`**`superclass`**
alias of `AbstractChar`

class **`StandardFormat`** (*about=None*, *meta=None*, *states=None*, *char=None*, *set=None*, *valueOf_=None*)
The StandardFormat class is the container of standard column definitions.

`StandardFormat.`**`superclass`**
alias of `AbstractFormat`

class **`StandardMapping`** (*state=None*, *valueOf_=None*)
A standard character ambiguity mapping.

`StandardMapping.`**`superclass`**
alias of `AbstractMapping`

class **`StandardMatrixObsRow`** (*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *cell=None*, *set=None*, *valueOf_=None*)
This is a row in a matrix of standard data as granular obervations.

`StandardMatrixObsRow.`**`superclass`**
alias of `AbstractObsRow`

class **`StandardMatrixSeqRow`** (*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *seq=None*, *valueOf_=None*)
This is a row in a matrix of standard data as character sequences.

`StandardMatrixSeqRow.`**`superclass`**
alias of `AbstractSeqRow`

class **`StandardObs`** (*about=None*, *meta=None*, *label=None*, *char=None*, *state=None*, *valueOf_=None*)
This is a single cell in a matrix containing a standard observation.

`StandardObs.`**`superclass`**
alias of `AbstractObs`

class **`StandardObsMatrix`** (*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)
A matrix of rows with single character observations.

`StandardObsMatrix.`**`superclass`**
alias of `AbstractObsMatrix`

class **`StandardPolymorphicStateSet`** (*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *member=None*, *uncertain_state_set=None*, *valueOf_=None*)
The StandardPolymorphicStateSet type is a single polymorphic ambiguity mapping.

`StandardPolymorphicStateSet.`**`superclass`**
alias of `AbstractPolymorphicStateSet`

class **`StandardSeqMatrix`** (*about=None*, *meta=None*, *row=None*, *set=None*, *valueOf_=None*)
A matrix of rows with seq strings of type standard.

`StandardSeqMatrix.`**`superclass`**
alias of `AbstractSeqMatrix`

class StandardSeqs(*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *format=None*, *matrix=None*, *valueOf_=None*)
A standard characters block consisting of sequences preceded by metadata.

StandardSeqs.**superclass**
alias of AbstractSeqs

class StandardState(*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *valueOf_=None*)
This is a concrete implementation of the state element, which requires a symbol element, in this case restricted to integers, and optional mapping elements to refer to other states.

StandardState.**superclass**
alias of AbstractState

class StandardStates(*about=None*, *meta=None*, *label=None*, *id=None*, *state=None*, *polymorphic_state_set=None*, *uncertain_state_set=None*, *set=None*, *valueOf_=None*)
A container for a set of states.

StandardStates.**superclass**
alias of AbstractStates

class StandardUncertainStateSet(*about=None*, *meta=None*, *label=None*, *id=None*, *symbol=None*, *member=None*, *valueOf_=None*)
The StandardUncertainStateSet type is a single uncertain ambiguity mapping.

StandardUncertainStateSet.**superclass**
alias of AbstractUncertainStateSet

class TaxaLinked(*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *valueOf_=None*)
The TaxaLinked complexType is a super class for objects that that require an otus id reference.

TaxaLinked.**superclass**
alias of IDTagged

class TaxonLinked(*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *valueOf_=None*)
The TaxonLinked complexType is a super class for objects that require a taxon id reference.

TaxonLinked.**superclass**
alias of IDTagged

class TreeFloatEdge(*about=None*, *meta=None*, *label=None*, *id=None*, *source=None*, *length=None*, *target=None*, *valueOf_=None*)
A concrete edge implementation, with float length.

TreeFloatEdge.**superclass**
alias of AbstractEdge

class TreeFloatRootEdge(*about=None*, *meta=None*, *label=None*, *id=None*, *length=None*, *target=None*, *valueOf_=None*)
A concrete root edge implementation, with float length.

TreeFloatRootEdge.**superclass**
alias of AbstractRootEdge

class TreeIntEdge(*about=None*, *meta=None*, *label=None*, *id=None*, *source=None*, *length=None*, *target=None*, *valueOf_=None*)
A concrete edge implementation, with int length.

TreeIntEdge.**superclass**
alias of AbstractEdge

class **TreeIntRootEdge** (*about=None*, *meta=None*, *label=None*, *id=None*, *length=None*, *target=None*, *valueOf_=None*)
A concrete root edge implementation, with int length.

TreeIntRootEdge.**superclass**
alias of AbstractRootEdge

class **TreeNode** (*about=None*, *meta=None*, *label=None*, *id=None*, *otu=None*, *root=False*, *valueOf_=None*)
A concrete node implementation.

TreeNode.**superclass**
alias of AbstractNode

class **Trees** (*about=None*, *meta=None*, *label=None*, *id=None*, *otus=None*, *network=None*, *tree=None*, *set=None*, *valueOf_=None*)
A concrete container for tree objects.

Trees.**superclass**
alias of TaxaLinked

class **attrExtensions** (*valueOf_=None*)
This element is for use in WSDL 1.1 only. It does not apply to WSDL 2.0 documents. Use in WSDL 2.0 documents is invalid.

class **Nexml** (**args*, ***kargs*)
Creates a new nexml project.

Nexml.**build_from_file** (*fname*, *index_otus=True*)
Populate Nexml project with data in a nexml file.

class **NexmlTree** (*newick=None*, *alignment=None*, *alg_format='fasta'*, *sp_naming_function=<function _parse_species at 0x280d410>*, *format=0*)
Special PhyloTree object with nexml support

New in version 2.1.

4.6 Phyloxml Module

4.6.1 Phyloxml classes linked to ETE

class **Phyloxml** (**args*, ***kargs*)

class **PhyloxmlTree** (*phyloxml_clade=None*, *phyloxml_phylogeny=None*, ***kargs*)
PhyloTree object supporting phyloXML format.

4.6.2 Generic Phyloxml classes

class **Accession** (*source=None*, *valueOf_=None*)
Element Accession is used to capture the local part in a sequence identifier (e.g. 'P17304' in 'UniProtKB:P17304', in which case the 'source' attribute would be 'UniProtKB').

class Annotation (*source=None*, *type_=None*, *ref=None*, *evidence=None*, *desc=None*, *confidence=None*, *property=None*, *uri=None*, *valueOf_=None*)

The annotation of a molecular sequence. It is recommended to annotate by using the optional 'ref' attribute (some examples of acceptable values for the ref attribute: 'GO:0008270', 'KEGG:Tetrachloroethene degradation', 'EC:1.1.1.1'). Optional element 'desc' allows for a free text description. Optional element 'confidence' is used to state the type and value of support for a annotation. Similarly, optional attribute 'evidence' is used to describe the evidence for a annotation as free text (e.g. 'experimental'). Optional element 'property' allows for further, typed and referenced annotations from external resources.

class BinaryCharacters (*lost_count=None*, *absent_count=None*, *present_count=None*, *type_=None*, *gained_count=None*, *gained=None*, *lost=None*, *present=None*, *absent=None*, *valueOf_=None*)

The names and/or counts of binary characters present, gained, and lost at the root of a clade.

class BranchColor (*red=None*, *green=None*, *blue=None*, *valueOf_=None*)

This indicates the color of a clade when rendered (the color applies to the whole clade unless overwritten by the color(s) of sub clades).

class Clade (*id_source=None*, *branch_length_attr=None*, *name=None*, *branch_length=None*, *confidence=None*, *width=None*, *color=None*, *node_id=None*, *taxonomy=None*, *sequence=None*, *events=None*, *binary_characters=None*, *distribution=None*, *date=None*, *reference=None*, *property=None*, *clade=None*, *valueOf_=None*)

Element Clade is used in a recursive manner to describe the topology of a phylogenetic tree. The parent branch length of a clade can be described either with the 'branch_length' element or the 'branch_length' attribute (it is not recommended to use both at the same time, though). Usage of the 'branch_length' attribute allows for a less verbose description. Element 'confidence' is used to indicate the support for a clade/parent branch. Element 'events' is used to describe such events as gene-duplications at the root node/parent branch of a clade. Element 'width' is the branch width for this clade (including parent branch). Both 'color' and 'width' elements apply for the whole clade unless overwritten in-sub clades. Attribute 'id_source' is used to link other elements to a clade (on the xml-level).

class CladeRelation (*id_ref_0=None*, *id_ref_1=None*, *type_=None*, *distance=None*, *confidence=None*, *valueOf_=None*)

This is used to express a typed relationship between two clades. For example it could be used to describe multiple parents of a clade.

class Confidence (*type_=None*, *valueOf_=None*)

A general purpose confidence element. For example this can be used to express the bootstrap support value of a clade (in which case the 'type' attribute is 'bootstrap').

class Date (*unit=None*, *desc=None*, *value=None*, *minimum=None*, *maximum=None*, *valueOf_=None*)

A date associated with a clade/node. Its value can be numerical by using the 'value' element and/or free text with the 'desc' element' (e.g. 'Silurian'). If a numerical value is used, it is recommended to employ the 'unit' attribute to indicate the type of the numerical value (e.g. 'mya' for 'million years ago'). The elements 'minimum' and 'maximum' are used the indicate a range/confidence interval

class Distribution (*desc=None*, *point=None*, *polygon=None*, *valueOf_=None*)

The geographic distribution of the items of a clade (species, sequences), intended for phylogeographic applications. The location can be described either by free text in the 'desc' element and/or by the coordinates of one or more 'Points' (similar to the 'Point' element in Google's KML format) or by 'Polygons'.

class `DomainArchitecture` (*length=None*, *domain=None*, *valueOf_=None*)

This is used describe the domain architecture of a protein. Attribute 'length' is the total length of the protein

class `Events` (*type_=None*, *duplications=None*, *speciations=None*, *losses=None*, *confidence=None*, *valueOf_=None*)

Events at the root node of a clade (e.g. one gene duplication).

class `Id` (*provider=None*, *valueOf_=None*)

A general purpose identifier element. Allows to indicate the provider (or authority) of an identifier.

class `MolSeq` (*is_aligned=None*, *valueOf_=None*)

Element 'mol_seq' is used to store molecular sequences. The 'is_aligned' attribute is used to indicated that this molecular sequence is aligned with all other sequences in the same phylogeny for which 'is aligned' is true as well (which, in most cases, means that gaps were introduced, and that all sequences for which 'is aligned' is true must have the same length).

class `Phylogeny` (*rerootable=None*, *branch_length_unit=None*, *type_=None*, *rooted=None*, *name=None*, *id=None*, *description=None*, *date=None*, *confidence=None*, *clade=None*, *clade_relation=None*, *sequence_relation=None*, *property=None*, *valueOf_=None*)

Element Phylogeny is used to represent a phylogeny. The required attribute 'rooted' is used to indicate whether the phylogeny is rooted or not. The attribute 'rerootable' can be used to indicate that the phylogeny is not allowed to be rooted differently (i.e. because it is associated with root dependent data, such as gene duplications). The attribute 'type' can be used to indicate the type of phylogeny (i.e. 'gene tree'). It is recommended to use the attribute 'branch_length_unit' if the phylogeny has branch lengths. Element clade is used in a recursive manner to describe the topology of a phylogenetic tree.

`Phylogeny.`**`subclass`**

alias of `PhyloxmlTree`

class `Point` (*geodetic_datum=None*, *alt_unit=None*, *lat=None*, *long=None*, *alt=None*, *valueOf_=None*)

The coordinates of a point with an optional altitude (used by element 'Distribution'). Required attributes are the 'geodetic_datum' used to indicate the geodetic datum (also called 'map datum', for example Google's KML uses 'WGS84'). Attribute 'alt_unit' is the unit for the altitude (e.g. 'meter').

class `Polygon` (*point=None*, *valueOf_=None*)

A polygon defined by a list of 'Points' (used by element 'Distribution').

class `Property` (*datatype=None*, *id_ref=None*, *ref=None*, *applies_to=None*, *unit=None*, *valueOf_=None*, *mixedclass_=None*, *content_=None*)

Property allows for typed and referenced properties from external resources to be attached to 'Phylogeny', 'Clade', and 'Annotation'. The value of a property is its mixed (free text) content. Attribute 'datatype' indicates the type of a property and is limited to xsd-datatypes (e.g. 'xsd:string', 'xsd:boolean', 'xsd:integer', 'xsd:decimal', 'xsd:float', 'xsd:double', 'xsd:date', 'xsd:anyURI'). Attribute 'applies_to' indicates the item to which a property applies to (e.g. 'node' for the parent node of a clade, 'parent_branch' for the parent branch of a clade). Attribute 'id_ref' allows to attached a property specifically to one element (on the xml-level). Optional attribute 'unit' is used to indicate the unit of the property. An example: <property datatype="xsd:integer" ref="NOAA:depth" applies_to="clade" unit="METRIC:m"> 200 </property>

class `ProteinDomain` (*to=None*, *confidence=None*, *fromxx=None*, *id=None*, *valueOf_=None*)

To represent an individual domain in a domain architecture. The name/unique identifier is described via the 'id' attribute. 'confidence' can be used to store (i.e.) E-values.

class `Reference` (*doi=None*, *desc=None*, *valueOf_=None*)

A literature reference for a clade. It is recommended to use the 'doi' attribute instead of the free text 'desc' element whenever possible.

class `Sequence` (*id_source=None*, *id_ref=None*, *type_=None*, *symbol=None*, *accession=None*, *name=None*, *location=None*, *mol_seq=None*, *uri=None*, *annotation=None*, *domain_architecture=None*, *valueOf_=None*)

Element Sequence is used to represent a molecular sequence (Protein, DNA, RNA) associated with a node. 'symbol' is a short (maximal ten characters) symbol of the sequence (e.g. 'ACTM') whereas 'name' is used for the full name (e.g. 'muscle Actin'). 'location' is used for the location of a sequence on a genome/chromosome. The actual sequence can be stored with the 'mol_seq' element. Attribute 'type' is used to indicate the type of sequence ('dna', 'rna', or 'protein'). One intended use for 'id_ref' is to link a sequence to a taxonomy (via the taxonomy's 'id_source') in case of multiple sequences and taxonomies per node.

class `SequenceRelation` (*id_ref_0=None*, *id_ref_1=None*, *type_=None*, *distance=None*, *confidence=None*, *valueOf_=None*)

This is used to express a typed relationship between two sequences. For example it could be used to describe an orthology (in which case attribute 'type' is 'orthology').

class `Taxonomy` (*id_source=None*, *id=None*, *code=None*, *scientific_name=None*, *authority=None*, *common_name=None*, *synonym=None*, *rank=None*, *uri=None*, *valueOf_=None*)

Element Taxonomy is used to describe taxonomic information for a clade. Element 'code' is intended to store UniProt/Swiss-Prot style organism codes (e.g. 'APLCA' for the California sea hare 'Aplysia californica') or other styles of mnemonics (e.g. 'Aca'). Element 'authority' is used to keep the authority, such as 'J. G. Cooper, 1863', associated with the 'scientific_name'. Element 'id' is used for a unique identifier of a taxon (for example '6500' with 'ncbi_taxonomy' as 'provider' for the California sea hare). Attribute 'id_source' is used to link other elements to a taxonomy (on the xml-level).

class `Uri` (*type_=None*, *desc=None*, *valueOf_=None*)

A uniform resource identifier. In general, this is expected to be an URL (for example, to link to an image on a website, in which case the 'type' attribute might be 'image' and 'desc' might be 'image of a California sea hare').

class `PhyloxmlTree` (*phyloxml_clade=None*, *phyloxml_phylogeny=None*, ***kargs*)

PhyloTree object supporting phyloXML format.

4.7 PhylomeDB3 Connector

class `PhylomeDB3Connector` (*host='84.88.66.245'*, *db='phylomedb_3'*, *user='public'*, *passwd='public'*, *port=3306*)

Returns a connector to a phylomeDB3 database.

> db: database name in the host server. host: hostname in which phylomeDB is hosted. user: username to the database. port: port used to connect database. passwd: password to connect database.
>
> An object whose methods can be used to query the database.

count_algs (*phylome_id*)
Returns how many alignments are for a given phylome

count_trees (*phylome_id*)
Retuns the frequency of each evolutionary method in the input phylome

get_algs (*id*, *phylome_id*, *raw_alg=True*, *clean_alg=True*)
Return the either the clean, the raw or both alignments for the input phylomeDB ID in the input phylome

get_all_isoforms (*id*)
Returns all the isoforms registered for the input phylomeDB ID

get_available_trees_by_phylome (*id*, *collateral=True*)
Returns information about which methods have been used to reconstruct every tree for a given phylomeDB ID grouped by phylome

get_best_tree (*id*, *phylome_id*)
return a tree for input id in the given phylome for the best fitting evolutionary model in terms of LK

get_clean_alg (*id*, *phylome_id*)
Return the raw alignment for the input phylomeDB ID in the given phylome

get_collateral_seeds (*protid*)
Return the trees where the protid is presented as part of the homolog sequences to the seed protein

get_external_ids (*ids*)
Returns all the external IDs registered in the 'external_id' table that are associated to the input phylomeDB IDs

get_genome_ids (*taxid*, *version*, *filter_isoforms=True*)
Returns the phylomeDB IDs for a given genome in the database filtering out, or not, the different isoforms for each ID

get_genome_info (*genome*)
Returns all available information about a registered genome/proteome

get_genomes ()
Returns all current available genomes/proteomes

get_genomes_by_species (*taxid*)
Return all the proteomes/genomes registered for the input taxaid code

get_go_ids (*ids*)
Returns all available GO Terms associated to the input phylomeDB IDs

get_id_by_external (*external*)
Returns the protein id associated to a given external id

get_id_translations (*id*)
Returns all the registered translations of a given phylomeDB ID

get_info_homologous_seqs (*protid*, *phylome_id*, *tree=None*, *tree_method=False*, *sequence=False*)
Return all the available information for a given set of homologous sequences extracted from a tree from a given phylome.

get_longest_isoform(*id*)
Returns the longest isoform for a given phylomeDB ID

get_new_phylomedb_id(*old_id*)
Return the conversion between an old phylomeDB ID and a new one

get_old_phylomedb_ids(*ids*)
Returns all old phylomeDB IDs associated to each of the input phylomeDB IDs

get_phylome_algs(*phylome_id*)
Returns all alignments available for a given phylome

get_phylome_info(*phylome_id*)
Returns available information on a given phylome

get_phylome_seed_ids(*phylome_id*, *filter_isoforms=True*)
Returns the seed phylomeDB IDs for a given phylome being possible to filter out the longest isoforms

get_phylome_seed_ids_info(*phylome_id*, *start=0*, *offset=None*, *filter_isoforms=False*)

get_phylome_trees(*phylome_id*)
Returns all trees available for a given phylome

get_phylomes()
Returns all current available phylomes

get_phylomes_for_seed_ids(*ids*)
Given a list of phylomeDB IDs, return in which phylomes these IDs have been used as a seed

get_prot_gene_names(*ids*)
Returns all possible protein and gene names associated to the input phylomeDB IDs

get_proteomes_in_phylome(*phylome_id*)
Returns a list of proteomes associated to a given phylome_id

get_raw_alg(*id*, *phylome_id*)
Return the raw alignment for the input phylomeDB ID in the given phylome

get_seq_info_in_tree(*id*, *phylome_id*, *method=None*)
Return all the available information for each sequence from tree/s associated to a tuple (protein, phylome) identifiers.

get_seq_info_msf(*id*, *phylome_id*)
Return all available information for the homologous sequences to the input phylomeDB ID in the input phylome using the best tree to compute the set of homologous sequences

get_seqid_info(*id*)
Returns available information about a given protid

get_seqs_in_genome(*taxid*, *version*, *filter_isoforms=True*)
Returns all sequences of a given proteome, filtering the

get_species()
Returns all current registered species in the database

get_species_in_phylome(*phylome_id*)
Returns a list of proteomes associated to a given phylome_id

get_species_info (*taxid=None*, *code=None*)
Returns all information on a given species/code

get_tree (*id*, *phylome_id*, *method=None*, *best_tree=False*)
Depending in the input parameters select either .- a tree with the best evolutionary model in terms of LK (best_tree) .- a tree reconstructed using a specific model (method) .- all available model/trees for the tuple (phylomeDB ID, phylome ID)

search_id (*id*)
Returns a list of the longest isoforms for each proteome where the ID is already registered. The ID can be a current phylomeDB ID version, former phylomeDB ID or an external ID.

4.8 Seqgroup class

class **SeqGroup** (*sequences=None*, *format='fasta'*)
Bases: `object`

SeqGroup class can be used to store a set of sequences (aligned or not).

Parameters

- **sequences** – Path to the file containing the sequences or, alternatively, the text string containing the same information.
- **format** (*fasta*) – the format in which sequences are encoded. Current supported formats are: `fasta`, `phylip` (phylip sequencial) and `iphylip` (phylip interleaved). Phylip format forces sequence names to a maximum of 10 chars. To avoid this effect, you can use the relaxed phylip format: `phylip_relaxed` and `iphylip_relaxed`.

```
msf = ">seq1\nAAAAAAAAAAA\n>seq2\nTTTTTTTTTTTTT\n"
seqs = SeqGroup(msf, format="fasta")
print seqs.get_seq("seq1")
```

get_entries ()
Returns the list of entries currently stored.

get_seq (*name*)
Returns the sequence associated to a given entry name.

iter_entries ()
Returns an iterator over all sequences in the collection. Each item is a tuple with the sequence name, sequence, and sequence comments

set_seq (*name*, *seq*, *comments=None*)
Updates or adds a sequence

write (*format='fasta'*, *outfile=None*)
Returns the text representation of the sequences in the supplied given format (default=FASTA). If "oufile" argument is used, the result is written into the given path.

New in version 2.1.

4.9 WebTreeApplication object

class **WebTreeApplication**

Provides a basic WSGI application object which can handle ETE tree visualization and interactions. Please, see the webplugin example provided with the ETE installation package (http://pypi.python.org/pypi/ete2).

register_action(*name*, *target*, *handler*, *checker*, *html_generator*)

Adds a new web interactive function associated to tree nodes.

set_default_layout_fn(*layout_fn*)

Fix the layout function used to render the tree.

set_external_app_handler(*handler*)

Sets a custom function that will extend current WSGI application.

set_external_tree_renderer(*handler*)

If the tree needs to be processed every time is going to be drawn, the task can be delegated.

set_tree_loader(*TreeConstructor*)

Delegate tree constructor. It allows to customize the Tree class used to create new tree instances.

set_tree_size(*w*, *h*, *units='px'*)

Fix the size of tree image

set_tree_style(*handler*)

Fix a `TreeStyle` instance to render tree images.

Frequently Asked Questions (FAQs)

Contents

5.1 General

5.1.1 How do I use ETE?

From 2.1 version, ETE includes a basic standalone program that can be used to quickly visualize your trees. Type `ete2` in a terminal to access the program. For instance:

```
# ete2 "((A,B),C);"
```

or

```
# ete2 mytreefile.nw
```

However, ETE is not a standalone program. The `ete2` script is a very simple implementation and does not allow for fancy customization. The main goal of ETE is to provide a Python programming library, so you can create your own scripts to manipulate and visualize phylogenetic trees. Many examples are available here and along with the ETE tutorial.

5.2 Tree Browsing

5.2.1 How do I find a leaf by its name?

You can use the `TreeNode.search_nodes()` function:

```
matching_nodes = tree.search_nodes(name="Tip1")
```

Or use the following shortcut (not that it assumes no duplicated names)

```
node = tree&"Tip1"
```

5.2.2 How do I visit all nodes within a tree?

There are many ways, but this is the easiest one:

```
for node in t.traverse():
    print node.name
```

5.2.3 Can I control the order in which nodes are visited?

Yes, currently 3 strategies are implemented: pre-order, post-order and level-over. You can check the differences at http://packages.python.org/ete2/tutorial/tutorial_trees.html#traversing-browsing-trees

5.2.4 What's the difference between `Tree.get_leaves()` and `Tree.iter_leaves()`?

All methods starting with `get_` (i.e. get_leaves, get_descendants, etc.) return an independent list of items. This means that tree traversing is fully performed before returning the result. In contrast, iter_ methods return one item at a time, saving memory and, increasing the performance of some operations.

Note also that tree topology cannot be modified while iterating methods are being executed. This limitation does not apply for get_ methods.

In addition, get_ methods can be used to cache tree browsing paths (the order in which nodes must be visited), so the same tree traversing operations don't need to be repeated:

```
nodes_in_preorder = tree.get_descendants("preorder")
for n in nodes_in_preorder:
    pass # Do something
#
# (...)
#
for n in nodes_in_preorder:
    pass # Do something else
```

5.3 Reading and writing tree

5.3.1 How do I load a tree with internal node names?

Newick format can be slightly different across programs. ETE allows to read and write several newick subformats, including support for internal node labeling:

FORMAT	DESCRIPTION	SAMPLE
0	flexible with support values	((D:0.723274,F:0.567784)1.000000:0.067192,(B:0.279326,H:0.756049)1.00
1	flexible with internal node names	((D:0.723274,F:0.567784)E:0.067192,(B:0.279326,H:0.756049)B:0.807788)
2	all branches + leaf names + internal supports	((D:0.723274,F:0.567784)1.000000:0.067192,(B:0.279326,H:0.756049)1.00
3	all branches + all names	((D:0.723274,F:0.567784)E:0.067192,(B:0.279326,H:0.756049)B:0.807788)
4	leaf branches + leaf names	((D:0.723274,F:0.567784),(B:0.279326,H:0.756049));
5	internal and leaf branches + leaf names	((D:0.723274,F:0.567784):0.067192,(B:0.279326,H:0.756049):0.807788);
6	internal branches + leaf names	((D,F):0.067192,(B,H):0.807788);
7	leaf branches + all names	((D:0.723274,F:0.567784)E,(B:0.279326,H:0.756049)B);
8	all names	((D,F)E,(B,H)B);
9	leaf names	((D,F),(B,H));
100	topology only	((,),(,));

In order to load (or write) a tree with internal node names, you can specify format 1:

```
from ete2 import Tree
t = Tree("myTree.nw", format=1)

t.write(format=1)
```

5.3.2 How do I export tree node annotations using the Newick format?

You will need to use the extended newick format. To do so, you only need to specify the name of the node attributes that must be exported when calling tree.write() function. For instance:

```
tree.write(features=["name", "dist"])
```

If you want all node features to be exported in the newick string, use “features=[]”:

```
tree.write(features=[])
```

5.4 Tree visualization

5.4.1 Can ETE draw circular trees?

Yes, starting from version 2.1, ete can render trees in circular mode. Install the latest version from http://pypi.python.org/pypi/ete2 or by executing `easy_install -U ete2`.

5.4.2 What are all these dotted lines that appear in my circular trees?

Opposite to other popular visualization software, ETE's drawing engine will try by all means to avoid overlaps among lines and all other graphical elements. When faces are added to nodes (specially to internal nodes), the required space to allocate such elements requires to expand the branches of the tree. Instead of breaking the relative length of all branches, it will add dotted lines until reaching the its minimal position. This effect could only be avoided by increasing the branch scale. Alternatively, you can modify the aspect of the dotted lines using `TreeStyle` options, such as `extra_branch_line_type`.

As by Jun 2012, ETE 2.1 includes a patch that allows to automatically detect the optimal scale value that would avoid dotted lines. Two levels of optimization are available, see `optimal_scale_level` option in `TreeStyle` class. This feature is now user-transparent and enabled by default, so, if no scale is provided, the optimal one will be used.

5.4.3 Why some circular trees are too large?

In order to avoid overlaps among elements of the tree (i.e. node faces), ETE will expand branch lengths until the desired layout is fully satisfied.

5.4.4 How do I export tree images as SVG

Image format is automatically detected from the filename extension. The following code will automatically render the tree as a vector image.

```
tree.render("mytree.svg")
```

5.4.5 How do I visualize internal node names?

You will need to change the default tree layout. By creating your custom layout functions, you will be able to add, remove or modify almost any element of the tree image.

A basic example would read as follow:

```
from ete2 import Tree, faces, AttrFace, TreeStyle

def my_layout(node):
    if node.is_leaf():
         # If terminal node, draws its name
         name_faces = AttrFace("name")
    else:
         # If internal node, draws label with smaller font size
         name_faces = AttrFace("name", fsize=10)
    # Adds the name face to the image at the preferred position
    faces.add_face_to_node(name_face, node, column=0, position="branch-right")

ts = TreeStyle()
# Do not add leaf names automatically
ts.show_leaf_name = False
# Use my custom layout
ts.layout_fn = my_layout

t = Tree("((B,(E,(A,G)M1_t1)M_1_t2)M2_t3,(C,D)M2_t1)M2_t2;", format=8)
```

```
# Tell ETE to use your custom Tree Style
t.show(tree_style=ts)
```

5.4.6 Can the visualization of trees with very unbalanced tree branches be improved?

Yes, the experience of visualizing trees with extreme differences in branch lengths can be improved in several ways.

1) Convert your tree to ultrametric topology. This will modify all branches in your tree to make all nodes to end at the same length.

```
from ete2 import Tree, TreeStyle

t = Tree()
t.populate(50, random_branches=True)
t.convert_to_ultrametric()
t.show()
```

2) You can enable the `force_topology` option in `TreeStyle`, so all branches will be seen as the same length by the tree drawing engine (Note that in this case, actual tree branches are not modified)

```
from ete2 import Tree, TreeStyle

t = Tree()
t.populate(50, random_branches=True)
ts = TreeStyle()
ts.force_topology = True
t.show(tree_style=ts)
```

CHAPTER 6

Help and Support

Could not find an answer? Join the ETE toolkit community and post your question!!

You will also be updated with important news and announcements.

https://groups.google.com/forum/#!forum/etetoolkit

(etetoolkit@googlegroups.com)

CHAPTER 7

About ETE

ETE is currently developed by the Comparative genomics group at the Centre for Genomic Regulation

You can cite ETE as:

Jaime Huerta-Cepas, Joaquín Dopazo and Toni Gabaldón. **ETE: a python Environment for Tree Exploration.** *BMC Bioinformatics 2010, 11:24.*

7.1 People using ETE

Warning: UNDER CONSTRUCTION

genindex

modindex

Python Module Index

Index