Exercise 1

Write a script that adds edge lengths to a tree structure. The edge lengths should be drawn from a normal distribution with a mean and variance equal to the edge length of the edge of the parent node. The edge length of the root node should be drawn from a normal distribution with a mean and variance equal to 1.0.

Notes

- Some tree strings for you to work with can be found in the "`exercises/data/`" directory. You can either copy and paste them into your Python script directly, and then read them into a `Tree` or `TreeList` object like this:

```
import dendropy
datastr = """\
... (tree data here) ...
"""
trees = dendropy.TreeList.get(
    data=datastr,
    schema="newick",)
```

  Or you can read them directly from the file source:

```
import dendropy
trees = dendropy.TreeList.get(
    path="exercises/data/cetaceans.mcmc.newick",
    schema="newick",)
```

- You can work with the `Tree` objects collected in a `TreeList` instance just like you would elements in a Python `list`:

```
number_of_trees = len(trees)
first_tree = trees[0]
second_tree = trees[1]
last_tree = trees[-1]
for tree in trees:
    print(tree.as_string("newick"))
```

- You can *traverse* a tree (visit all the nodes in a tree) in a number of different ways.
  - By default, iterating over a tree *visits each node in a tree in* ***pre-order***. That is, each node is visited before its children, with the root node (or "seed node" in DendroPy terms, to remain neutral as to whether the tree itself is rooted or not) being the first node.

```
for nd in tree:
    print(nd)
```

  - You can also visit nodes in other orders:

```
for nd in tree.postorder_node_iter():
    print(nd)
for nd in tree.preorder_node_iter():
```

```
    print(nd)
for nd in tree.levelorder_node_iter():
    print(nd)
for nd in tree.inorder_node_iter():
    print(nd)
```

– Specialized iterators visit only particular nodes:

```
for nd in tree.internal_node_iter():
    print(nd)
for nd in tree.leaf_node_iter():
    print(nd)
```

– If you want to focus on edges, a parallel set of iterators exist for them:

```
for edge in tree.postorder_edge_iter():
    print(edge)
for edge in tree.preorder_edge_iter():
    print(edge)
for edge in tree.internal_edge_iter():
    print(edge)
for edge in tree.leaf_edge_iter():
    print(edge)
```

- A `Node` object includes the following attributes or properties:

  `edge` A reference to the `Edge` instance representing the edge subtending this node.

  `taxon` A reference to the `Taxon` instance representing the OTU associated with this node. This can be `None` if there is no OTU associated with this node.

  `parent_node` A reference to the `Node` instance representing the parent of this node on the tree. If this node is the root or seed node, then the value of this property is `None`.

  and methods:

  `child_node_iter()` An iterator over the children of this node:

```
for nd in tree.postorder_edge_iter():
   for ch in nd.child_node_iter():
       print("Node {} has child: {}".format(nd, ch))
```

  `is_leaf()` Returns `True` if this node has no children; `False` otherwise.

  `num_child_nodes` Returns the number of (immediate) children of this node.

- An `Edge` object includes the following attributes or properties:

  `tail_node` A reference to the `Node` instance representing the *parent* or source of this edge. This will `None` if this edge subtends the root or seed node.

  `head_node` A reference to the `Node` instance representing the *parent* or source of this edge.

  `taxon` A reference to the `Taxon` instance representing the OTU associated with this node. This can be `None` if there is no OTU associated with this node.

  `length` The weight or length of this edge. This can be any value, but DendroPy typically expects this to be either numeric (floating point or integer) or `None`.

and methods:

`child_node_iter()` An iterator over the children of this node:

```
for nd in tree.postorder_edge_iter():
   for ch in nd.child_node_iter():
       print("Node {} has child: {}".format(nd, ch))
```

Exercise 2

Read in a collection of trees ("`exercises/data/cetaceans.mcmc.nex`") and calculate a summary tree for the collection (for e.g., a 50% majority-rule consensus tree or the maximum clade credibility tree). Then read in a (DNA) character data set ("`exercises/data/cetaceans.chars.nex`") and score the summary tree under a parsimony criterion with this data. Write out a data file that includes the character matrix and the summary tree, with the score indicated either as an annotation or a comment.

Notes

- As you will be reading data from two independent data sources, you will have to ensure that the data sources are read with reference to the same taxonomic namespace. You will have to create a taxonomic namespace instance and pass these into the factory methods.

```
tns = dendropy.TaxonNamespace()
trees = dendropy.TreeList.get(
    path=...
    schema=...
    taxon_namespace=tns,
    )
chars = dendropy.DnaCharacterMatrix.get(
    path=...
    schema=...
    taxon_namespace=tns,
    )
```

- A `TreeList` object supports a number of methods that summarize its collection of trees:

  `consensus()` This returns the majority-rule consensus tree for the collection. More information can be found in the Primer or in the API documentation.

  `maximum_product_of_split_support_tree()` This returns the tree that maximizes the product of split supports, also known as the "Maximum Clade Credibility Tree" or the MCCT.

  `maximum_sum_of_split_support_tree()` This returns the tree that maximizes the sum of split supports.

  The trees returned by these methods are automatically given the same taxon namespace as the collection. There are MANY options for customizing the tree, including how the support is indicated, different ways of summarizing the branch lengths, etc.

- The "`dendropy.calculate.treescore`" module provides different methods for scoring a tree. Here, we are interested in the "`parsimony_score`" method of this module. This method, at minimum, takes a tree and a character matrix, and returns its score. You can also specify how gaps are to be treated and if you spply it with a list object you get the individual site scores as well.

```
pscore = treescore.parsimony_score(
        tree=tree_to_be_scored,
        chars=character_matrix,
        gaps_as_missing=True,)
```

- To write out a complete phylogenetic dataset of mixed data (i.e., tree and characters), we have to assemble a `DataSet` object:

```
dataset = dendropy.DataSet()
```

  Adding character data to this object is simple:

```
dataset.char_matrices.add(chars)
```

  However, we cannot add the `Tree` object directly, because the `DataSet` level of organization only keeps tracks of `TreeList` instances. So we first have to create a new `TreeList` instance, making sure that it has the same `TaxonNamespace` reference as our summary `Tree`, add the `TreeList` instance to the `DataSet`, and *then* add our summary `Tree` to the `TreeList`. We can use the `new_tree_list` of the `DataSet` object to combine the first and second steps.

```
summary_tree_list = dataset.new_tree_list(taxon_namespace=tns)
summary_tree_list.append(summary_tree)
```

- We can add a comment to most phylogenetic data objects by simply appending to its "`comments`" attribute, which is a `list`.

```
summary_tree_list.comments.append("Hello, world")
dataset.comments.append("Consensus tree scored under parsimony")
dataset.comments.append("Parsimony score = {}".format(pscore))
```

- A more structured approach to recording this information would be to use metadata annotations.

```
summary_tree_list.annotations["parsimony_score"] = pscore
```

- We can write out the `DataSet` (as well as most phylogenetic data objects) using the instance's `write` method:

```
dataset.write(path="data.nex", schema="nexus")
dest = open("stree.tre", "w")
summary_tree.write(file=dest, schema="newick")
```

- Or, to obtain a string rendering, we can call "`as_string`". This method has an identical signature to "`write`", *except* it does not take a destination argument (i.e., you do not have to specify a "`path`" or "`file`" keyword argument).

```
print(dataset.as_string(schema="nexus"))
print(summary_tree.as_string(schema="newick"))
```

Exercise 3

Write a script that samples the distribution of the mean pairwise patristic distance for trees of a fixed size generated under two different branching models, the birth-death and the pure neutral coalescent. For this exercise, these can be 100-tip trees. In the case of the birth-death model, use a birth rate of 0.10 and a death rate of 0.01. Population size is an important parameter for the coalescent, but if we can live with our branch lengths being in units of (haploid) population size (i.e., units of N), we can leave it at 1.0 for the simulation.

Notes

- When we do not really care about the identities of our OTU's, we can quickly generate a taxon namespace to use by passing in a list of labels to the `TaxonNamespace` constructor:

```
tns = dendropy.TaxonNamespace(["t1", "t2", ..., "t100"])
```

  Or, more elegantly:

```
tax_labels = ["T{:03d}".format(i+1) for i in range(ntax)]
tns = dendropy.TaxonNamespace(tax_labels)
```

- DendroPy provides for simulating trees under a very broad range of models, all collected together in the module "`dendropy.simulate`".

- To simulate a birth-death tree, you would call "`dendropy.simulate.birth_death`". This method is described in detail in the API documentation as well as covered in the Primer. We probably want to use the "general sampling approach" of Hartmann *et alia*, and we can probably get away with a small threshold given our low death rate relative to the birth rate.

```
from dendropy import simulate
tree = simulate.birth_death_tree(
        birth_rate=0.10,
        death_rate=0.01,
        gsa_ntax=ntax * 3,
        taxon_namespace=tns)
```

- To simulate a neutral coalescent tree, you would call "`dendropy.simulate.pure_kingman_tree`". This method is described in detail in the API documentation as well as covered in the Primer. We call it the "pure Kingman" process because DendroPy also supports simulation under other coalescent processes, such as the multispecies coalescent ("gene trees in species trees").

```
from dendropy import simulate
tree = simulate.pure_kingman_tree(taxon_namespace=tns)
```

- Given a `Tree` instance, you can ask DendroPy to generate a phylogenetic distance matrix for the taxa on the tips of tree, represented by a `PhylogeneticDistanceMatrix` object, by calling on the `phylogenetic_distance_matrix` method of the `Tree` object.

```
pdm = tree.phylogenetic_distance_matrix()
```

  A phylogenetic distance matrix is a matrix in which the top row and first column list OTU's, and the cells list the distance between the two cross-referenced OTU's.

- The `PhylogeneticDistanceMatrix` is a very powerful calculator, providing not only distances between OTU's, but also ways to, e.g., quickly dereference the MRCA (Most Recent Common Ancestor) between any pair of taxa, as well as calculations of various community ecology statistics, such as the MPD, MNTD, standardized effect sizes of these, etc. This class is discussed in detail in the Primer.

- To get the distance between any two pairs of OTU's, we can call the "`distance()`" method:

```
d = pdm.distance(
    taxon1=t1,
    taxon2=t2,
    is_weighted_edge_distances=True,
    is_normalize_by_tree_size=True)
```

  Here, "`is_weighted_edge_distances`" argument determines whether we want to consider the edge weights of the tree when calculating the distances between two taxa, or just the number of edges or path steps. We are interested in the former, i.e. the *patristic distance*, so we indicate `True`. The "`is_normalize_by_tree_size`" argument determines whether the distance should be divided by the total tree length (if weighted) or the number of tips (if unweighted). Again, we indicate `True`, so that we can compare these values across trees of different sizes or with different branch length units (as is the case here).

- To collect the distances for all distinct pairs of taxa on a tree, we could do something like:

```
distances = []
for i1, t1 in enumerate(tns[:-1]):
    for i2, t2 in enumerate(tns[i1+1:]):
        distances.append(pdm.patristic_distance(
            taxon1=t1,
            taxon2=t2,
            is_normalize_by_tree_size=True,))
mean_d = sum(distances)/len(distances)
```

  However, the `PhylogeneticDistanceMatrix` object conveniently provides a `distances()` method, which gives us a list of all the distances at once:

```
dists = pdm.distances(is_weighted_edge_distances=True,
   is_normalize_by_tree_size=True)
mean_dists = sum(dists)/len(dists)
```

Exercise 4

In the previous exercise we compiled a distance matrix from a tree. Here, we generate a tree from a distance matrix.

The file "`exercises/data/laurasiatherian.distances.ml.csv`" contains a matrix of the maximum likelihood estimate of distances between pairs of some Eutheria. Generate and write the NJ (neighbor-joining) and UPGMA (Unweighted Pair Group Method with Arithmetic Mean) trees for these distances.

Notes

- This one is a quick and easy one, thanks to the `PhylogeneticDistanceMatrix` methods for generating the trees: `nj_tree` and `upgma_tree`.

```
nj_tree = pdm.nj_tree()
upgma_tree = pdm.upgma_tree()
```

- To instantiate a `PhylogeneticDistanceMatrix` object from a distance matrix specified as delimited file, we use the class factory method `from_csv`. As always, we take charge of ensuring we know what the taxonomic namespace is.

```
src = open("data/laurasiatherian.distances.ml.csv")
tns = dendropy.TaxonNamespace()
pdm = dendropy.PhylogeneticDistanceMatrix.from_csv(
        src=src,
        taxon_namespace=tns,)
```

Exercise 5

How similar are ML bootstrap trees vs. MCMC trees? The *Robinson-Foulds* distance is probably the most well-known tree comparison metrics in phylogenetics. Calculate the mean, sample variance, and 5% and 95% quantiles of the distribution of RF distances over all pairwise comparisons of bootstrap vs. MCMC trees in two collections: "`exercises/data/cetaceans.bootstraps.trees`" and "`exercises/data/cetaceans.mcmc.nex`".

Notes

- These are not very large tree samples, but as we are doing all pairwise comparisons, the number of calculations can result in run times that are too long for development work. As you would when developing solutions for real-world problems, you want a small test data set you can use while developing your script until it is ready for application, to speed up the "edit-run-edit" cycle. These are provided for you here for this exercise in the form of: "`exercises/data/cetaceans.bootstraps10.trees`" and "`exercises/data/cetaceans.mcmc10.nex`". Use them initially!

- The "`dendropy.calculate.treecompare`" module provides a rich range of methods for comparing two trees. The RF distances come in two flavors: one that takes into consideration branch lengths (`weighted_robinson_foulds_distance`) and one that does not (`unweighted_robinson_foulds_distance`). Using them is simple:

```
from dendropy.calculate import treecompare
rfd = treecompare.unweighted_robinson_foulds_distance(t1, t2)
```

- The "`dendropy.calculate.statistics`" modules provides a range of convenient statistical functions. This module is documented in the API reference.

Exercise 6

The NEXUS tree file "`data/seasia.nex`" contains a tree in which the taxa associated with the leaf nodes are differentially distributed across the following areas in Southeast Asia and the Western Pacific: Malaya, Indochina, Borneo, Java, Sumatra, Palawan, Papua, and the Solomons. Note that Malaya and Indochina are mainland Southeast Asia areas, while all the others are insular Southeast Asia or Western Pacific areas. The distribution of each leaf node taxon is indicated by metadata annotations. For example,

```
[&malaya=True,indochina=False,borneo=True,java=True,sumatra=True,sulawesi=
   False,palawan=False,papua=True,solomons=False]
```

indicates that the taxon is only found on Malaya, Borneo, Java, Sumatra, and Papua, and not in Indochina, Sulawesi, Palawan, and the Solomons. Out of this tree, extract a tree that ONLY includes tips if the associated taxa are found exclusively in the islands, but not in the mainland (i.e., do not occur on either Malaya or Indochina).

Notes

- Reading in a tree in NEXUS formats automatically reads in the metadata annotations. These metadata annotations can be associated with OTU's/taxa if given in the TAXA block, or with nodes if given in the tree. Most phylogenetic data objects have a `annotations` attribute that has a special `AnnotationSet` object as a value. This object is a collection of `Annotation` objects. `Annotation` objects have two importan attributes: `name`, which is the key or label of the metadata, and `value`, which is its value. The support for metadata annotations in NEXUS, especially as far as value typing and complex values (like dictionaries, etc.) is limited. Nonetheless, a lot can still be reliably and flexibly expressed. For the purposes of this exercise, we will use the `get_value` method of the `AnnotationSet` object to directly access the metadata value associated with an object by key.

```
for nd in tree.leaf_node_iter():
   if nd.annotations.get_value("malaya"):
       print("{} is found in Malaya".format(nd.taxon.label))
   print(nd.annotations.get_value("indochina"))
   nd.annotations["fiji"] = False # key-based value assignment
```

- So we can sort our taxa by:

```
to_retain = set()
to_prune = set()
for nd in tree.leaf_node_iter():
    if not nd.annotations.get_value("malaya") and not nd.annotations.
   get_value("indochina"):
        to_retain.add(nd.taxon)
    else:
        to_prune.add(nd.taxon)
```

- `Tree` objects have a very rich suite of methods for manipulation. Key among those are the various prune and retain methods: `prune_taxa`, `retain_taxa`, etc. The `prune_taxa` method takes a collection of `Taxon` objects as an argument and modifies the `Tree` instance in-place by pruning the nodes associated with those taxa.

```
to_prune = set(...)
tree.prune_taxa(to_prune)
```

The `retain_taxa` method takes a collection of `Taxon` objects as an argument and modifies the `Tree` instance in-place by pruning all nodes *not* associated with those taxa.

```
to_retain = set(...)
tree.retain_taxa(to_retain)
```

Exercise 7

The tree file "`exercises/data/Bininda-emonds_2007_mammals.nexus`" contains a very large tree of 4510 mammal taxa. Split this tree into multiple smaller trees based on genera, as long as there are three or more species in represented for the genus. That is, the output should consist of a collection of trees, where for every distinct genus with three or more representatives, there will be a tree consisting of just the members of that genus.

Notes

- This is a very large tree. So, as before, you should develop and finalize your script using a smaller dataset (e.g., "`exercises/data/pythonidae.mle.nex`").
- The quickest way to establish the groups is to iterate over the taxa in the taxon namespace, using a dictionary to keep track of the groups:

```
groups = collections.defaultdict(list)
for taxon in tree.taxon_namespace:
    genus_name = taxon.label.split(" ")[0]
    groups[genus_name].append(taxon)
```

- A very important thing to note: do NOT be tripped up by the underscores you see in taxon names! If you were to open the data file in a text editor, you will see that the taxon label might be listed as "Python_regius"; but when this gets instantiated into a `Taxon` object in DendroPy, the resulting object will have a `label` of "Python regius". This is correct behavior! As per the Newick/NEXUS standards, by default *underscores in labels are* ***automatically translated into spaces*** when read by DendroPy! Many people are unaware of this, and, to add to the confusion, many programs (some major) blithely fail to respect this convention, even though it is part of a published standard. And admittedly, standard or not, this can cause major problems when traversing or using mixed formats, such as going from NEXUS to FASTA or vice versa, as other formats do not have this convention: you then have to make it a point to either strip all underscores and spaces completely from labels or remember to translate appropriately each and everytime (quotes will preserve the underscores, but then you have to deal with quotes on the FASTA end...).

  DendroPy does, in fact, allow you the specify that underscores should not be translated when reading in data:

```
tree = dendropy.Tree.get(
        ...
        ...
        preserve_underscores=True)
```

  Here, we assume that the "`preserve_underscores=True`" has not been specified. This is why we use a space character above to split the OTU label into genus and species epithets, and *not* an underscore.

- Once we have got our groups, it seems straightforward: just use the "`retain_taxa`" method to get our trees. But this is **NOT** the way to go here! This is a very large tree, and `retain_taxa` and `prune_taxa` are relatively expensive operations because they modify an existing tree in place. Furthermore, because they modify the tree in-place, it means that if we want to pull

multiple trees out of the original tree, we will either have to clone the tree or re-read it from the data source each time: again, expensive operations either way.

- Instead, we will use the nifty "`extract*`" family of `Tree` methods:
  - `extract_tree`
  - `extract_tree_with_taxa`
  - `extract_tree_without_taxa`
  - `extract_tree_with_taxa_labels`
  - `extract_tree_without_taxa_labels`

  These methods are discussed in detail in the Primer. Basically, instead of modifying the existing tree in place, they create a *new* `Tree` instance based on the structure of the current tree, either pruning (e.g. `extract_tree_without_taxa`) or retaining the specified taxa (e.g. `extract_tree_with_taxa`).

```
genus_taxa = set(...)
genus_tree = tree.extract_tree_with_taxa(genus_taxa)
```

  These methods are much, much, much faster than cloning/re-reading a tree and then using `prune_taxa` or `retain_taxa`. The major downside to this family of methods? Metadata annotations are NOT copied to the new tree. In this case, this is not a problem. The take-home message is that, unless you want metadata annotations to be preserved in your pruned tree, you should prefer the "`extract*`" family of methods for all your pruning needs!

Exercise 8

You have three trees that summarize split support under parsimony, maximum likelihood, and Bayesian approaches to tree inference. Compile for presentation a tree that summarizes split support across these different approaches by an internal node label, e.g. "30/70.8/0.9". The trees are:

`data/cetaceans.x.mcmc.con.support-as-brlens.nex` The tree summarizing support under parsimony, with the support indicated by branch lengths.

`data/cetaceans.x.ml.con.support-as-labels.nex` The tree summarizing support under maximum likelihood, with the support indicated by internal node labels.

`data/cetaceans.x.mcct.support-as-annotes.nex` The tree summarizing support under Bayesian inference, with the support indicated by metadata annotation "support".

You can use the third topology (the MCCT tree) as the target tree onto which you are going map the description of support. Note that the three topologies differ slightly, and bipartitions/splits on one tree may not be found in the others. In this case, you should indicate this by a "NA".

Notes

- You are going to need to make extensive use of `Bipartition` objects to relate split identity across different trees.
- Read the trees in as you would normally, making sure they all are created within the same taxonomic namespace. Also it is a good idea to explicitly order a rooting, so that there is no confusion.

```
tns = dendropy.TaxonNamespace()
t1 = dendropy.Tree.get(
    path="data/cetaceans.x.mcmc.con.support-as-brlens.nex",
    schema="nexus",
    taxon_namespace=tns,
    rooting="force-rooted",
    )
t2 = dendropy.Tree.get(
    path="data/cetaceans.x.ml.con.support-as-labels.nex",
    schema="nexus",
    taxon_namespace=tns,
    rooting="force-rooted",
    )
t3 = dendropy.Tree.get(
    path="data/cetaceans.x.mcct.support-as-annotes.nex",
    schema="nexus",
    taxon_namespace=tns,
    rooting="force-rooted",
    )
```

- Then call the `encode_bipartitions` method of each tree so that the bipartition hashes get calculated.

```
t1.encode_bipartitions()
t2.encode_bipartitions()
t3.encode_bipartitions()
```

- Then iterate over all internal nodes of the target tree:

```
for nd in tree.preorder_internal_node_iter():
    ...
```

- As you visit each node, you can access the `Bipartition` object associated with the split/edge subtending that node by:

```
    bipartition = node.edge.bipartition
```

- We will use this `Bipartition` object to dereference the same split in all three of our source trees. As we have encoded the bipartitions, each `Tree` instance will have an attribute, "`bipartition_edge_map`". This is a dictionary where the keys are `Bipartition` instances representing bipartitions on the tree, and the values are `Edge` instances on the tree corresponding to the edge associated with that bipartition.

```
edge_on_tree1 = tree1.bipartition_edge_map[bipartition]
edge_on_tree2 = tree2.bipartition_edge_map[bipartition]
edge_on_tree3 = tree3.bipartition_edge_map[bipartition]
```

  Note that not all bipartitions may be shared across each tree. If a bipartition is not found on a particular tree, the corresponding `Bipartition` object key will be missing from the "`bipartition_edge_map`" of that `Tree`. Your script will have to anticipate this and deal with it:

```
try:
    edge = t1.bipartition_edge_map[bipartition]
    ....
except KeyError:
    label_parts.append("NA")
```

- Once you have a handle on an `Edge` instance on the tree of interest, you can access the edge length by the `length` attribute:

```
    br_len = edge.length
```

- You can access the `Node` instance that edge subtends by its `head_node` attribute.

```
    tree2_node = edge.head_node
```

- The `Node` instance itself has a `label` attribute that tracks the node label

```
    internal_node_label = tree2_node.label
```

- While the metadata value for the "support" annotation can be accessed this way:

```
    support = tree2_node.annotations["support"].value
```

- You can compose and set the target tree node label by setting its `label` attribute:

```
target_node.label = "foo"
target_node.label = "/".join(label_parts) # label_parts = [0.1, 0.2,
0.4]
```