Pylogeny Documentation

Release 0.3.6

Alex Safatli

April 23, 2015

Contents:

CHAPTER

ONE

PYLOGENY PACKAGE

1.1 Submodules

Serialize a phylogenetic landscape into a JSON object.

class pylogeny.JSONWriter.**JSONWriter**(*ls*, *name*)
Bases: pylogeny.landscapeWriter.landscapeWriter

Writes a landscape and associated node information to a JSON object.

__init__(*ls*, *name*)
Instantiates this writer.

Parameters

- **ls** (a landscape.landscape object) – a landscape object
- **name** (*a string*) – the name of this landscape

getCompleteLandscape()
Returns the landscape as a JSON string.

Returns a JSON string

getJSON()
Returns the landscape as a JSON string.

Returns a JSON string

getOnlyImprovements(*groups=None*)

nodeToJSON(*node*)
Returns a JSON formatted node, given a node ID.

Parameters **node** (*a string*) – a name of a tree/node in the graph

Object model defining a sequence alignment (DNA, RNA, protein sequences). Handle input biological sequence alignment files for the purposes of phylogenetic inference. Will read all types of alignment files by utilizing the P4 python phylogenetic library.

class pylogeny.alignment.**alignment**(*inal=None*)
Bases: object

Wrap a biological sequence alignment to enable functionality necessary for phylogenetic inference. Makes use of temporary files; requires to be closed once no longer needed.

__init__(*inal=None*)
Instantiate an object intended to wrap an alignment for the purposes of running phylogenetic inference.

Parameters **inal** – An alignment file path (most formats are accepted).

close()
Forcefully delete all temporary files and clear data.

getApproxMLNewick()
Get a tree in newick format via use of FastTree that serves as an approximation of the maximum likelihood tree for this data.

Returns a Newick or New Hampshire string

getApproxMLTree()
Get a tree object for an approximation of the maximum likelihood tree for this data using FastTree.

Returns a `tree.tree` object

getDataType()
Get the data type associated with this alignment (e.g., protein).

Returns a string indicating the data type ('protein', 'DNA')

getDim()
Return the dimensionality of the sequence alignment (how many different types of characters).

Returns an integer

getFASTA()
Get (and create if not already) a path to a temporary FASTA file. This will be deleted upon closure of the alignment instance.

Returns a string associated with a path in the file system

getNumSeqs()
Return the number of sequences that are present in the sequence alignment.

Returns an integer

getSequenceString(*i*)
Acquire the ith sequence as a string.

Parameters **i** (*an integer*) – an index in the alignment (associated with a sequence)

Returns a string associated with the sequence

getSize()
Return the size of the alignment, or how many characters there are in each respective item in the alignment.

Returns an integer

getStateModel()
Get the state model associated with this alignment. See model module for more information.

Returns a `model.DiscreteStateModel` object

getTaxa()
Get a list of taxa names associated with the alignment.

Returns a list of strings

toStrList()
Get all sequences as a list of strings.

Returns a list of strings

class pylogeny.alignment.**phylipFriendlyAlignment**(*inal=None*)
Bases: `pylogeny.alignment.alignment`

An alignment object that renames all comprising taxa in order to be able to be written as a strict Phylip file.

__init__(*inal=None*)

getPhylip()
Get (and create if not already) a path to a temporary Phylip file. This will be deleted upon closure of the alignment instance.

Returns a string associated with a path in the file system

getProperName(*n*)
Return the actual name for an integer-based sequence name that was reassigned at initialization.

Parameters **n** (*a string*) – a shortened taxon name from this object

Returns a string (replaced with the original taxon name)

getTaxa()
Return current taxa names in the alignment.

Returns a list of shortened taxa names

reassignFromReinterpretedNewick(*tr*)
Return a Newick string with taxa names replaced with shortened forms as they are defined in this object.

Parameters **tr** (*a string*) – a Newick string

Returns a Newick string with all replaced names

recreateObject()
Reintializes the object.

reinterpretNewick(*tr*)
Revert the replacing of taxa names with shortened names by changing them back to their original form.

Parameters **tr** (*a string*) – a Newick string

Returns a Newick string with all replaced names

writeProperNexus(*wri*)
Write a Nexus file with proper names.

Parameters **wri** (*a string*) – a path to a (existent or unexistent) file to write to

Definitions for generalized containers and objects used by other structures in this framework.

pylogeny.base.**longest_common_substring**(*s1*, *s2*)
Simplified, traditional LCS algorithm implementation.

Returns a string (longest common substring of s1, s2)

class pylogeny.base.**patriciaTree**
Bases: pylogeny.base.trie

Defines a PATRICIA tree (condensed trie) across a range of strings.

delete(*seq*)
Remove a sequence from the PATRICIA tree. Will not remove added characters to alphabet.

Parameters **seq** – a sequence present in the trie

insert(*seq*)
Dynamically insert a sequence into the PATRICIA tree. Returns the unique index in the tree for that string.

search(*seq*)
Search for a sequence in the PATRICIA tree. Returns its position in addition sequence if it exists. Else, returns 0.

class pylogeny.base.**treeBranch**(*parent=None*, *child=None*, *label=''*)
Bases: object

A branch in a tree.

__init__(*parent=None*, *child=None*, *label=''*)
Instantiate this branch.

Parameters

- **parent** (a treeNode object) – an optional parent node
- **child** (a treeNode object) – an optional child node
- **label** (*a string*) – an optional string label

getChild()
Return the child node of this branch.

Returns a treeNode object

getLabel()
Return the label of this branch.

Returns a string

getParent()
Return the parent node of this branch.

Returns a treeNode object

setChild(*c*)
Set the child node of this branch.

Parameters **c** (a treeNode object) – the child node of this object

setLabel(*lbl*)
Set the label of this branch.

Parameters **lbl** (*a string*) – a string label

setParent(*p*)
Set the parent node of this branch.

Parameters **p** (a treeNode object) – the parent node of this object

class pylogeny.base.**treeNode**(*lbl=None*, *children=None*, *parent=None*)
Bases: object

A node in a tree.

__init__(*lbl=None*, *children=None*, *parent=None*)
Initialize this tree node.

Parameters

- **lbl** (*a string*) – an optional string to label this node
- **children** (a list of treeBranch objects) – an optional list of branches as children
- **parent** (a treeBranch object) – an optional branch to act as parent to this one

addChild(*item*)
Add a branch as a child.

Parameters **item** (a treeBranch object) – a branch to add as a child

getChildByIndex(*i*)
Get a child branch by index in the list of children.

Parameters **i** (*an integer*) – an index

Returns a `treeBranch` object

getChildren()
Return the list of children branch object.

Returns a string

getLabel()
Return the label of this node.

Returns a string

getParent()
Return the parent of this node.

Returns a `treeBranch` object

isInternalNode()
Determine if this node is not a leaf (has children).

Returns a boolean

isLeaf()
Determine if this node is a leaf (has no children).

Returns a boolean

class `pylogeny.base.`**treeStructure**(*root=None*)
Bases: `_abcoll.Container`

Defines a base collection of treeNodes and treeBranches in a rooted, hierarchical tree structure.

__init__(*root=None*)
Initialize this tree structure.

Parameters **root** (a `treeNode` object) – an optional node to root the tree with

getAllLeaves()
Acquire all leaf nodes for this structure.

Returns a list of `treeNode` objects

getAllNodes()
Acquire all nodes for this structure.

Returns a list of `treeNode` objects

getPostOrderTraversal()

getRoot()
Return the top-level, root, node of the tree.

Returns a `treeNode` object

static leaves(*root*)
Static method to acquire all leaf nodes of a tree structure in order of how they are defined in children of nodes (DFS).

Parameters **root** (a `treeNode` object) – a root node of a tree structure

Returns a list of `treeNode` objects

static nodes(*root*)
Static method to acquire all nodes of a tree structure in order of how they are defined in children of nodes (DFS).

Parameters **root** (a treeNode object) – a root node of a tree structure

Returns a list of treeNode objects

static postOrderTraversal(*root*)
Static method to acquire all nodes of a tree structure as a post order traversal.

Parameters **root** (a treeNode object) – a root node of a tree structure

Returns a list of treeNode objects

class pylogeny.base.**trie**
Bases: _abcoll.Sized, pylogeny.base.treeStructure

Defines a trie across a range of strings.

__init__()
Instantiate this trie as empty.

delete(*seq*)
Remove a sequence from the trie. Will not remove added characters to alphabet.

Parameters **seq** – a sequence present in the trie

getAlphabet()
Acquire the unique alphabet of characters present across strings in this trie.

Returns a list of characters

getRoot()
Get the root node of this trie.

Returns a trieNode object

insert(*seq*)
Dynamically insert a sequence into the trie.

Returns the label for this inserted sequence

search(*seq*)
Search for a sequence in the trie. Returns true if it exists.

Returns a boolean

class pylogeny.base.**trieNode**(*lbl=None*, *children=None*, *parent=None*)
Bases: pylogeny.base.treeNode

A subclass of treeNode that allows for checking non-zero members amongst children branches and other conveniences.

getNonEmptyChildrenBranchLabels()
Acquire a list of labels for all non-empty children branches.

Returns a list of strings

getNonEmptyChildrenBranches()
Acquire a list of all non-empty children branches.

Returns a list of treeBranch objects

getNonEmptyChildrenNodes()
Acquire a list of all non-empty children nodes.

Returns a list of `treeNode` objects

getParentNode()
Get the parent node of this node (assumes a parent branch).

Returns the parent of the parent branch to this node

iterNonEmptyChildrenNodes()
Iterate over all children nodes that are not empty.

Returns a generator yielding children `treeNode` objects

numEmptyChildrenNodes()
Acquire the number of children nodes that are marked 0 or nonexistent.

setChildNode(*child*, *newchild*)
Set a given child node (traversing branches along the way) of this node to a new object.

Parameters

- **child** (a `treeNode` object) – a child node of this object
- **newchild** (a `treeNode` object) – the child node to replace

Connect, access, + manipulate external tree data from a remote SQL server or from an sqlite file.

class `pylogeny.database.`**DatabaseLandscape**(*ali*, *starting_tree=None*, *root=True*, *operator='SPR'*)
Bases: `pylogeny.landscape.landscape`

Abstract the landscape to one comprising a database.

getNode(*i*)

class `pylogeny.database.`**SQLDatabase**(*host*, *user*, *pw*, *db*)
Bases: `pylogeny.database.database`

Database object to allow reading from a MySQL database.

__init__(*host*, *user*, *pw*, *db*)

close()

connect()

getColumns(*table*)
Return column information for a given table.

getTables()

query(*q*)

querymany(*q*, *i*)

class `pylogeny.database.`**SQLExhaustiveLandscape**(*dbobj*, *aliname*)
Bases: `pylogeny.database.DatabaseLandscape`

Abstract the landscape to one comprising an SQL database.

__init__(*dbobj*, *aliname*)
Instantiate this landscape.

Parameters

- **dbobj** (a `database`) – a database object
- **aliname** (*a string*) – the name of the alignment (table in the database)

exploreRandomTree(*i*)

exploreTree(*i*)

getDatabaseNode(*i*)

class pylogeny.database.**SQLiteDatabase**(*filepath*)
Bases: pylogeny.database.database

__init__(*filepath*)
Instantiate this SQLite database object.

Parameters **filepath** (*a string*) – a path to the file

close()

getColumns(*table*)
Return column information for a given table.

Parameters **table** (*a string*) – a table name

getTables()

query(*q*)

querymany(*q*, *i*)

class pylogeny.database.**SQLiteLandscape**(*dbobj*)
Bases: pylogeny.landscape.landscape

Allow random access of all landscape data from an sqlite file found on the hard disk.

__init__(*dbobj*)

class pylogeny.database.**database**
Bases: object

Allow interfacing with a SQL/sqlite database.

__init__()

close()

filterRecords(*table*, *condn*)
Get all records from a given table following a condition. The equivalent of calling "SELECT * FROM table WHERE cond".

Parameters

- **table** (*a string*) – a table name
- **condn** (*a string*) – a condition in SQL syntax

Returns a list of tuples

getColumns(*table*)

getHeaders(*table*)
Get only header names for a given table's columns.

Parameters **table** (*a string*) – a table name

getRecords(*table*)
Get all records from a given table in the database. The equivalent of a call "SELECT * FROM table".

Parameters **table** (*a string*) – a table name

Returns a list of tuples

getRecordsAsDict (*table*)
Acquires records using getRecords() and then leverages access using a dictionary data structure where keys are headers (column names).

Parameters **table** (*a string*) – a table name

Returns a dictionary (of records as values)

getRecordsColumn (*table*, *col*)
Get all data for a single column from records for a table. The equivalent of a call "SELECT col FROM table".

Parameters

- **table** (*a string*) – a table name
- **col** (*a string*) – a column name

Returns a list of strings

getTables ()

insertRecord (*tablename*, *record*)
Insert a single record.

Parameters

- **tablename** (*a string*) – the name of the table
- **record** (*a tuple*) – a tuple

insertRecords (*tablename*, *items*)
Insert a number of records into a table.

Parameters

- **tablename** (*a string*) – the name of the table
- **items** (*a list of tuples*) – a list of record tuples

isEmpty ()
Determine if the database is empty.

Returns a boolean

iterRecords (*table*)
Get a record, one at a time, from a table in the database.

Parameters **table** (*a string*) – a table name

Returns a generator of records

newTable (*tablename*, **args*)
Create a new table.

Parameters **tablename** (*a string*) – the name of this table

query (*q*)

querymany (*q*, *i*)

Defines an interface to manage interfacing with the system for respective application calls and implements some of these for executables such as FastTree and RAxML. Currently requires a UNIX-like environment (e.g., Mac OS X or a Linux-based environment).

class pylogeny.executable.**aTemporaryDirectory**(*dir=None*)
Bases: object

A class intended to be used as a context manager that allows Python to run in a temporary directory for a finite period of time.

__init__(*dir=None*)

class pylogeny.executable.**consel**(*treeset*, *alignment*, *name*)
Bases: pylogeny.executable.executable

Denotes a single run of the CONSEL workflow in order to acquire a confidence interval and perform an AU test on a set of trees. Requires CONSEL to be installed.

__init__(*treeset*, *alignment*, *name*)

getInstructionString()
Get the instruction string.

Returns a string (of a UNIX command)

getInterval()
Compute the AU test. Return the interval of trees.

Returns a list of tree.tree objects

pylogeny.executable.**exeExists**(*cmd*)
Determines whether a function exists in a UNIX environment.

class pylogeny.executable.**executable**
Bases: object

An abstract class for the instantation and running of a single instance for a given application.

exeName = None

getInstructionString()

run()
Perform a run of this application.

class pylogeny.executable.**fasttree**(*inp_align*, *out_file=None*, *isProtein=True*)
Bases: pylogeny.executable.executable

Denotes a single run of the FastTree executable in order to acquire an approximate maximum likelihood tree for the input alignment. See http://www.microbesonline.org/fasttree/ for more information on FastTree. Requires FastTree to be installed.

__init__(*inp_align*, *out_file=None*, *isProtein=True*)

exeName = 'fasttree'

getInstructionString()

class pylogeny.executable.**raxml**(*inp_align*, *out_file*, *model=None*, *is_Protein=True*, *interTrees=False*, *alg=None*, *startingTree=None*, *rapid=False*, *slow=False*, *optimizeBootstrap=False*, *numboot=100*, *log=None*, *wdir=None*)
Bases: pylogeny.executable.executable

Denotes a single run of the RAxML executable. See http://sco.h-its.org/exelixis/software.html for more information on RAxML. Requires RAxML to be installed.

__init__(*inp_align*, *out_file*, *model=None*, *is_Protein=True*, *interTrees=False*, *alg=None*, *startingTree=None*, *rapid=False*, *slow=False*, *optimizeBootstrap=False*, *numboot=100*, *log=None*, *wdir=None*)

exeName = 'raxmlHPC'

getInstructionString()

runFunction(*alg*)

class pylogeny.executable.**rspr**(*treeA*, *treeB*, *algorithm=''*, *overlap=True*)
Bases: pylogeny.executable.executable

Denotes a single run of the rSPR executable by Dr. Chris Whidden (2014), a software package for computing rooted subtree-prune-and-regraft (SPR) distances. See http://kiwi.cs.dal.ca/Software/RSPR. Requires the executable to be on PATH.

RSPR_ALG_APPROX = '-approx'

RSPR_ALG_BB = '-bb'

RSPR_ALG_DEFAULT = ''

RSPR_ALG_FPT = '-fpt'

__init__(*treeA*, *treeB*, *algorithm=''*, *overlap=True*)
Algorithm choices are defined in this class. If overlap is set to True, will attempt to consolidate taxa names such that they are overlapping (otherwise, RSPR will return an error if they do not match).

exeName = 'rspr'

getInstructionString()

getSPRDistance()

class pylogeny.executable.**treepuzzle**(*ali*, *treefile*)
Bases: pylogeny.executable.executable

Wrap TREE-PUZZLE in order to create an intermediate file for CONSEL to read and assign confidence to a set of trees. Requires TREE-PUZZLE to be installed.

__init__(*ali*, *treefile*)

exeName = 'puzzle'

getInstructionString()

getSiteLikelihoodFile()

Define the interface for a heuristic in order to implement any manner of heuristic for a combinatorial problem that can be abstracted into a state graph. In this case, a phylogenetic tree space.

class pylogeny.heuristic.**RAxMLIdentify**(*ls*, *startNode*, *workdir='.rxml'*)
Bases: pylogeny.heuristic.phylogeneticLinearHeuristic

RAxML-driven landscape evaluation of intermediate checkpoint trees output from the RAxML executable.

__init__(*ls*, *startNode*, *workdir='.rxml'*)
Initialize this heuristic.

Parameters

- **ls** (a landscape.landscape object) – a landscape object
- **startNode** (*a node (dictionary) from the landscape (getNode())*) – what node to start with

explore()
Explore using RAxML.

Returns None; landscape is modified.

class pylogeny.heuristic.**heuristic**(*G=None*, *start=None*)
Bases: object

A base interface for a heuristic that explores a state graph.

__init__(*G=None*, *start=None*)

explore()

getStartState()

getStateGraph()

class pylogeny.heuristic.**likelihoodGreedy**(*ls*, *startNode*)
Bases: pylogeny.heuristic.phylogeneticLinearHeuristic

Greedy (hill-climbing) landscape exploration by comparsion of likelihood.

__init__(*ls*, *startNode*)
Initialize this heuristic.

Parameters

- **ls** (a landscape.landscape object) – a landscape object
- **startNode** (*a node (dictionary) from the landscape (getNode())*) – what node to start with

explore()
Perform greedy search of the landscape using a method of greed via likelihood.

Returns None; landscape is modified

class pylogeny.heuristic.**parsimonyGreedy**(*ls*, *startNode*)
Bases: pylogeny.heuristic.phylogeneticLinearHeuristic

Greedy (hill-climbing) landscape exploration by comparsion of parsimony.

__init__(*ls*, *startNode*)
Initialize this heuristic.

Parameters

- **ls** (a landscape.landscape object) – a landscape object
- **startNode** (*a node (dictionary) from the landscape (getNode())*) – what node to start with

explore()
Perform greedy search of the landscape using a method of greed via parsimonious criterion.

Returns None; landscape is modified

class pylogeny.heuristic.**phylogeneticLinearHeuristic**(*ls*, *startNode*)
Bases: pylogeny.heuristic.heuristic

A base class for a heuristic that works on a phylogenetic landscape and only possesses a single path (of search).

__init__(*ls*, *startNode*)

bestTree = None

getBestTree()

getPath()

path = []

class pylogeny.heuristic.**smoothGreedy**(*ls*, *startNode*)
Bases: pylogeny.heuristic.phylogeneticLinearHeuristic

Parsimony-driven greedy landscape exploration by comparsion of likelihoods.

__init__(*ls*, *startNode*)
Initialize this heuristic.

Parameters

- **ls** (a landscape.landscape object) – a landscape object
- **startNode** (*a node (dictionary) from the landscape (getNode())*) – what node to start with

explore()
Perform greedy search of the landscape using a method of greed via parsimonious criterion and then performing final smoothing via likelihood on top 10% of 1-SPR neighbors ranked on basis of parsimony.

Returns None; landscape is modified

Encapsulate a phylogenetic tree space. A phylogenetic landscape or tree space refers to the entire combinatorial space comprising all possible phylogenetic tree topologies for a set of n taxa. The landscape of n taxa can be defined as consisting of a finite set T of tree topologies. Tree topologies can be associated with a fitness function f(t_i) describing their fit. This forms a discrete solution search space and finite graph (T, E) = G. E(G) refers to the neighborhood relation on T(G). Edges in this graph are bidirectional and represent transformation from one tree topology to another by a tree rearrangement operator. An edge between t_i and t_j would be notated as e_{ij} in E(G).

class pylogeny.landscape.**graph**(*gr=None*, *defWeight=0.0*)
Bases: object

Define an empty graph object.

__init__(*gr=None*, *defWeight=0.0*)
Instantiate a graph. Default edge weights are 0.

Parameters

- **gr** (a networkx.Graph object) – a networkx graph object, if already exists.
- **defWeight** (*a floating point number*) – the default edge weight of weights

clearEdgeWeights()
Set all edge weights to the default edge weight.

getCenter()
Get the centre of the graph.

getCliqueNumber()
Get the clique number of the graph.

Returns an integer

getCliques()
Get the cliques present in the graph.

getCliquesOfNode(*i*)
Get the clique that a node corresponds to.

getComponentOfNode(*i*)
Get the graph component of a given node.

getComponents()
Get the connected components in the graph.

getDegreeFor(*i*)
Return in- and out-degree for node named i.

Parameters **i** (*a string*) – a node name

Returns an integer

getDiameter()
Acquire the diameter of the graph.

getEdge(*i*, *j*)
Get the data associated with an edge (including weight).

Parameters

- **i** (*a string*) – a node name
- **j** (*a string*) – a node name

Returns an edge (and associated data)

getEdges()
Get all edges (as defined for NetworkX graphs). Recommended to use an iterator for large graphs.

Returns a list of edges (and associated data)

getEdgesFor(*i*)
Get all edges associated with a certain node.

Parameters **i** (*a string*) – a node name

Returns a list of edges (and associated data) for all neighbors

getMST()
Acquire the minimum spanning tree for the graph.

getNeighborsFor(*i*)
Get a list of all node names neighbor to a node.

Parameters **i** (*a string*) – a node name

Returns a list of strings (node names)

getNode(*i*)
Get a single node by name.

Parameters **i** (*a string*) – a node name:

Returns a dictionary (with node information)

getNodeNames()
Return the names of nodes in the graph.

Returns a list of strings

getNodes()
Get all node values. Recommended to use an iterator for large graphs (iterNodes()).

Returns a list of node values (whatever is associated with nodes)

getNumCliques()
Get the number of cliques found in the graph.

Returns an integer

getNumComponents()
Get the number of components of the graph.

Returns an integer

getShortestPath(*nodA*, *nodB*)
Get the shortest path between two nodes.

getShortestPathLength(*nodA*, *nodB*)
Get the shortest path length between two nodes.

getSize()
Return the number of nodes in the graph.

Returns an integer

hasPath(*nodA*, *nodB*)
See if a path exists between two nodes.

isEdge(*i*, *j*)
See if an edge exists between two nodes.

Parameters

- **i** (*a string*) – a node name
- **j** (*a string*) – a node name

Returns a boolean

iterNodes()
Iterate over all node keys.

Returns a generator of node keys

setDefaultWeight(*w*)
Set the default weight of edges (weight of edges if not overridden).

Parameters **w** (*a floating point*) – a weight

class pylogeny.landscape.**landscape**(*ali*, *starting_tree=None*, *root=True*, *operator='SPR'*)
Bases: pylogeny.landscape.graph, pylogeny.tree.treeSet

Defines an entire phylogenetic tree space.

__init__(*ali*, *starting_tree=None*, *root=True*, *operator='SPR'*)
Initialize the landscape.

Parameters

- **ali** (an alignment.alignment object) – an alignment
- **starting_tree** (a tree.tree object) – an optional tree object to start with
- **root** (*a boolean*) – whether or not to compute an approximate maximum likelihood tree (FastTree) or start the landscape with a given starting tree.
- **operator** (*a string*) – a string that describes what operator the landscape is mostly comprised of.

addTree(*tr*, *score=True*, *check=True*, *newick=None*, *struct=None*)
Add a tree to the landscape. Will return its index.

Parameters

- **tr** (a tree.tree object) – a tree
- **score** (*a boolean*) – defaults to True, whether to score this tree or not

Returns the index of the tree

addTreeByNewick (*newick*, *score=True*, *check=True*, *struct=None*)
Add tree to the landscape by Newick string. Will return index.

Parameters

- **newick** (*a string*) – a Newick string
- **score** (*a boolean*) – defaults to True, whether to score this tree or not

Returns the index of the tree

exploreRandomTree (*i*, *type=1*)
Acquire a single neighbor to a tree in the landscape by performing a random rearrangement of type SPR (by default), NNI, or TBR – this is done by performing a rearrangement on a random branch in the topology. Rearrangement type is provided as a rearrangement module type definition of form, for example, TYPE_SPR, TYPE_NNI, etc.

Parameters

- **i** – a tree index
- **type** – the type of rearrangement (e.g., TYPE_SPR, TYPE_NNI)

Returns the new tree index or None in case of failure

exploreTree (*i*, *type=1*)
Get all neighbors to a tree named i in the landscape using a respective rearrangement operator as defined in the rearrangement module. Rearrangement type is provided as a rearrangement module type definition of form, for example, TYPE_SPR, TYPE_NNI, etc. By default, this is TYPE_SPR.

Parameters

- **i** – a tree index
- **type** – the type of rearrangement (e.g., TYPE_SPR, TYPE_NNI)

Returns a list of neighbors as tree names (usually integers)

findTree (*newick*)
Find a tree by Newick string, taking into account branch lengths. Returns the index of this tree in the landscape. Warning: naively performs a sequential search.

Parameters **newick** (*a string*) – a Newick string

Returns a tree name (usually an integer index) or None if not found

findTreeTopology (*newick*)
Find a tree by topology, not taking into account branch lengths.

Parameters **newick** (*a string*) – a Newick string

Returns a tree name (usually an integer index) or None if not found

findTreeTopologyByStructure (*struct*)
Find a tree by topology, not taking into account branch lengths, given the topology.

Parameters **struct** (*a string*) – a Newick string without branch lengths (a "structure")

Returns a tree name (usually an integer index) or None if not found

getAlignment ()
Acquire the alignment object associated with this space.

Returns an `alignment.alignment` object

getAllPathsOfBestImprovement ()
Return all paths of best improvement as a dictionary.

Returns a dictionary of tree name to paths (lists of tree names)

getBestImprovement(*i*)
For a tree in the landscape, investigate neighbors to find a tree that leads to the best improvement of fitness function score on the basis of likelihood.

Parameters **i** – a tree name (usually an integer)

Returns a tree name (usually an integer) or None if no better tree

getBipartitionFoundInTreeByIndex(*tr*, *brind*, *topol=None*)
Given a tree node and a branch index, return the associated bipartition.

Parameters

- **tr** – a tree name
- **brind** – a branch index in that tree, a la post-order traversal

Returns a `tree.bipartition` object

getGlobalOptimum()
Get the global optimum of the current space.

Returns a tree name (usually an integer)

getLocalOptima()
Get all trees in the landscape that can be labelled as a local optimum.

Returns a list of tree names (usually integers)

getLocks()
Get all restrictions (locks which cannot be violated on splits).

Returns a list of `tree.bipartition` objects

getNumberTaxa()
Return the number of different taxa present in any respective tree in the landscape.

Returns an integer

getPathOfBestImprovement(*i*)
For a tree in the landscape, investigate neighbors iteratively until a best path of score improvement is found on basis of likelihood.

Parameters **i** – a tree name (usually an integer)

Returns a list of tree names (usually integers)

getPossibleNumberRootedTrees()
Assuming all of the trees in the space are rooted, return the maximum possible number of unrooted trees that can possibly be generated for the number of taxa of trees in the landscape.

Returns an integer

getPossibleNumberUnrootedTrees()
Assuming all of the trees in the space are unrooted, return the maximum possible number of unrooted trees that can possibly be generated for the number of taxa of trees in the landscape.

Returns an integer

getRoot()
Returns the index to the root (starting) tree of the space.

Returns an integer

getRootNode()
Returns the root (starting) tree of the space in its node form.

Returns a dictionary (with node information)

getRootTree()
Acquire the first tree that was placed in this space.

Returns a `tree.tree` object

getTree(*i*)
Get the object for a tree by its name.

Parameters **i** – a tree name (usually an integer)

Returns a `tree.tree` object

getVertex(*i*)
Acquire a vertex object from the landscape; this is a high-level representation of a tree in the landscape with additional functionality. Object created upon invocation of this function.

Parameters **i** – a tree name (usually an integer)

Returns a `vertex` object

indexOf(*tr*)
Acquire the index/name in this landscape of a tree object. Returns -1 if not found. Warning: naively performs a sequential search.

Parameters **tr** (a `tree.tree` object) – a tree

Returns a tree name (usually an integer index) or -1 if not found

isLocalOptimum(*i*)
Determine if a tree is a local optimum. This means it has the following properties:

1. Possesses a likelihood score.
2. Local neighborhood completely enumerated (and scored).
3. None of its neighbors is a better improvement.

Returns a boolean

isViolating(*i*)
Determine if a tree is violating any locks intrinsic to the landscape. Will also return False if the tree (name) is not present in the landscape.

Parameters **i** – a tree name (usually an integer)

iterAllPathsOfBestImprovement()
Return an iterator for all paths of best improvement.

Returns a generator of paths (lists of tree names)

lockBranchFoundInTree(*tr*, *br*)
Given a tree node and a branch object, add a given bipartition to the bipartition lock list. Returns bipartition if locked.

Parameters

- **tr** – a tree name
- **br** (a `base.treeBranch` object) – a branch in that tree

Returns a `tree.bipartition` object that has been locked or None

lockBranchFoundInTreeByIndex(*tr*, *brind*)
Given a tree node and a branch index, add an associated bipartition to the bipartition lock list. Returns the bipartition if locked.

Parameters

- **tr** – a tree name
- **brind** – a branch index

Returns a `tree.bipartition` object if it has been locked

removeTree(*tree*)
Remove a tree from the landscape by object.

Parameters **tree** (a `tree.tree` object) – a tree that exists in the landscape

Returns a boolean (success or failure)

removeTreeByIndex(*i*)
Remove a tree from the landscape by index.

Parameters **i** – a tree name (usually an integer)

Returns a boolean (success or failure)

setAlignment(*ali*)
Set the alignment present in this landscape. WARNING; will not modify existing scores.

Parameters **ali** (an `alignment.alignment` object) – an alignment

setOperator(*op*)
Set the operator assigned to this landscape.

Parameters **op** (*a string*) – an operator (string description)

toProperNewickTreeSet()
Convert this landscape into an unorganized set of trees where taxa names are transformed to their original form (i.e. not transformed to a state friendly for the Phylip format).

Returns a `tree.treeSet` object

toTreeSet()
Convert this landscape into an unorganized set of trees.

Returns a `tree.treeSet` object

toggleLock(*lock*)
Add a biparition to the list of locked bipartitions if not present; otherwise, remove it. Return status of lock.

Parameters **lock** (a `tree.bipartition` object) – a bipartition that cannot be violated

Returns a boolean (on or off)

class `pylogeny.landscape.`**vertex**(*obj*, *ls*)
Bases: `object`

Encapsulate a single vertex in the landscape and add convenient functionality to alias parent landscape functions.

__init__(*obj*, *ls*)
Initialize this vertex.

approximatePossibleNumNeighbors()
Approximate the possible number of neighbors to this vertex by considering the type of tree rearrangement operator. Returns LS_NOT_DEFINED if the operator is not known yet.

Returns an integer

getBestImprovement()
Alias function for function of same name in parent landscape.

getBipartitionScores()
Get all corresponding bipartition vectors of SPR scores.

getBipartitions()
Get all bipartitions for this vertex.

Returns a list of `tree.bipartition` objects

getDegree()
Get the degree of this tree in the graph.

Returns an integer

getDict()
Get the dictionary object (key-value pairs) associated with this tree as it is in the NetworkX graph.

Returns a dictionary

getIndex()
Get the index of this tree in the space.

Returns a tree name (usually an integer)

getNeighbors()
Get any neighbors to this tree in the landscape.

Returns a list of tree names (usually integers)

getNeighborsOfBipartition(*bi*)
Get corresponding neighbors of a bipartition in this vertex's tree.

getNeighborsOfBranch(*br*)
Get corresponding neighbors of a branch in this vertex's tree.

getNewick()
Get the Newick string of this tree.

Returns a string

getObject()
Get the dictionary object (key-value pairs) associated with this tree as it is in the NetworkX graph.

Returns a dictionary

getOrigin()
Get the origin of this tree (how it was acquired).

Returns a string

getPathOfBestImprovement()
Alias function for function of same name in parent landscape.

getProperNewick()
Get the proper Newick string for a tree. :returns: A string.

getScore()
Get (any) score(s) associated with this tree.

Returns a tuple of floating point values (scores)

getTree()
Get the tree object associated with this tree.

Returns a tree.tree object

isBestImprovement()
Check to see if this vertex is a best move for another node.

Returns a boolean

isExplored()
See if this tree has had all possible rearrangements performed.

Returns a boolean

isFailed()
Determine if any errors are associated with this node.

Returns a boolean

isLocalOptimum()
Determine if this tree is an optimum.

Returns a boolean

isViolating()
Alias function for function of same name in parent landscape.

iterBipartitions()
Return a generator to iterate over all bipartitions for this vertex.

scoreLikelihood()
Acquire the log-likelihood for this vertex.

Returns the log-likelihood score

setExplored(*exp*)
Override the "explored" flag of this node in the landscape.

Parameters **exp** – a boolean

Serialize a phylogenetic landscape into an SQLlite database file made up of three components: all tree IDs and respective scores, the alignment file as a set of sequences, and a representation of the graph as an edge list.

class pylogeny.landscapeWriter.**landscapeParser**(*path*)
Bases: object

Encapsulates the construction of a landscape object from a sqlite landscape file.

__init__(*path*)
Instantiate this parser.

Parameters **path** (*a string*) – the filepath to the landscape file

getName()
Acquire the name of the parsed landscape.

Returns a string

parse()
Parse the file.

Returns a tuple of a landscape.landscape object and its name (a string)

class pylogeny.landscapeWriter.**landscapeWriter**(*landscape*, *name*)
Bases: object

Encapsulate the writing of a landscape to a file format.

__init__(*landscape*, *name*)
Instantiates this writer.

Parameters

- **landscape** (a landscape.landscape object) – a landscape object
- **name** (*a string*) – the name of this landscape

writeFile(*path='.'*)
Write the landscape serialized file to given path.

Parameters **path** (*a string*) – a directory path, defaulting to the current one

Returns the relative filepath to the written file

Phylogenetic tree scoring models; intended to be coupled with the use of pytbeaglehon (BEAGLE) high-performance library.

class pylogeny.model.**DiscreteStateModel**(*alignment*)
Bases: object

Initialize a discrete state model for phylogenetic data. State frequencies and character time are determined from the given alignment object.

__init__(*alignment*)

getAlignment()

getAlignmentAsStateList()

getCharType()

getFrequencyOfState(*i*)

getRawFrequencyOfState(*i*)

getRawStateFreqs()

getRawStateFreqsAsDict()

getRawStateFreqsAsList()

getSequenceMatrix()

getStateFreqs()

exception pylogeny.model.**PhyloModelError**(*v*)
Bases: exceptions.Exception

__init__(*v*)

Newick string parsing and object interaction. A Newick string can represent a phylogenetic tree.

exception pylogeny.newick.**ParsingError**(*val*)
Bases: exceptions.Exception

__init__(*val*)

pylogeny.newick.**assignParents**(*top*)
Should be a one-time use function. Goes through and assigns parents to the parsed newick tree structure nodes and branches to allow for up-traversal.

Parameters **top** (a node object) – a top-level node for a tree (root node)

class pylogeny.newick.**branch**(*chi*, *l*, *parent=None*, *s=None*)
Bases: pylogeny.base.treeBranch

Branch for a tree parsed from a Newick string.

__init__(*chi*, *l*, *parent=None*, *s=None*)
Initialize a branch in a tree parsed from a Newick string.

Parameters

- **chi** (a node object) – a child node
- **l** (*a floating point value*) – a branch length
- **parent** (a node object) – an optional parent node; default none

pylogeny.newick.**getAllBranches**(*br*)
Given a branch, traverse subtree and return comprising branches as a list.

Parameters **br** (a branch object) – a branch from a tree

pylogeny.newick.**getBalancingBracket**(*newick*, *i*)
Given a position of an opening bracket in a newick string, i, output the closing bracket's position that corresponds to this opening bracket.

Parameters

- **newick** (*a string*) – a Newick string
- **i** (*an integer < length of the string*) – a position in the string (index)

Returns an integer

pylogeny.newick.**getBranchLength**(*newick*, *i*)
Given a position of a colon symbol (indicating a branch length), return the branch length.

Parameters

- **newick** (*a string*) – a Newick string
- **i** (*an integer < length of the string*) – a position in the string (index)

Returns an integer

pylogeny.newick.**getLeafName**(*newick*, *i*)
Given the position of a leaf, find its complete name.

Parameters

- **newick** (*a string*) – a Newick string
- **i** (*an integer < length of the string*) – a position in the string (index)

Returns an integer

pylogeny.newick.**invertAlongPathToNode**(*target*, *top*)
DANGEROUS: Reverses all directionality to a given node from a top-level node. Intended as a low-level function for rerooting a tree.

Parameters

- **target** (a node object) – a target node
- **top** (a node object) – a top-level node for a tree (root node)

pylogeny.newick.**isSibling**(*br*, *other*)
Given a branch, determine if that branch is adjacent to another branch.

Parameters

- **br** (a branch object) – a branch from a tree
- **other** (a branch object) – another branch from a tree

class pylogeny.newick.**newickParser**(*newick*)
Parsing object for Newick strings.

__init__(*newick*)
Initialize this parser (with a Newick string).

Parameters **newick** (*a string*) – a Newick string

parse()
Parse the stored newick string into a topological structure.

Returns the top-level root node object

class pylogeny.newick.**node**(*lbl=''*, *children=None*, *parent=None*)
Bases: pylogeny.base.treeNode

Node for a tree parsed from a Newick string.

__init__(*lbl=''*, *children=None*, *parent=None*)
Initialize a node in a tree parsed from a Newick string.

Parameters

- **lbl** (*a string*) – a label for this node
- **children** (a list of branch objects) – an optional set of children (branches); default none
- **parent** (a branch object) – an optional parent branch for this node; default none

pylogeny.newick.**parseNewick**(*newick*, *i*, *j*, *top*)
Parse a newick string into a topological newick structure given a top-level node.

Parameters

- **newick** (*a string*) – a Newick string
- **i** (*an integer*) – a starting position to start parsing
- **j** (*an integer*) – an end position to stop parsing
- **top** (a node object) – a top-level node; start parsing with None

pylogeny.newick.**removeBranchLengths**(*top*)
Goes through and removes any stored branch lengths.

Parameters **top** (a node object) – a top-level node for a tree (root node)

pylogeny.newick.**removeUnaryInternalNodes**(*top*)
Goes through and ensures any degree-2 internal nodes are smoothed into a single degree-3 internal node.

Parameters **top** (a node object) – a top-level node for a tree (root node)

pylogeny.newick.**shuffleLeaves**(*top*)
DANGEROUS: Given a top-level node, shuffle all leaves in this tree.

Parameters **top** (a node object) – a top-level node for a tree (root node)

Toolkit for performance of Parsimonious Criterion (Parsimony) methods of optimization of a phylogenetic topology with a particular set of data.

pylogeny.parsimony.**fitch**(*topology*, *alignment*)
Perform the Fitch algorithm on a given tree topology and associated alignment. Deprecated: Python implementation of the Fitch algorithm; see fitch C++ module for a C++ implementation that is roughly four times faster.

pylogeny.parsimony.**fitch_cost**(*topology*, *profiles*)
Calculate the cost using Fitch algorithm on profile set and alignment. Deprecated: Python implementation of the Fitch algorithm; see fitch C++ module for a C++ implementation that is roughly four times faster.

class pylogeny.parsimony.**profile_set**(*alignment*)
Hold a set of site_profile profiles for an entire alignment.

__init__(*alignment*)
Initialize this profile set by indicating an alignment.

Parameters **alignment** (an alignment.alignment object) – an alignment object

get(*val*)
Acquire the site profile at an index.

Parameters **val** (*an integer*) – an index of the set

Returns a site_profile object

getForTaxa(*val*, *tax*)
Acquire the string of sequence alphabet characters for a taxon.

Parameters

- **val** (*an integer*) – an index of the set
- **tax** (*a string*) – a taxon name

Returns a string of characters

weight(*val*)
Acquire the weight associated with an index.

Parameters **val** (*an integer*) – an index of the set

Returns a weight (integer)

class pylogeny.parsimony.**site_profile**(*alignment*, *site*)
Consolidate a single column of the alignment into a set of components with associated counts.

__init__(*alignment*, *site*)
Initialize this profile.

Parameters

- **alignment** (an *.alignment.alignment* object) – an alignment object
- **site** (*an integer*) – a site/column index along the alignment

Wrap C extension for libpll library for use in natural Python.

class pylogeny.pll.**dataModel**(*topo*, *alignm*, *model=None*)
Encapsulating a phylogenetic tree (as topology) + corresponding alignment into a libpll-associated data structure. Allows for log-likelihood scoring of this model. MUST BE CLOSED AFTER USE.

__init__(*topo*, *alignm*, *model=None*)
Initialize the data model and respective structures.

Parameters

- **topo** (rearrangement.topology) – a topology object

- **alignm** (`alignment.phylipFriendlyAlignment`) – a phylip-friendly alignment object.

close()
If done with this particular problem. Frees associated memory.

getLogLikelihood()
Calculates log-likelihood using libpll.

Returns a floating point value

getNewickString()
Acquire the Newick string of the problem instance.

Returns a Newick string

class pylogeny.pll.**partitionModel**(*ali*)
A partition model intended for libpll.

__init__(*ali*)
Initialize a partition model (for internal use by libpll).

Parameters **ali** (`alignment.alignment`) – an alignment object

close()
Delete file.

createModel(*models*, *partnames*, *ranges*)
Establish a more complex model.

Parameters

- **models** (*a list of strings*) – a list of model names (e.g., 'WAG', 'DNA')
- **partnames** (*a list of strings*) – a list of partition names (e.g., 'p1', 'p2')
- **ranges** (*a list of integer tuples*) – a list of range tuples (what ranges of alignment)

Returns None

createSimpleModel(*pmodel='WAG'*)
Establish a simple model (e.g., one type).

Parameters **pmodel** (*a string (default 'WAG')*) – optional; what protein model to use (as described in pll)

Returns None

getFileName()
Get the file name of the model file.

Returns a string

Phylogenetic tree structure encapsulation; allow rearrangement of said structure. Tree rearrangements inducing other topologies include Nearest Neighbor Interchange (NNI), Subtree Pruning and Regrafting (SPR), and Tree Bisection and Reconstruction (TBR). Each of these describe a transfer of one node in phylogenetic trees from one parent of a tree to a new parent. Respectively, these operators describe transformations that are subsets of those possible by the successive operator. For example, an NNI operator can perform transformations that are a subset of the transformations possible by the SPR operator.

exception pylogeny.rearrangement.**RearrangementError**(*val*)
Bases: exceptions.Exception

__init__(*val*)

pylogeny.rearrangement.**dup**(*topo*, *where=None*)

class pylogeny.rearrangement.**rearrangement**(*struct*, *type*, *targ*, *dest*)
Encapsulates a single rearrangement move of type SPR, NNI, ...

__init__(*struct*, *type*, *targ*, *dest*)
Initialize by providing a pointer to a base topology, a target branch to be moved, and its destination.

Parameters

- **struct** (a topology object) – a topology object
- **type** – the type of movement to perform
- **targ** – a target branch
- **dest** – a destination branch

doMove()
Commit the move and return the topology.

Returns a topology object

getType()
Get the type of movement.

Returns a string

isNNI()

isSPR()

isTBR()

toNewick()
Commit the move but do not create a new structure. Only retrieve resultant Newick string; will be more efficient.

Returns a Newick string

toTopology()
Commit the actual move and return the topology.

Returns a new topology object

toTree()
Commit the move and transform to tree object.

Returns a tree.tree object

class pylogeny.rearrangement.**topology**(*t=None*, *rerootToLeaf=True*, *toLeaf=None*)
Bases: pylogeny.base.treeStructure

Encapsulate a tree topology, wrapping the newick tree structure as a richer, rooted tree data structure object. Is immutable.

NNI(*branch*, *destination*)
Perform an NNI move of a branch to a destination, only if that destination branch is a parent's parent or a parent's sibling. Returns a rearrangement structure (not the actual new structure) that can then be polled for the actual move; this is in order to save memory.

SPR(*branch*, *destination*)
Perform an SPR move of a branch to a destination branch, creating a new node there. Returns a rearrangement structure (not the actual new structure) that can then be polled for the actual move; this is in order to save memory.

__init__(*t=None*, *rerootToLeaf=True*, *toLeaf=None*)
Initialize structure with a top-level internal node OR nothing.

Parameters

- **t** – a top-level internal node
- **rerootToLeaf** – whether to not reroot the structure to a lowest-lexicographic order taxon name
- **toLeaf** – reroot to a specifically provided leaf

allNNI()
Consider all valid NNI moves for a given topology and return all possible rearrangements.

allNNIForBranch(*br*, *flip=True*)
Consider all valid NNI moves for a given branch in the topology and return all possible rearrangements.

allSPR()
Consider all valid SPR moves for a given topology and return all possible rearrangements.

allSPRForBranch(*br*, *flip=True*)
Consider all valid SPR moves for a given branch in the topology and return all possible rearrangements.

allType(*type=1*)
Consider all valid moves of a given rearrangement operator for a given topology. Uses a given rearrangement operator type defined in this module. For example, calling this function by providing TYPE_NNI as the type will iterate over all NNI operations. By default, the type is TYPE_SPR.

fromNewick(*newickstr*)
Alias for parse().

getBipartitions()
Get all bipartitions.

Returns a list of `tree.bipartition` objects

getBranchFromBipartition(*bip*)
Given a bipartition object, return a branch that creates that partition of taxa.

Parameters **bip** (a `tree.bipartition` object) – a bipartition

Returns a `newick.branch` object

getBranchFromStrBipartition(*bip*)
Given a bipartition of taxa, return a branch that creates that partition of tree taxa.

Parameters **bip** – a tuple of taxa names

Returns a `newick.branch` object

getBranches()
Return all branches from this topology.

Returns a list of `newick.branch` objects

getLeaves()
Return all leaves from this topology.

Returns a list of `newick.node` objects

getStrBipartitionFromBranch(*br*)
Given a branch, return corresponding bipartition.

Parameters **br** (a `newick.branch` object) – a branch

Returns a `tree.bipartition` object

iterNNIForBranch(*br*, *flip=True*)
Consider all valid NNI moves for a given branch in the topology and and yield all possible rearrangements as a generator.

iterSPRForBranch(*br*, *flip=True*)
Consider all valid SPR moves for a given branch in the topology and yield all possible rearrangements as a generator.

iterTypeForBranch(*br*, *type=1*, *flip=True*)
Iterate over all possible rearrangements for a branch using a given rearrangement operator type defined in this module. For example, calling this function by providing TYPE_NNI as the type will iterate over all NNI operations. By default, the type is TYPE_SPR.

lockBranch(*branch*)
Given a branch, lock it such that no transitions can ever occur across it.

Parameters **branch** (a `newick.branch` object) – a branch

Returns a boolean (True if success)

move(*branch*, *destination*, *returnStruct=True*)
Move a branch and attach to a destination branch. Return new structure, or return merely the resultant Newick string.

parse(*newickstr*)
Parse a newick string and assign the tree to this object. Cannot already be initialized with a tree.

Returns None

rerootToLeaf(*toleaf=None*)
Reroots the given tree structure such that it is rooted nearest the lowest-order leaf or a provided leaf.

Parameters **toleaf** (a `newick.node` object) – a leaf node from this topology

toNewick()
Return the newick string of the tree.

Returns a Newick string (rooted)

toTree()
Return the tree object for this topology.

Returns a new `tree.tree` object

toUnrootedNewick()
Return the newick string of the tree as an unrooted topology with a multifurcating top-level node.

Returns a Newick string (unrooted)

toUnrootedTree()
Return the tree object of the unrooted version of this topology.

Functions for phylogenetic tree goodness-of-fit scoring.

pylogeny.scoring.**beaglegetLogLikelihood**(*tree*, *alignment*)
Acquire log-likelihood via C++ library BEAGLE via use of pybeaglethon wrapper library. Currently uses HKY85 model.

Parameters

- **tree** – A tree object.
- **alignment** – An alignment object.

Returns A floating point value.

pylogeny.scoring.**getLogLikelihood**(*tree*, *alignment*, *updateBranchLengths=True*)
Acquire log-likelihood via C library libpll.

Parameters

- **tree** – A tree object.
- **alignment** – An alignment object.
- **updateBranchLengths** – Whether or not to update the branch lengths

in the provided tree with optimized ones. :returns: A floating point value.

pylogeny.scoring.**getParsimony**(*newick*, *alignment*)
Acquire parsimony via a C++ implementation.

Parameters

- **newick** – A New Hampshire (Newick) tree string.
- **alignment** – An alignment object.

Returns An integer value.

pylogeny.scoring.**getParsimonyForTopology**(*topo*, *alignment*)
Acquire parsimony via a C++ implementation.

Parameters

- **topo** – A topology object.
- **alignment** – An alignment object.

Returns An integer value.

pylogeny.scoring.**getParsimonyFromProfiles**(*newick*, *profiles*)
Acquire parsimony via a C++ implementation.

Parameters

- **newick** – A New Hampshire (Newick) tree string.
- **profiles** – A set of profiles corresponding to an alignment.

Returns An integer value.

pylogeny.scoring.**getParsimonyFromProfilesForTopology**(*topology*, *profiles*)
Acquire parsimony via a C++ implementation.

Parameters

- **topo** – A topology object.
- **profiles** – A set of profiles corresponding to an alignment.

Returns An integer value.

Container definition for (phylogenetic) bifurcating or multifurcating trees defined using Newick strings, collections of them, and for splits of these trees.

class pylogeny.tree.**bipartition**(*topol*, *bra=None*)
Bases: object

A tree bipartition. Requires a tree topology. Using the term borrowed from nomenclature of a bipartite graph, a bipartition for a phylogenetic tree coincides with the definition of two disjoint sets U and V . A branch in a phylogenetic tree defines a single bipartition that divides the tree into two disjoint sets U and V . The set U comprises all of the children leaf of the subtree associated with that branch. The set V contains the rest of the leaves or taxa in the tree.

__init__(*topol*, *bra=None*)
Construct a bipartition from a branch in a topology.

Parameters

- **topol** (rearrangement.topology) – A topology.
- **bra** (newick.branch) – An optional branch object.

fromStringRepresentation(*st*)
Acquire all component elements from a string representation of a bipartition.

Parameters **st** – A string representation from a tree.bipartition object.

getBestSPRScore(*ls*, *node=None*)
Given a landscape, return the best SPR score.

getBranch()
Get branch corresponding to this bipartition.

Returns newick.branch

getBranchIndex()
Return an index of the branch with respect to a post order traversal of the topology.

Returns an integer

getBranchListRepresentation()
Get the tuple of lists of branches that represent this bipartition.

getMedianSPRScore(*ls*, *node=None*)
Given a landscape, return the median SPR score.

getSPRRearrangements()
Return the set of all scores related to this bipartition.

getSPRScores(*ls*, *node=None*)
Given a landscape, return all possible scores, not actively performing scoring if not done.

getShortStringMappings()
Get the mapping of symbols from taxa names for the shorter string representation.

getShortStringRepresentation()
Get the shorter string representation corresponding to this bipartition.

Returns a string

getStringRepresentation()
Get the string representation corresponding to this bipartition.

Returns a string

pylogeny.tree.**median**(*lst*)

pylogeny.tree.**numberRootedTrees**(*t*)

pylogeny.tree.**numberUnrootedTrees**(*t*)

class pylogeny.tree.**tree**(*newi=''*, *check=False*, *structure=None*)
Bases: object

Defines a single (phylogenetic) tree by Newick string; can possess other metadata.

__init__(*newi=''*, *check=False*, *structure=None*)
If enabled, "check" will force the structure to reroot the given Newick string tree to a lowest-order leaf in order to ensure a consistent Newick string among any duplicate topologies. If a structure is provided and

check is disabled, all parsing routines are bypassed and the Newick and Structure fields of this tree are overriden by the appropriate arguments.

Parameters

- **newi** (*a string*) – A Newick or New Hampshire string for a tree.
- **check** (*a boolean*) – Perform parsing checks on the string input.

getName()
Gets the name of this tree if it has been defined.

Returns a string

getNewick()
Gets the Newick (New Hampshire) string for this tree.

Returns a string

getOrigin()
Gets the "origin" of this tree, or where this tree was acquired or constructed from. Usually set by other code or an interface.

Returns string or None

getRerootedNoBranchLengthNewick()
Returns the tree's "structure", a Newick string without any branch lengths.

Returns a string

getScore()
Gets the score(s) (objective function) for this tree if it/they has/have been defined.

Returns a tuple of floats or integers

getSimpleNewick()
Return a Newick string with all taxa name replaced with successive integers.

Returns a string

getStructure()
Returns the tree's "structure", a Newick string without any branch lengths.

Returns a string

setName(*n*)
Sets the name of this tree (object).

Parameters **n** (*a string*) – a string indicating this tree's name

setOrigin(*o*)
Set the "origin" or specification of where this tree was acquired or constructed from.

Parameters **o** (*string or None*) – A string indicating where the tree came from.

setScore(*s*)
Sets the score(s) for this tree. Should be performed by a scorer (see scoring functions in the appropriate module).

Parameters **s** (*a tuple of floats or integers*) – a set of objective function scores.

toNewick()
Gets the Newick (New Hampshire) string for this tree.

Returns a string

toTopology()
Return a topology object instance for this tree to allow for rearrangement of the actual structure of the tree.

Returns a rearrangement.topology object

updateNewick(*n*, *reroot=False*)
Update the contained Newick string only as long as the structure obtained (after rerooting, which is an optional parameter) is identical to the contained structure.

Parameters

- **n** (*a string*) – A Newick or New Hampshire formatted string.
- **reroot** (*a boolean*) – reroot to lexicographically lowest-order leaf.

class pylogeny.tree.**treeSet**
Bases: _abcoll.Sized, _abcoll.Iterable

Represents an ordered, disorganized collection of trees that do not necessarily comprise a combinatorial space.

__init__()

addTree(*tr*)
Add a tree object to the collection.

Parameters **tr** (tree.tree) – A tree object.

addTreeByNewick(*newick*)
Add a tree to the structure by Newick string.

Parameters **newick** (*a string*) – A New Hampshire or Newick string.

static fromTreeFile(*fin*)
Acquire a file where newlines separate Newick strings, and create an instance of treeSet from those trees.

indexOf(*tr*)
Acquire the index in this collection of a tree object. Returns -1 if not found.

Parameters **tr** (tree.tree) – A tree object.

Returns an integer [-1,length of collection)

iterTrees()
Iterate over all trees found in this set.

removeTree(*tr*)
Remove a tree object from the collection if present.

Parameters **tr** (tree.tree) – A tree object (present in the collection).

toTreeFile(*fout*)
Output this landscape as a series of trees, separated by newlines, as a text file saved at the given path.

Parameters **fout** (*a string*) – A string indicating a file system path to a file.

1.2 Module contents

Pylogeny is a Python library and code framework for phylogenetic tree reconstruction and scoring.

Allows one to perform the following tasks: (1) Generate and manage phylogenetic tree landscapes. (2) Build and rearrange phylogenetic trees using preset operators such as NNI, SPR, and TBR. (3) Score phylogenetic trees by Log-likelihood and Parsimony.

Dependencies: Pandas, P4 Phylogenetic Library. Suggested: FastTree, RAxML, PytBEAGLEhon.

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

p

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

P

Q

R

S

T

U

V

W