Physcraper Documentation

Release 0.1

OPenTreeOfLife

May 13, 2022

Documentation topics:

CHAPTER 1

Automated gene tree updating with the Open Tree of Life

Use a phylogenetic tree and a DNA alignment to automatically find and add nucleotide sequences from a genetic database, to reproducibly improve and advance phylogenetic knowledge within a biological group.

Physcraper relies on taxonomic and phylogenetic resources and programmatic tools from the Open Tree of Life project.

Physcraper also leverages on programmatic tools from the TreeBASE project and NCBI, as well as multiple software projects listed as *requirements* below, to create an automatic and reproducible workflow for phylogenetics.

CHAPTER 2

Citation

If you use Physcraper, please cite:

- Sánchez-Reyes, L.L., M. Kandziora, & E.J McTavish. (2021). *Physcraper: a Python package for continually updated phylogenetic trees using the Open Tree of Life*. BMC Bioinformatics 22, 355. doi: doi.org/10.1186/s12859-021-04274-6.
- Open Tree of Life, B. Redelings, L.L. Sanchez Reyes, K.A. Cranston, J. Allman, M.T. Holder, & E.J. McTavish. (2019). *Open Tree of Life Synthetic Tree (Version 12.3)*. Zenodo. doi: 10.5281/zenodo.3937741

CHAPTER 3

License

Physcraper is made available through the GNU General Public License v3.0

CHAPTER 4

Contact

The tool is under active development in the McTavish Lab. Please post a GitHub issue here or contact ejmctavish@ucmerced.edu if you need any help or have feedback.

CHAPTER 5

Requirements

Physcraper requires to install:

- Anaconda *Anaconda Software Distribution. Computer software. Vers. 4.8.0. Anaconda, July. 2021. Web. https://anaconda.com*
- Virtualenv
- MUSCLE *Edgar RC. MUSCLE: multiple sequence alignment with high accuracy and high throughput. Nucleic Acids Res. 2004 Mar 19;32(5):1792-7.* doi: 10.1093/nar/gkh340
- RAxML *Stamatakis, Alexandros. "RAxML version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies." Bioinformatics 30.9 (2014): 1312-1313.* doi: 10.1093/bioinformatics/btu033
- BLAST + This software is only needed if using a local genetic database. Note that BLAST + is automatically installed when installing Physcraper using Anaconda. *Camacho, C., Coulouris, G., Avagyan, V. et al. BLAST+: architecture and applications. BMC Bioinformatics 10, 421 (2009).* doi: 10.1186/1471-2105-10-421

Physcraper relies on the following Python packages that are installed:

- argparse
- biopython *Cock, P. J., Antao, T., Chang, J. T., Chapman, B. A., Cox, C. J., Dalke, A., et al. (2009). Biopython: freely available Python tools for computational molecular biology and bioinformatics. Bioinformatics, 25(11), 1422–1423.*
- configparser
- coverage
- DateTime
- DendroPy *Sukumaran, J and MT Holder. 2010. DendroPy: a Python library for phylogenetic computing. Bioinformatics 26: 1569-1571* doi: 10.1093/bioinformatics/btq228
- future
- m2r2
- nexson

- numpy *Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., … Oliphant, T. E. (2020). Array programming with NumPy. Nature, 585, 357–362.* doi: 10.1038/s41586-020-2649-2
- OpenTree *Emily Jane McTavish, Luna Luisa Sanchez-Reyes, Mark T. Holder. (2020). OpenTree: A Python package for Accessing and Analyzing data from the Open Tree of Life. BioRxiv 2020.12.14.422759* doi: 10.1101/2020.12.14.422759
- pandas *McKinney, W., & others. (2010). Data structures for statistical computing in python. In Proceedings of the 9th Python in Science Conference (Vol. 445, pp. 51–56).*
- pytest
- pytest-cov
- pytest-xdist
- recommonmark
- requests *Chandra, R. V., & Varanasi, B. S. (2015). Python requests essentials. Packt Publishing Ltd.*
- sh
- sphinx
- urllib3

5.1 Introduction to Physcraper

5.1.1 The Physcraper framework

While genome scale data is increasing rapidly, there are still large quantities of single locus nucleotide sequence data being uploaded to the US National Center on Biotechnology Information (NCBI) database GenBank. These data are often appropriate for looking at phylogenetic relationships, and have the advantage of being orthologous to genetic sequences that have been used to construct existing phylogenetic trees.

If you have access to a single gene or multilocus nucleotide alignment, and a phylogenetic tree, Physcraper automates adding nucleotide sequences of new lineage samples into your tree by using *Open Tree of Life* tools to reconcile Taxonomy, and the BLAST algorithm to search for loci in the GenBank genetic database that are likely to be locally similar to sequences in the initial DNA alignment.

By using a starting alignment and tree, Physcraper takes advantage of DNA loci alignments as homology hypotheses (ideally orthology, see FAQs) that previous researchers have assessed, curated, and deemed appropriate for the phylogenetic scope. The sequences added during a BLAST search are limited either to a user specified taxon or monophyletic group, or within the taxonomic scope of the ingroup of the starting tree.

These automated, reproducible trees can provide a quick inference of potential phylogenetic relationships, as well as flag problems in the taxonomic assignments of sequences, paralogy and orthology, and areas of potential systematic interest.

Figure 1 from Sanchez-Reyes et al. 2021: The Physcraper framework consists of 4 general steps. The methodology is further described in the Implementation section of this documentation.

5.1.2 The Open Tree of Life

The Open Tree of Life (OpenTree) is a project that unites expert, peer-reviewed phylogenetic inferences and taxonomy to generate a synthetic tree estimate of species relationships across all life.

OpenTree synthetic tree. Figure 1 from Hinchliff et al. 2015. For more information on the OpenTree project go to https://opentreeoflife.github.io

OpenTree aims to construct a comprehensive, dynamic and digitally-available tree of life by synthesizing published phylogenetic trees along with taxonomic data. Currently the tree comprises 2.3 million tips. However, only around 90,000 of those taxa are represented by phylogenetic estimates - the rest are placed in the tree based on their taxonomic names.

To achieve this, the OpenTree Taxonomy (OTT) constructs a reference taxonomy for taxonomic reconciliation, through an algorithmic combination of several source taxonomies, such as:

- Hibbet et al. 2007,
- SILVA,
- the Index Fungorum,
- Schäferhoff et al. 2010,
- the World Register of Marine Species
- the NCBI Taxonomy,
- the Global Biodiversity Information facility (GBIF) backbone Taxonomy, and
- the Interim Register of Marine and Nonmarine Genera (IRMNG).

5.2 Quick start with Physcraper

5.2.1 Updating a tree from Open Tree of Life

The Open Tree of Life data store, Phylesystem, contains more than 4,500 phylogenetic trees from published studies. The tips in these trees are mapped to a unified taxonomy, which makes these data searchable in a phylogenetically explicit way. This is a great place to start of finding existing estimates of phylogenetic relationships, and assessing regions of the tree of life which are lacking available phylogenetic estimates. There is a lot of sequence data available that has never been incorporated into any phylogenetic estimates.

Find a starting tree with your taxon of interest

For this example we will use a tree that is already in the Open Tree of Life database. You can find more details about finding a tree to update at the start section of this documentation.

To find trees containing your taxon of interest (e.g. 'Malvaceae') on OpenTree use:

```
$ find_trees.py --taxon_name "Malvaceae"
```

This prints a bunch of studies out to the screen. We will need an alignment to update (which OpenTree does not store), so let's just look at trees that have data stored in TreeBASE.

```
$ find_trees.py --taxon_name "Malvaceae" --treebase
```

There are a bunch of options!

Lets update the Wilkie et al. 2006 (https://doi.org/10.1600/036364406775971714) study. You can view the study on the OpenTree database at Wilkie, 2006

While this study was focused on the family Sterculiacea, phylogenetic inference have suggested that this taxon is not monophyletic, as you can see on its OpenTree homepage)

Let's take a look at how recent molecular data affect our inferences of relationships, and if there is sequence data for taxa that don't have any phylogenetic information available in the tree.

Run the auto-update

The script `physcraper_run.py` wraps together linking the tree and alignment, blasting, aligning sequences, and inferring an updated tree. Detailed explanation of the inputs needed can be found in the Run section of this documentation.

The BLAST search part of updating trees takes a long time. For example, this analysis took around 12 hours! We recommend running it on a cluster or other remote computing option.

```
$ physcraper_run.py -s pg_55 -t tree5864 -tb -r -o pg_55
```

The `-r` flag repeats the search on new sequences until no additional sequences are found. We have put example outputs from this command in `docs/examples/pg_55`, so that you can explore the outputs without waiting for the searches to complete.

5.2.2 Updating your own tree and alignment

You can upload your own tree to OpenTree to update it, and that way it will be included in the OpenTree synthetic tree! See Submitting-phylogenies-to-Open-Tree-of-Life for more info on this.

If you aren't ready to share your tree publicly, you can update it without posting it to OpenTree.

You need an alignment (single locus) and a tree. Note that the taxon labels in these two files should be the same.

You also need a file linking the labels in your tree and alignment to broader taxonomy. This can be easily generated via OpenTree's Bulk Taxonomic Name Resolution Service (bulk TNRS).

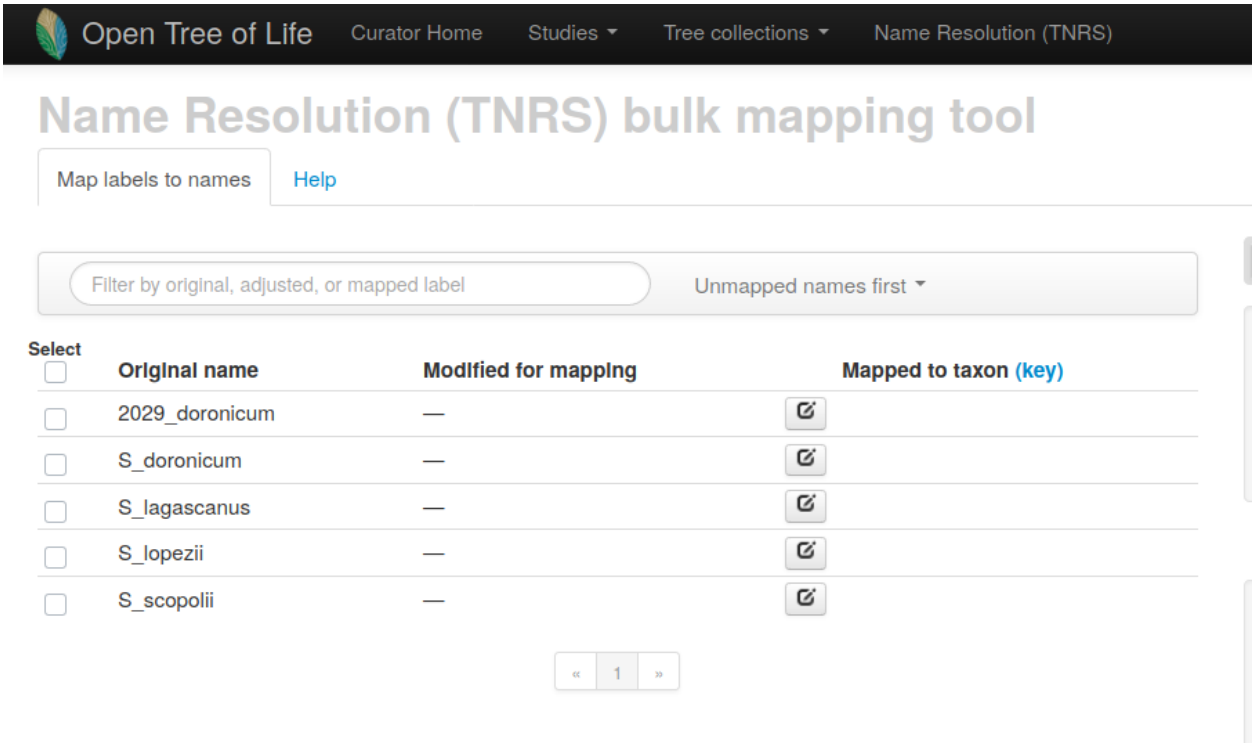
Automatically mapping names to taxa

You can automatically map your tip names to unique taxonomic identifiers using OpenTree's bulk Taxonomic Name Resolution Service (TNRS) tool. This can be done using R or Python programming languages, and the graphical user interface (GUI) version is available at https://tree.opentreeoflife.org/curator/tnrs/.

This is a brand new beta-version of this functionality, so some parts might be a bit finicky.

The first step is to save your tip taxon labels in a ".txt" file. There is an example file in the Physcraper folder at `docs/examples/example_tiplabels.txt`

Then, got to https://tree.opentreeoflife.org/curator/tnrs/ and click on `add names`, and upload the names file.

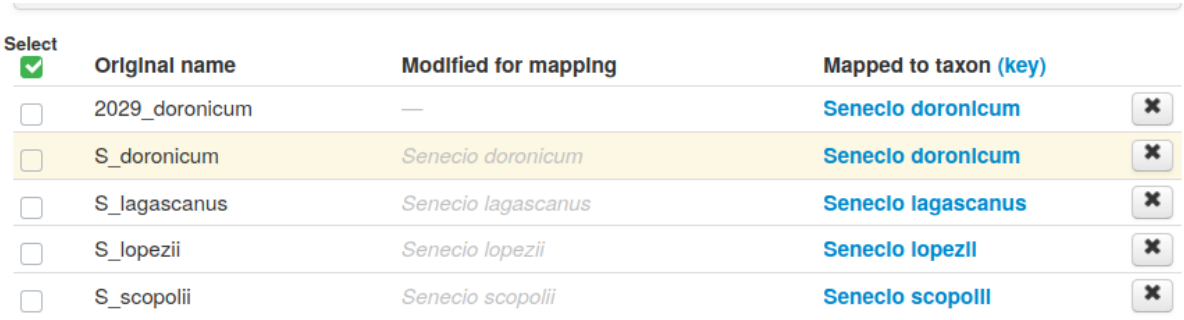


In the mapping options section, you can select a taxonomic group to narrow down the possibilities and speed up mapping. You can use regular expressions to replace or remove parts of labels for mapping.

Click on `Map selected names`

Exact matches will show up in green, and can be accepted by clicking `accept exact matches`.

Some taxa may show several suggested names. Click through to the taxonomy, and select the one that you think is correct based on the phylogenetic context.



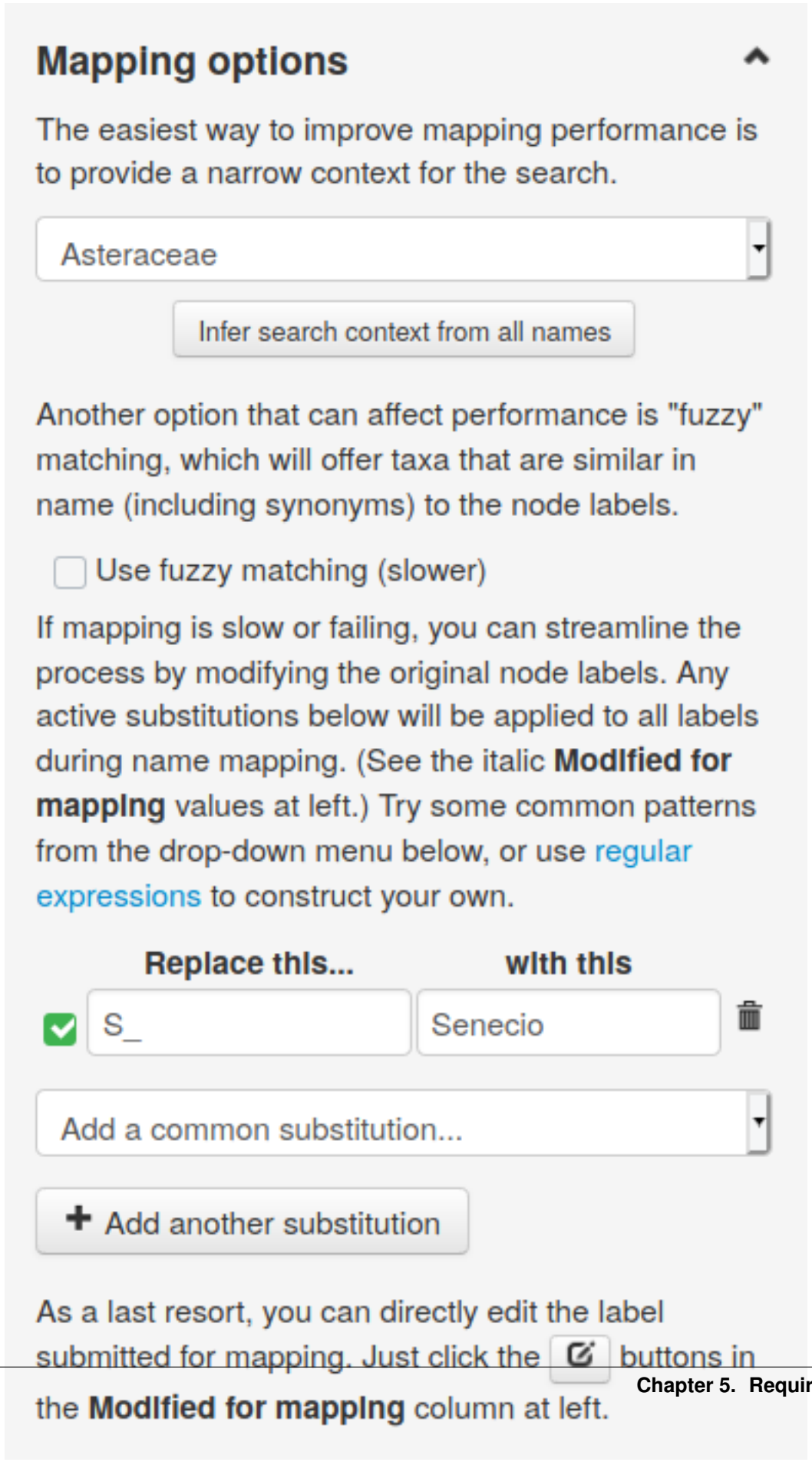
Once you have accepted names for each of the taxa, click `save nameset`.

Make sure your mappings were saved! If you don't click on ''accept matches'', they don't download.

Download your results to your laptop. Extract the files. Take a look at the human readable version at `output/main.csv`. You will see that this file also links to NCBI and GBIF identifiers for your taxa!

`output/main.json` contains the same data in a more computer readable format.

By passing in the `main.json` file, Physcraper can link your sequences to their correct taxonomic context.


Mapping options
The easiest way to improve mapping performance is to provide a narrow context for the search.
Asteraceae
Infer search context from all names
Another option that can affect performance is "fuzzy" matching, which will offer taxa that are similar in name (including synonyms) to the node labels.
Use fuzzy matching (slower)
If mapping is slow or failing, you can streamline the process by modifying the original node labels. Any active substitutions below will be applied to all labels during name mapping. (See the italic Modified for mapping values at left.) Try some common patterns from the drop-down menu below, or use regular expressions to construct your own.
Replace this...
with this
S_
Senecio
Add a common substitution...
Add another substitution
As a last resort, you can directly edit the label submitted for mapping. Just click the buttons in the Modified for mapping column at left.

Run the auto-update on your tree

Example run on local files using test data:

```
physcraper_run.py -tf tests/data/tiny_test_example/test.tre -tfs newick -a tests/data/
↪tiny_test_example/test.fas --taxon_info tests/data/tiny_test_example/main.json -as␣
↪fasta -o owndata
```

5.3 Installing Physcraper

While Physcraper can be installed via pip, in order to easily access the example data and ancillary files, we recommend downloading the Physcraper repository from GitHub and installing it locally following the instructions below. This process will also install the following python packages:

- Dendropy https://pythonhosted.org/DendroPy/
- Peyotl https://github.com/OpenTreeOfLife/peyotl (currently needs to be on the Physcraper branch)
- Biopython http://biopython.org/wiki/Download
- ConfigParser

5.3.1 Downloading Physcraper

First step is to download Physcraper to your computer.

You can do this with Git:

```
git clone https://github.com/McTavishLab/physcraper.git
```

or, you can download the repository from https://github.com/McTavishLab/physcraper.git

Now, move to the newly created "physcraper" directory with `cd physcraper` to continue.

Next step is to create a virtual environment to run Physcraper on. You can do this using Anaconda or Virtualenv.

5.3.2 Anaconda virtual environment

For this option you will of course need Anaconda installed. You can follow installation instructions on Anaconda's documentation website.

Now you can create a "conda virtual environment" with:

```
conda env create -f cond_env.yml
conda activate physcraper_env
pip install -r requirements.txt
pip install -e .
```

Note the "dot" at the end of that last command, and it should be ready!

5.3.3 Virtualenv virtual environment

For this option you will need Virtualenv installed.

Now you can go ahead and create a "Python virtual environment".

Remember you need to be in the "physcraper" folder (go there with `cd physcraper`). Once there do:

```
virtualenv -p python3 venv-physcraper
```

This will create a python 3 virtual environment named "venv-physcraper".

Activate the virtual environment with:

```
source venv-physcraper/bin/activate
```

Finally, install Physcraper inside the virtual environment:

```
pip install -r requirements.txt
pip install -e  .
```

Do not miss the "dot" at the end of that last command!

The virtual environment remains active even if you change directories. So, Physcraper will run from anywhere, while the virtual environment is activated.

Note that you will have to activate the virtual environment with `source venv-physcraper/bin/activate` every time you want to run Physcraper.

After you are finished working with Physcraper and you don't want to run it anymore, deactivate the virtual environment with:

```
deactivate
```

5.3.4 Checking for dependencies

Currently complete phylogenetic updating with Physcraper requires raxmlHPC and MUSCLE to be installed and in the path.

You can check if they are already installed with:

```
which muscle
which raxmlHPC
```

5.3.5 Checking installation success on remote searches

To test a full run with pre-downloaded BLAST results, copy the example results using:

```
cp -r docs/examples/pg_55_web pg_55_test
```

and then run:

```
physcraper_run.py --study_id pg_55 --tree_id tree5864 --treebase --bootstrap_reps 10 -
↪-output pg_55_test
```

There is more info on all the parameter settings in the documentation section Run, but briefly, this gets a tree (tree5864) from study pg_55 on OpenTree, pulls the alignment from TreeBASE, blasts the sequences, and does 10 bootstrap reps on the final phylogeny.

This example tests all the components except for the actual remote BLAST searches (because they can be very slow). To check if your installation was successful for remote searches, try running a full analysis:

```
physcraper_run.py --study_id pg_55 --tree_id tree5864 --treebase --bootstrap_reps 10 -
↪-output pg_55_new
```

This run will take a while - once it starts blasting, that means it's working! You can use Ctrl-C to cancel.

5.3.6 Local Databases

The BLAST tool can be run using local databases, which can be downloaded and updated from the National Center for Biotechnology Information (NCBI).

Installing BLAST command line tools

To BLAST locally you will need to install BLAST command line tools first. If you perfomed Physcraper installation using Anaconda, the BLAST command line tools will already be installed.

Find general instructions at BLAST's command line applications user manual and

at the index of blast executables

e.g. installing BLAST command line tools on **linux**:

```
wget https://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/LATEST/ncbi-blast-2.11.0+-
↪x64-linux.tar.gz
tar -xzvf ncbi-blast-2.11.0+-x64-linux.tar.gz
```

This link may be broken by NCBI BLAST executables updates - if so check https://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/LATEST/ for the newest version.

The binaries/scripts/executables will be installed in the `/bin` folder.

Installing BLAST command line tools on **MAC OS** is easy, with the installer. Note, however, that the BLAST executables will be installed in `usr/local/ncbi/blast` and that you will have to add this to your path in order to be able to run the executables, by adding `export PATH=$PATH:"usr/local/ncbi/blast/bin"` to the .bash_profile

If your terminal uses zshell instead of bash, make sure you're running the .bash_profile there too.

Downloading the NCBI database

If you want to download the NCBI BLAST database and taxonomy for faster local searches note that the download can take several hours, depending on your internet connection.

This is what you should do:

```
mkdir local_blast_db  # create the folder to save the database
cd local_blast_db  # move to the newly created folder
update_blastdb.pl nt  # download the NCBI nucleotide databases
cat *.tar.gz | tar -xvzf - --ignore-zeros  # unzip the nucleotide databases
update_blastdb.pl taxdb  # download the NCBI taxonomy database
gunzip -cd taxdb.tar.gz | (tar xvf - )  # unzip the taxonomy database
```

Downloading the nodes and names into the physcraper/taxonomy directory

```
cd physcraper/taxonomy
wget 'ftp://ftp.ncbi.nlm.nih.gov/pub/taxonomy/taxdump.tar.gz'
gunzip -f -cd taxdump.tar.gz | (tar xvf - names.dmp nodes.dmp)
```

Updating an existing BLAST database

```
cd local_blast_db  # move to the nucleotide database folder
update_blastdb nt  # download the NCBI nucleotide databases
# update_blastdb.pl nt  # on Mac OS
cat *.tar.gz | tar -xvzf - --ignore-zeros  # unzip the nucleotide databases
update_blastdb taxdb  # download the NCBI taxonomy database
# update_blastdb.pl taxdb  # on Mac OS
gunzip -cd taxdb.tar.gz | (tar xvf - )  # unzip the taxonomy database
```

Checking install success of local BLAST database

```
physcraper_run.py --study_id pg_55 --tree_id tree5864 --treebase --bootstrap_reps 10 -
↪db local_blast_db --output pg_55_local
```

This should start running a query using your local BLAST database.

Setting up an AWS BLAST database

To run BLAST searches without NCBI's required time delays, you can set up your own server on AWS (for $). See instructions at AWS marketplace NCBI BLAST

Create an NCBI API key

Generating an NCBI API key will speed up downloading full sequences following BLAST searches. See NCBI API keys for details

You can add your api key to your config using

```
Entrez.api_key = <apikey>
```

or as a flag in your physcraper_run script `--api_key`

5.4 How to find Physcraper inputs

Physcraper takes as input a phylogenetic tree and the corresponding DNA alignment. This section shows how to get one for any given taxon from the OpenTree database using the Physcraper command line. If you already have a tree and an alignment of your own (or downloaded from somewhere else) that you want to update with Physcraper, please go to the Run section - Starting with your own tree.

5.4.1 Find a tree to update from OpenTree

The `find_trees.py` script searches for trees stored in OpenTree containing your taxon of interest.

Usage:

```
find_trees.py [-h] [-t TAXON_NAME] [-ott OTT_ID] [-tb] [-o OUTPUT]
```

Arguments:

For example, to find all trees in OpenTree that contain one or more members of the Malvaceae, the family of flowring plants encompassing cotton, cacao, and durian, among others, you can do:

```
find_trees.py -t Malvaceae -o malvacea.txt
```

If you happen to know the taxon OTT id of the Malvaceae, or you have already obtained it from the OpenTree website taxon homepage, you can do:

```
find_trees.py -ott 279960 -o malvacea.txt
```

5.4.2 Find a corresponding alignment on TreeBASE

To find trees with a corresponding alignment on TreeBASE, use the flag `-tb` or `--treebase`:

```
find_trees.py -t Malvaceae -tb -o malvacea.txt
```

5.5 How to run Physcraper

The easiest way to run Physcraper is using the command line tools. This way, you can directly specify arguments. A configuration file will be written down for the sake of reproducibility.

5.5.1 Example Physcraper runs from the command line

Starting with only an OpenTree study and tree id

As input, you will minimally need a study and tree ids from a tree uploaded to the OpenTree website (https://tree.opentreeoflife.org/curator). The `--treebase` flag (or `-tb`) will automatically download an alignment for that tree from TreeBASE.

```
physcraper_run.py [-s OPENTREE_STUDY_ID] [-t OPENTREE_TREE_ID] [-tb] [-o OUTPUT]
```

e.g.,

```
physcraper_run.py -s pg_55 -t tree5864 -tb -o pg_55_web
```

The output files generated by this example run are stored in "docs/examples/pg_55_web"

Starting with an OpenTree study and tree id and an alignment

Alternatively, you can provide the gene alignment that you want to update using the `-a` command:

```
physcraper_run.py [-s OPENTREE_STUDY_ID] [-t OPENTREE_TREE_ID] [-o OUTPUT] [-a␣
↪ALIGNMENT] [-as ALIGNMENT_SCHEMA]
```

For example, to update a tree from Crous et al. 2012 using an alignment already downloaded from TreeBASE, you can do:

```
physcraper_run.py -s ot_350 -t Tr53297 -a docs/examples/inputdata/ot_350Tr53297.aln -
↪as "nexus" -o ot_350
```

Starting with your own tree

If the tree you want to update is not posted to the OpenTree website, you need to match the labels on your tree to taxa using the OpenTree Bulk Taxonomic Name Resolution Service. Download your matched names, unzip the folder, and pass the "json" file that is output from the OpenTree Bulk TNRS tool as `--taxon_info` or `-ti` argument:

```
physcraper_run.py [-tf TREE_FILE] [-tfs TREEFILE_SCHEMA] [-a ALIGNMENT] [-as␣
↪ALIGNMENT_SCHEMA] [-ti TAXON_INFO_JSONFILE]  [-o OUTPUT]
```

e.g.,

```
physcraper_run.py -tf tests/data/tiny_test_example/test.tre -tfs newick -a tests/data/
↪tiny_test_example/test.fas  -as fasta --taxon_info tests/data/tiny_test_example/
↪main.json -o owndata
```

Checking the inputs before a full run

Use the flag `-no_est` to simply download a tree from OpenTree and the corresponding alignment from TreeBASE. This will not run the BLAST and tree estimation steps:

```
physcraper_run.py -s pg_55 -t tree5864 -tb -no_est -o pg55_C
```

To initiate a full Physcraper run from that tree and alignment, simply remove the `-no_est` flag. It will re-load the inputs from the specified output directory and will use your same config settings that are automatically written out to "OUTPUT_run.config".

The `-re` flag will re-run a Physcraper cycle on a given output directory. If the initial or previous run completed, it will use the final output tree and alignment as input. If the run was not completed, it will reload the original input files.

```
physcraper_run.py -re pg_55_C -o pg_55_C
```

You can also re-run with a different configuration file:

```
physcraper_run.py -re  pg_55_C/ -c alt_config -o  pg_55_D
```

5.5.2 Configuration parameters

To see all the configuration parameters, use `physcraper_run.py -h`.

The configuration parameters may be set in a configuration file, and then passed into the analysis run. See file "example.config" for an example.

-c CONFIGFILE, --configfile CONFIGFILE Gives the path to the configuration file

If a config file input is combined with command line configuration parameters, the command line values will override those in the configuration file.

The configuration settings for the current run are written to standard out, and saved in the output directory as "run.config", e.g.,

```
[blast]
Entrez.email = None
e_value_thresh = 1e-05
hitlist_size = 20
location = local
localblastdb = /home/projects/ncbi/localblastdb/
url_base = None
num_threads = 8
delay = 90
[physcraper]
spp_threshold = 3
seq_len_perc = 0.8
trim_perc = 0.8
min_len = 0.8
max_len = 1.2
taxonomy_path = /home/projects/physcraper/taxonomy
```

Input Data

Tree information (required):

-s STUDY_ID, --study_id STUDY_ID OpenTree study id

-t TREE_ID, --tree_id TREE_ID OpenTree tree id

OR

Alignment information (required):

-a ALIGNMENT, --alignment ALIGNMENT Gives the path to alignment file

-as ALN_SCHEMA, –aln_schema ALN_SCHEMA Specifies the alignment schema, one of nexus or fasta

OR

Tree and alignment information are required. After an analysis has been run, they can be reloaded from a directory from a previous run.

REQUIRED:

-o OUTPUT, --output OUTPUT Specifies the path to output directory

Optional:

Blast search parameters

You can use a local BLAST database. To setup see Local Databases section of this documentation.

You can use your own BLAST database, for example set up on an AWS server.

Sequence filtering parameters

Tree search parameters

Internal arguments

5.6 The Physcraper results

5.6.1 Files generated by a Physcraper run

Within the output directory defined by the user, a Physcraper run generates various subdirectories which are labeled with a TAG name that corresponds to the file name of the input alignment.

The subdirectories consist of:

- Input files

Within the `inputs_TAG` directory, Physcraper writes tree and alignment files used in a Physcraper run for the sake of reproducibility, taxon name matching, and taxon reconciliation. It also writes the ".config" file down if none was provided, as well as the results of the mapping of the tree taxon labels, saved as "otu_info.csv"

- Run files

Within the `run_TAG` directory, all run files are also automatically written down: Blast runs, alignments, RAxML trees, bootstrap results, etc.

Intermediate processing files, and the json formatted otu information are also stored here. Many fo these files are re-used in the event that the analysis crashes and is restarted. Make sure you use a new output directory or otherwise empty this folder if you want to modify the initial run parameters.

The trees are reconstructed using RAxML, with tip labels corresponding to local ids (e.g., otu42009, otuPS1) and not taxon names (e.g., *Ceiba*), nor taxonomic ids (e.g., ott or ncbi). Branch lengths are proportional to relative substitution rates.

- BLAST files

The `blast_TAG` folder contains XML files with BLAST results for each sequence in the input alignment.

- Output files

In the `output_TAG` folder, the updated tree with taxon names as tip labels is saved as "updated_taxonname.tre". A version of the updated tree in nexson format containing several types of tip labels is also saved here. From the nexson tree, a tree with any label can be produced. See section *relabeling the trees* below for more info on how to generate trees with different types of labels.

The "seqlen_mismatch.txt" file contains the acession numbers, taxa, and sequence lengths of BLAST matches that didn't meet the sequence length cutoffs.

Finally, the CSV file "otu_info_TAG.csv" contains a summary of information about original and newly added sequences.

5.6.2 Visualizing the Physcraper results

There are several tree visualization tools available. For reproducibility purposes, we will use some handy functions from the R language to see our results.

```
updated_tree_otus <- ape::read.tree(file = "docs/examples/pg_55/run_pg_55tree5864_
↪ndhf/RAxML_bestTree.2020-06-18")
ape::plot.phylo(ape::ladderize(updated_tree_otus), type = "phylogram", cex = 0.25,␣
↪label.offset = 0.001, edge.width = 0.5)
ape::add.scale.bar(cex = 0.3, font = 1, col = "black")
mtext("Updated tree - otu tags as labels", side = 3)
```

```
updated_tree_taxonname <- ape::read.tree(
  file = "docs/examples/pg_55/outputs_pg_55tree5864_ndhf/updated_taxonname.tre")
```

```
ape::plot.phylo(ape::ladderize(updated_tree_taxonname), type = "phylogram", cex = 0.2,
↪ label.offset = 0.001, edge.width = 0.5)
ape::add.scale.bar(cex = 0.3, font = 1, col = "black")
mtext("Updated tree - Taxon names as labels", side = 3)
```

Compare the original tree with the pruned updated tree

```
original_tree_otus <- ape::read.tree(file = "docs/examples/pg_55/inputs_pg_55tree5864_
↪ndhf/physcraper_pg_55tree5864_ndhf.tre")
updated_tree_otus_pruned <- ape::read.tree(
  file = "../data/pg_55/pruned_updated.tre"
)
```

Now plot them face to face.

We can prune the updated tree, so it is a straight forward comparison:

```
cotree <-  phytools::cophylo(original_tree_otus, updated_tree_otus_pruned, rotate.
↪multi =TRUE)
```

Rotating nodes to optimize matching... Done.

```
phytools::plot.cophylo(x = cotree, fsize = 0.5, lwd = 0.5, mar=c(.1,.2,2,.3), ylim=c(-
↪.1,1), scale.bar=c(0, .05))
title("Original tr.   Pruned updated tr.", cex = 0.1)
```

But it is more interesting to plot it with all the new tips, so we see exactly where the new things are:

```
original_tree_taxonname <- ape::read.tree(file = "docs/examples/pg_55/inputs_pg_
↪55tree5864_ndhf/taxonname.tre")
cotree2 <-  phytools::cophylo(
  original_tree_taxonname,
  updated_tree_taxonname,
  rotate.multi =TRUE)
```

Rotating nodes to optimize matching... Done.

```
phytools::plot.cophylo(
  x = cotree2,
  fsize = 0.3,
  lwd = 0.4,
  mar=c(.1,.1,2,.5),
```

(continues on next page)

(continued from previous page)

```
  ylim=c(-.1,1),
  scale.bar=c(0, .05),
  # link.type="curved",
  link.lwd=3,
  link.lty="solid",
  link.col=phytools::make.transparent("#8B008B",0.5))
title("Original tree      Updated tree", cex = 0.1)
```

We can also plot the updated tree against the synthetic subtree of Malvaceae, to visualize how it updates our current knowledge of the phylogeentic relationships within the family.

However, we are having some trouble with matching the tips right now! A fix will come soon:

```
tolsubtree <- rotl::tol_subtree(ott_id = 279960)
ape::Ntip(tolsubtree)
#> [1] 5898
grep("Pterygota_alata", tolsubtree$tip.label)
#> [1] 5714
updated_tree_taxonname$tip.label
#>  [1] "Fremontodendron_californicum_otuPS13"
#>  [2] "Quararibea_costaricensis_otuPS38"
#>  [3] "Matisia_cordata_otuPS39"
#>  [4] "Hibiscus_bojerianus_otuPS45"
#>  [5] "Macrostelia_laurina_otuPS29"
#>  [6] "Taliparitì_tiliaceum_var._tiliaceum_otuPS48"
#>  [7] "Taliparitì_hamabo_otuPS47"
#>  [8] "Papuodendron_lepidotum_otuPS46"
#>  [9] "Cephalohibiscus_peekelii_otuPS34"
#> [10] "Kokia_kauaiensis_otuPS32"
#> [11] "Kokia_drynarioides_otuPS31"
#> [12] "Kokia_cookei_otuPS30"
#> [13] "Ochroma_pyramidale_otuPS40"
#> [14] "Ochroma_pyramidale_otu376430"
#> [15] "Catostemma_fragrans_otuPS37"
#> [16] "Scleronema_micranthum_otuPS44"
#> [17] "Cavanillesia_platanifolia_otuPS50"
#> [18] "Spirotheca_rosea_otuPS42"
#> [19] "Spirotheca_rosea_otu376452"
#> [20] "Bombax_buonopozense_otuPS49"
#> [21] "Bombax_buonopozense_otu376420"
#> [22] "Ceiba_acuminata_otuPS36"
#> [23] "Ceiba_crispiflora_otuPS41"
#> [24] "Pachira_aquatica_otuPS35"
#> [25] "Pachira_aquatica_otu376439"
#> [26] "Septotheca_tessmannii_otuPS11"
#> [27] "Triplochiton_zambesiacus_otuPS43"
#> [28] "Heritiera_elata_otu376445"
#> [29] "Heritiera_littoralis_otu376454"
#> [30] "Heritiera_aurea_otu376446"
#> [31] "Heritiera_simplicifolia_otu376427"
#> [32] "Heritiera_aurea_otu376435"
#> [33] "Brachychiton_acerifolius_otu376453"
#> [34] "Brachychiton_acerifolius_otu376441"
#> [35] "Acropogon_bullatus_otu376442"
#> [36] "Acropogon_dzumacensis_otu376429"
#> [37] "Franciscodendron_laurifolium_otu376443"
#> [38] "Argyrodendron_peralatum_otu376431"
```

(continues on next page)

(continued from previous page)

```
#> [39] "Sterculia_balanghas_otu376450"
#> [40] "Sterculia_tragacantha_otu376428"
#> [41] "Sterculia_tragacantha_otuPS28"
#> [42] "Sterculia_stipulata_otu376440"
#> [43] "Sterculia_coccinea_otu376436"
#> [44] "Sterculia_parviflora_otu376444"
#> [45] "Hildegardia_barteri_otu376432"
#> [46] "Firmiana_malayana_otu376449"
#> [47] "Firmiana_platanifolia_otu376425"
#> [48] "Hildegardia_populifolia_otu376448"
#> [49] "Scaphium_linearicarpum_otu376434"
#> [50] "Scaphium_macropodum_otu376426"
#> [51] "Pterocymbium_tinctorium_otu376433"
#> [52] "Scaphium_macropodum_otu376451"
#> [53] "Octolobus_spectabilis_otu376447"
#> [54] "Cola_acuminata_otu376437"
#> [55] "Pterygota_alata_otu376438"
```

Trick for the cophylo titles and margins from https://cran.r-project.org/web/packages/phangorn/vignettes/IntertwiningTreesAndNetworks.html

5.6.3 Analysing the Physcraper results

Under construction

The functionalities described in the following sections are still under development. Even in this beta form, we think they have the potential to be useful, so we decided to document them here.

The first two functionalities are not accessible through the command line yet. This means that you can access them only through Python for the moment. The last functionality can be accessed through the command line, but it has not been fully tested yet. Use with caution.

Relabeling the trees

For downstream analyses and figure making, it can be handy to swap labels on tips of the updated phylogeny from alternative taxonomies or taxon id numbers.

Do that with the `write_labelled` function. For example, to change e.g.,

```
from physcraper import treetaxon
pg55 = treetaxon.generate_TreeTax_from_run('docs/examples/pg_55_web')
pg55.write_labelled(label='^ot:ottTaxonName', norepeats=False, path='tests/tmp/pg_55_
↪repeats.tre')
```

Rerooting the trees

A correctly rooted phylogeny is needed to compare relationships between two or more phylogenetic hypotheses. Automatic rooting of phylogenies is not straightforward. Physcraper's `root_tree_from_synth` function places a suggested root based on relationships in OpenTree's synthetic tree or in its taxonomic tree.

To root a Physcraper tree using either the OpenTree taxonomy, or the OpenTree synthetic tree. First load the tree object:

```
from physcraper import treetaxon
pg55 = treetaxon.generate_TreeTax_from_run('docs/examples/pg_55_web')
```

Then, to root based on the OpenTree taxonomy, set `base = "ott"`:

```
from physcraper import opentree_helpers
ott_rooted_tree = opentree_helpers.root_tree_from_synth(pg55.tre, pg55.otu_dict, base=
↪'ott')
pg55.tre = ott_rooted_tree  # set tree rooted based on taxonomy as the tree object
pg55.write_labelled(label="^ot:ottTaxonName", path="tests/tmp/pg_55_ott_root.tre")
```

And, to root based on phylogenetic relationships in the OpenTree synthetic tree, set `base = "synth"`:

```
from physcraper import opentree_helpers
synth_rooted_tree = opentree_helpers.root_tree_from_synth(pg55.tre, pg55.otu_dict,␣
↪base='synth')
pg55.tre = synth_rooted_tree  # set tree rooted based on synth as the tree object
pg55.write_labelled(label="^ot:ottTaxonName", path="tests/tmp/pg_55_synth_root.tre")
```

In this example both trees are the same even though they use the MRCA of different pairs of taxa, because those MRCA's map to the same node on the output tree.

However rooting based on OTT can be unreliable, especially if taxonomy is a poor fit to true evolutionary relationships. So whenever possible, the root should be specified by the user, for example by choosing from tips in the "otu_info.csv" file in the `outputs` folder of a Physcraper run, e.g.,

```
pg55 = treetaxon.generate_TreeTax_from_run('docs/examples/pg_55_web')
outgroup = ['otu376436','otu376444']
mrca = pg55.tre.mrca(taxon_labels=outgroup)
pg55.tre.reroot_at_node(mrca, update_bipartitions=True)
pg55.write_labelled(label="^ot:ottTaxonName", path="tests/tmp/pg_55_manual_root.tre")
```

Tree comparison with Robinson-Foulds (RF) distance

The `tree_comparison.py` script takes as an argument the output directory of a Physcraper run, and compares the relationships in the final tree to the relationships in the input tree.

Usage:

```
tree_comparison.py  [-h] [-d DIRECTORY_NAME] [-t1 FILE_NAME] [-t2 FILE_NAME] [-otu␣
↪FILE_NAME] [-og OUTGROUP] [-o DIRECTORY_NAME]
```

Arguments:

This is the simplest command line for comparison of two trees:

```
tree_comparison.py  [-h] [-d DIRECTORY_NAME] [-o DIRECTORY_NAME]
```

For example:

```
tree_comparison.py -d docs/examples/pg_55_web/ -o pg_55_comparison
```

It compares the original tree from the `inputs` folder and the updated tree from the `outputs` folder. It uses the rooting functions described above to ensure the two trees have the same root. By default, it will root trees based on the OpenTree Taxonomy.

Alternatively, you can pass in OpenTree taxonomic ids (OTT ids) of two or more taxa from the input tree to use as outgroups to root both trees.

For example:

```
tree_comparison.py -d docs/examples/pg_55_web/ -og otu376420 otu376439 otu376452 -o␣
↪pg_55_comparison
```

If the comparison between the two trees is possible (outgroup-wise), the script will print the results to screen, including:

- The number of new tips
- The number of new taxa
- Whether the taxa in the tree are included synthesis phylogenies currently in OpenTree
- Which taxa phylogenetic information is not currently incorporated into the synthetic tree
- The RF distance and weighted RF distance between the relationships of tips that are in both trees
- How the estimates of phylogenetic relationships of taxa included in the OpenTree taxonomy from both trees have conflict with the monophyly of the OpenTree synthetic tree.

5.7 How to combine analyses across multiple loci

Single locus analyses only provide a narrow view of the evolutionary history of a group.

After assembling individual gene data sets and phylogenies using Physcraper, it is straigtforward to combine results (alignments and data) from those analyses to obtain species tree estimates.

The `multi_loci.py` script combines results from multiple single locus Physcraper runs and generates concatenated and astral input files.

Usage:

```
multi_loci.py [-h] [-d MULTIPLE_RUNS_FOLDER] [-o OUTPUT] [-f {concatenate,astral}]
                  [-s {fasta,nexus}] [-m {False,True}]
```

Arguments:

5.7.1 Astral

To generate input files for an ASTRAL species tree analysis, (https://github.com/smirarab/ASTRAL) use -f astral. This will generate two files in the output directory. `genetrees.new`, a concatenation of all of the genetrees produced in individual analyses, and `mapping.txt`, a text file linking the tip lables in each of the gene trees to taxon names.

e.g.

```
multi_loci.py -d tests/data/precooked/multi_loc/ -f astral -o mini_species_tree
```

You can run Astral diretcly on these files e.g.

```
java -jar astral.5.7.5.jar -i mini_species_tree/genetrees.new -a mini_species_tree/
↪mappings.txt
```

5.7.2 Concatenation

To concatenate multiple loci into a single alignment use -f concatenate. Default settings only generate concatenated loci for taxa where there is a sequence at each locus .

e.g. multi_loci.py -d tests/data/precooked/multi_loc/ -f concatenate -s fasta -o mini_concat

To generate concatenated taxa with missing loci use -m (for include missing data).

```
multi_loci.py -d tests/data/precooked/multi_loc/ -f concatenate -s nexus -m -o mini_
↪concat_gaps
```

This will generate a concatenated alignment in the output directory with the name 'concat.aln' in the schema selected using -s (either fasta or nexus). Each concatenated sequences is labeled with the taxon name and an integer.

The sequences from each individual run comprising the concatenated sequence are described in "concat_info.txt" in the output directory.

5.7.3 SVD quartets

To write out a concatenated Nexus file with a taxon partitions block linking sequences for the same taxa, for use in SVD quartets analyses (tutorial at http://evomics.org/learning/phylogenetics/svdquartets/) use -f svdq

This will generate a Nexus file of concatenated sequences linked together by their taxon assignment in a taxon block. The sequences from each individual run comprising the concatenated sequence are described in "concat_info.txt" in the output directory, as above. e.g.

```
multi_loci.py -d tests/data/precooked/multi_loc/ -f svdq -m -o svdq_out
```

This file can be used to run SVDQ in Paup e.g.

```
paup4a168_ubuntu64 mini_concat2/svdq.nex
svdq evalq=all taxpartition=species nthreads=ncpus;
```

5.8 Physcraper use case examples

Interactive illustrated examples are available at the Physcraperex website for:

- The hollies, represented by the genus *Ilex*
- The family of cotton and chocolate, Malvaceae
- The Dothideomycetes, representing the most diverse class of ascomycete fungi

5.9 How does Physcraper work

5.10 Function Documentation

Physcraper module

The core blasting and new sequence integration module

class physcraper.scrape.**PhyscraperScrape**(*data_obj*, *ids_obj=None*, *search_taxon=None*)

This is the class that does the perpetual updating

To build the class the following is needed:

- **data_obj**: Object of class ATT (see above)
- **ids_obj**: Object of class IdDict (see above)

During the initializing process the following self.objects are generated:

- **self.workdir**: path to working directory retrieved from ATT object = data_obj.workdir
- **self.logfile**: path of logfile
- **self.data**: ATT object
- **self.ids**: IdDict object
- **self.config**: Config object
- **self.new_seqs**: dictionary that contains the newly found seq using blast:
  - key: gi id
  - value: corresponding seq
- **self.new_seqs_otu_id**: dictionary that contains

the new sequences that passed the remove_identical_seq() step:

- key: otu_id
- value: see otu_dict, is a subset of the otu_dict, all sequences that will be newly added to aln and tre

- **self.mrca_ncbi**: int ncbi identifier of mrca
- **self.blast_subdir**: path to folder that contains the files writen during blast
- **self.newseqs_file**: filename of files that contains the sequences from self.new_seqs_otu_id
- **self.date**: Date of the run - may lag behind real date!
- **self.repeat**: either 1 or 0, it is used to determine if we continue updating the tree,

no new seqs found = 0 * **self.newseqs_acc**: list of all gi_ids that were passed into remove_identical_seq(). Used to speed up adding process * **self.blocklist**: list of gi_id of sequences that shall not be added or need to be removed. Supplied by user. * **self.seq_filter**: list of words that may occur in otu_dict.status and which shall not be used in the building of FilterBlast.sp_d

(that's the main function), but it is also used as assert statement to make sure unwanted seqs are not added.

- **self.unpublished**: True/False. Used to look for local unpublished seq that shall be added if True.
- **self.path_to_local_seq:** Usually False, contains path to unpublished sequences if option is used.

Following functions are called during the init-process:

- **self.reset_markers()**: adds things to self: I think they are used to make sure certain function run,

if program crashed and pickle file is read in.

- self._blasted: 0/1, if run_blast_wrapper() was called, it is set to 1 for the round.
- self._blast_read: 0/1, if read_blast_wrapper() was called, it is set to 1 for the round.
- self._identical_removed: 0
- self._query_seqs_written: 0/1, if write_query_seqs() was called, it is set to 1 for the round.
- self._query_seqs_aligned: 0
- self._query_seqs_placed: 0/1, if place_query_seqs() was called, it is set to 1 for the round.
- self._reconciled: 0
- self._full_tree_est: 0/1, if est_full_tree() was called, it is set to 1 for the round.

align_new_seqs(*aligner='muscle'*)
Align the new sequences against each other

calculate_bootstrap(*alignment='default'*, *num_reps='100'*)
Calculates bootstrap and consensus trees.

-p: random seed -s: aln file -n: output fn -t: starting tree -b: bootstrap random seed -#: bootstrap stopping criteria -z: specifies file with multiple trees

calculate_final_tree(*boot_reps=100*)
Calculates the final tree using a trimmed alignment.

Returns final PS data

check_complement(*match*, *seq*, *gb_id*)
Double check if blast match is to sequence, complement or reverse complement, and return correct seq

est_full_tree(*alignment='default'*, *startingtree=None*)
Full RAxML run from the placement tree as starting tree. The PTHREAD version is the faster one, hopefully people install it if not it falls back to the normal RAxML.

filter_seqs(*tmp_dict*, *selection='random'*, *threshold=None*)
Subselect from sequences to a threshold of number of seqs per species

get_full_seq(*gb_id*, *blast_seq*)
Get full sequence from gb_acc that was retrieved via blast.

Currently only used for local searches, Genbank database sequences are retrieving them in batch mode, which is hopefully faster.

Parameters

- **gb_acc** – unique sequence identifier (often genbank accession number)
- **blast_seq** – sequence retrived by blast,

Returns full sequence, the whole submitted sequence, not only the part that matched the blast query sequence

make_sp_dict(*otu_list=None*)
Makes dict of OT_ids by species

map_taxa_to_ncbi()
Find NCBI ids for taxa from OpenTree

read_blast_wrapper(*blast_dir=None*)
reads in and processes the blast xml files

Parameters **blast_dir** – path to directory which contains blast files

Returns fills different dictionaries with information from blast files

read_local_blast_query(*fn_path*)
Implementation to read in results of local blast searches.

Parameters **fn_path** – path to file containing the local blast searches

Returns updated self.new_seqs and self.data.gb_dict dictionaries

read_webbased_blast_query(*fn_path*)
Implementation to read in results of web blast searches.

Parameters **fn_path** – path to file containing the local blast searches

Returns updated self.new_seqs and self.data.gb_dict dictionaries

remove_blocklistitem()
This removes items from aln, and tree, if the corresponding Genbank identifer were added to the blocklist.

Note, that seq that were not added because they were similar to the one being removed here, are lost (that should not be a major issue though, as in a new blast_run, new seqs from the taxon can be added.)

remove_identical_seqs()
goes through the new seqs pulled down, and removes ones that are shorter than LENGTH_THRESH percent of the orig seq lengths, and chooses the longer of two that are other wise identical, and puts them in a dict with new name as gi_ott_id.

replace_aln(*filename*, *schema='fasta'*)
Replace the alignment in the data object with the new alignment

replace_tre(*filename*, *schema='newick'*)
Replace the tree in the data object with the new tree

reset_markers()
set completion markers back to 0 for a re-run

run_blast_wrapper()
generates the blast queries and saves them depending on the blasting method to different file formats

It runs blast if the sequences was not blasted since the user defined threshold in the config file (delay).

Returns writes blast queries to file

run_local_blast_cmd(*query*, *taxon_label*, *fn_path*)
Contains the cmds used to run a local blast query, which is different from the web-queries.

Parameters

- **query** – query sequence
- **taxon_label** – corresponding taxon name for query sequence
- **fn_path** – path to output file for blast query result

Returns runs local blast query and writes it to file

run_muscle(*input_aln_path=None*, *new_seqs_path=None*, *outname='all_align'*)
Aligns the new sequences and the profile aligns to the exsiting alignment

run_web_blast_query(*query*, *equery*, *fn_path*)
Equivalent to run_local_blast_cmd() but for webqueries, that need to be implemented differently.

Parameters

- **query** – query sequence
- **equery** – method to limit blast query to mrca
- **fn_path** – path to output file for blast query result

Returns runs web blast query and writes it to file

select_seq_at_random(*otu_list*, *count*)
Selects sequences at random if there are more than the threshold.

seq_dict_build(*seq*, *new_otu_label*, *seq_dict*)
takes a sequence, a label (the otu_id) and a dictionary and adds the sequence to the dict only if it is not a subsequence of a sequence already in the dict. If the new sequence is a super sequence of one in the dict, it removes that sequence and replaces it

Parameters

- **seq** – sequence as string, which shall be compared to existing sequences
- **label** – otu_label of corresponding seq
- **seq_dict** – the tmp_dict generated in add_otu()

Returns updated seq_dict

summarize_boot(*besttreepath*, *bootpath*, *min_clade_freq=0.2*)
Summarize the bootstrap proportions onto the ML tree

write_mrca()
Write out search info to file

write_new_seqs(*filename='date'*)
writes out the query sequence file

physcraper.scrape.**debug**(*msg*)
short debugging command

physcraper.scrape.**set_verbose**()
Set output to verbose

AlignTreeTax: The core data object for Physcraper. Holds and links name spaces for a tree, an alignment, the taxa and their metadata.

class physcraper.aligntreetax.**AlignTreeTax**(*tree*, *otu_dict*, *alignment*, *search_taxon*, *workdir*, *configfile=None*, *tree_schema='newick'*, *aln_schema='fasta'*, *tag=None*)
Wrap up the key parts together, requires OTT_id, and names must already match. Hypothetically, all the keys in the otu_dict should be clean.

To build the class the following is needed:

- **newick**: dendropy.tre.as_string(schema=schema_trf) object
- **otu_dict**: json file including the otu_dict information generated earlier
- **alignment**: dendropy :class:‘DnaCharacterMatrix

<dendropy.datamodel.charmatrixmodel.DnaCharacterMatrix>‘ object * **search_taxon**: OToL identifier of the group of interest, either subclade as defined by user or of all tip labels in the phylogeny * **workdir**: the path to the corresponding working directory * **config_obj**: Config class * **schema**: optional argument to define tre file schema, if different from “newick”

During the initializing process the following self objects are generated:

- **self.aln**: contains the alignment and which will be updated during the run
- **self.tre**: contains the phylogeny, which will be updated during the run
- **self.otu_dict: dictionary with taxon information and physcraper relevant stuff**
  - key: otu_id, a unique identifier
  - **value: dictionary with the following key:values:**
    - ‘^ncbi:gi’: GenBank identifier - deprecated by Genbank - only older sequences will have it
    - ‘^ncbi:accession’: Genbanks accession number
    - ‘^ncbi:title’: title of Genbank sequence submission
    - ‘^ncbi:taxon’: ncbi taxon identifier
    - ‘^ot:ottId’: OToL taxon identifier
    - ‘^physcraper:status’: contains information if it

      was ‘original’, ‘queried’, ‘removed’, ‘added during filtering process’ * ‘^ot:ottTaxonName’: OToL taxon name * ‘^physcraper:last_blasted’: contains the date when the sequence was blasted. * ‘^user:TaxonName’: optional, user given label from OtuJsonDict * “^ot:originalLabel” optional, user given tip label of phylogeny
- **self.ps_otu**: iterator for new otu IDs, is used as key for self.otu_dict
- **self.workdir**: contains the path to the working directory, if folder does not exists it is generated.
- **self.mrca_ott**: OToL taxon Id for the most recent common ancestor of the ingroup
- **self.orig_seqlen**: list of the original sequence length of the input data
- **self.gi_dict**: dictionary, that has all information from sequences found during the blasting. * key: GenBank sequence identifier * value: dictionary, content depends on blast option, differs between webquery and local blast queries
  - **keys - value pairs for local blast:**
    - ‘^ncbi:gi’: GenBank sequence identifier
    - ‘accession’: GenBank accession number
    - ‘staxids’: Taxon identifier
    - ‘sscinames’: Taxon species name
    - ‘pident’: Blast percentage of identical matches
    - ‘evalue’: Blast e-value
    - ‘bitscore’: Blast bitscore, used for FilterBlast
    - ‘sseq’: corresponding sequence
    - ‘title’: title of Genbank sequence submission
  - **key - values for web-query:**

        * 'accession':Genbank accession number
        * 'length': length of sequence
        * 'title': string combination of hit_id and hit_def
        * 'hit_id': string combination of gi id and accession number
        * 'hsps': Bio.Blast.Record.HSP object
        * 'hit_def': title from GenBank sequence
      - **optional key - value pairs for unpublished option:**
        * 'localID': local sequence identifier
  - **self._reconciled**: True/False,
  - **self.unpubl_otu_json**: optional, will contain the OTU-dict for unpublished data, if that option is used

Following functions are called during the init-process:

  - **self._reconcile():** removes taxa, that are not found in both, the phylogeny and the aln
  - **self._reconcile_names():** is used for the own file stuff, it removes the character 'n' from tip names that start with a number

The physcraper class is then updating:

  - self.aln, self.tre and self.otu_dict, self.ps_otu, self.gi_dict

add_otu(*gb_id*, *ids_obj*)
Generates an otu_id for new sequences and adds them into self.otu_dict. Needs to be passed an IdDict to do the mapping.

Parameters

- **gb_id** – the Genbank identifier/ or local unpublished
- **ids_obj** – needs to IDs class to have access to the taxonomic information

Returns the unique otu_id - the key from self.otu_dict of the corresponding sequence

check_tre_in_aln()
Makes sure that everything which is in tre is also found in aln.

Extracted method from trim. Not sure we actually need it there.

get_otu_for_acc(*gb_id*)
A reverse search to find the unique OTU ID for a given accession number :param gb_id: the Genbank identifier

prune_short()
Prunes sequences from alignment if they are shorter than specified in the config file, or if tip is only present in tre.

Sometimes in the de-concatenating of the original alignment taxa with no sequence are generated or in general if certain sequences are really short. This removes those from both the tre and the alignment.

has test: test_prune_short.py

Returns prunes aln and tre

read_in_aln(*alignment*, *aln_schema*)
Reads in an alignment to the object taxon namespace.

read_in_tree(*tree*, *tree_schema=None*)
Imports a tree either from a file or a dendropy data object. Adds records in OTU dictionary if not already present.

remove_taxa_aln_tre(*taxon_label*)
Removes taxa from aln and tre and updates otu_dict, takes a single taxon_label as input.

note: has test, test_remove_taxa_aln_tre.py

Parameters **taxon_label** – taxon_label from dendropy object - aln or phy

Returns removes information/data from taxon_label

trim(*min_taxon_perc*)
It removes bases at the start and end of alignments, if they are represented by less than the value specified. E.g. 0.75 that 75% of the sequences need to have a base present.

Ensures, that not whole chromosomes get dragged in. It's cutting the ends of long sequences.

has test: test_trim.py

write_aln(*filename=None*, *alnschema='fasta'*, *direc='workdir'*)
Output alignment with unique otu ids as labels.

write_files(*treefilename=None*, *treeschema='newick'*, *alnfilename=None*, *alnschema='fasta'*, *direc='workdir'*)
Outputs both the streaming files, labeled with OTU ids. Can be mapped to original labels using otu_dict.json or otu_seq_info.csv

write_labelled(*label*, *filename='labelled'*, *direc='workdir'*, *norepeats=True*, *add_gb_id=False*)
Output tree and alignment with human readable labels. Jumps through a bunch of hoops to make labels unique.

NOT MEMORY EFFICIENT AT ALL

Has different options available for different desired outputs

Parameters

- **label** – which information shall be displayed in labelled files: possible options: '^ot:ottTaxonName', '^user:TaxonName', "^ot:originalLabel", "^ot:ottId", "^ncbi:taxon"
- **treepath** – optional: full file name (including path) for phylogeny
- **alnpath** – optional: full file name (including path) for alignment
- **norepeats** – optional: if there shall be no duplicate names in the labelled output files
- **add_gb_id** – optional, to supplement tiplabel with corresponding GenBank sequence identifier

Returns writes out labelled phylogeny and alignment to file

write_labelled_aln(*label*, *filename='labelled'*, *direc='workdir'*, *norepeats=True*, *add_gb_id=False*)
A wrapper for the write_labelled aln function to maintain older functionalities

write_labelled_tree(*label*, *filename='labelled'*, *direc='workdir'*, *norepeats=True*, *add_gb_id=False*)
A wrapper for the write_labelled tree function to maintain older functionalities

write_otus(*filename='otu_info'*, *schema='table'*, *direc='workdir'*)
Output all of the OTU information as either json or csv

write_papara_files(*treefilename='random_resolve.tre'*, *alnfilename='aln_ott.phy'*)
This writes out needed files for papara (except query sequences). Papara is finicky about trees and needs phylip format for the alignment.

NOTE: names for tree and aln files should not be changed, as they are hardcoded in align_query_seqs().

Is only used within func align_query_seqs.

write_random_resolve_tre(*treefilename='random_resolve.tre'*, *direc='workdir'*)
Randomly resolve polytomies, because some downstream approaches require that, e.g. Papara.

physcraper.aligntreetax.**generate_ATT_from_files**(*workdir*, *configfile*, *alnfile*, *aln_schema*, *treefile*, *otu_json*, *tree_schema*, *search_taxon=None*)
Build an ATT object without phylesystem, use your own files instead.

Spaces vs underscores kept being an issue, so all spaces are coerced to underscores when data are read in.

Note: has test -> test_owndata.py

Parameters

- **alnfile** – path to sequence alignment
- **aln_schema** – string containing format of sequence alignment
- **workdir** – path to working directory
- **config_obj** – config class including the settings
- **treefile** – path to phylogeny
- **otu_json** – path to json file containing the translation of tip names to taxon names, or to an otu_dictionary
- **tree_schema** – a string defining the format of the input phylogeny
- **search_taxon** – optional - OToL ID of the mrca of the clade of interest. If no search mrca ott_id is provided, will use all taxa in tree to calc mrca.

Returns object of class ATT

physcraper.aligntreetax.**generate_ATT_from_run**(*workdir*, *start_files='output'*, *tag=None*, *configfile=None*, *run=True*)
Build an ATT object without phylesystem, use your own files instead. :return: object of class ATT

physcraper.aligntreetax.**set_verbose**()
Set verbosity of outputs

physcraper.aligntreetax.**write_labelled_aln**(*aligntreetax*, *label*, *filepath*, *schema='fasta'*, *norepeats=True*, *add_gb_id=False*)
Output tree and alignment with human readable labels. Jumps through a bunch of hoops to make labels unique.

NOT MEMORY EFFICIENT AT ALL

Has different options available for different desired outputs.

Parameters

- **label** – which information shall be displayed in labelled files: possible options: '^ot:ottTaxonName', '^user:TaxonName', "^ot:originalLabel", "^ot:ottId", "^ncbi:taxon"
- **treepath** – optional: full file name (including path) for phylogeny
- **alnpath** – optional: full file name (including path) for alignment

- **norepeats** – optional: if there shall be no duplicate names in the labelled output files
- **add_gb_id** – optional, to supplement tiplabel with corresponding GenBank sequence identifier

Returns writes out labelled phylogeny and alignment to file

physcraper.aligntreetax.**write_labelled_tree**(*treetax*, *label*, *filepath*, *schema='newick'*, *norepeats=True*, *add_gb_id=False*)

Output tree and alignment with human readable labels. Jumps through a bunch of hoops to make labels unique.

NOT MEMORY EFFICIENT AT ALL

Has different options available for different desired outputs.

Parameters

- **label** – which information shall be displayed in labelled files: possible options: '^ot:ottTaxonName', '^user:TaxonName', "^ot:originalLabel", "^ot:ottId", "^ncbi:taxon"
- **treepath** – optional: full file name (including path) for phylogeny
- **alnpath** – optional: full file name (including path) for alignment
- **norepeats** – optional: if there shall be no duplicate names in the labelled output files
- **add_gb_id** – optional, to supplement tiplabel with corresponding GenBank sequence identifier

Returns writes out labelled phylogeny and alignment to file

physcraper.aligntreetax.**write_otu_file**(*treetax*, *filepath*, *schema='table'*)

Writes out OTU dict as json or table. :param treetax: eitehr a treetaxon object or an alignment tree taxon object :param filename: filename :param schema: either table or json format :return: writes out otu_dict to file

Linker Functions to get data from OpenTree

physcraper.opentree_helpers.**OtuJsonDict**(*id_to_spn*, *id_dict*)

Makes an OTU json dictionary, which is also produced within the openTreeLife-query.

This function is used, if files that shall be updated are not part of the OpenTreeofLife project. It reads in the file that contains the tip names and the corresponding species names. It then tries to get the unique identifier from the OpenTree project or from NCBI.

Reads input file into the var sp_info_dict, translates using an IdDict object using web to call OpenTree, then NCBI if not found.

Parameters

- **id_to_spn** – User file, that contains tip name and corresponding sp name for input files.
- **id_dict** – Uses the id_dict generated earlier

Returns dictionary with key: "otu_tiplabel" and value is another dict with the keys '^ncbi:taxon', '^ot:ottTaxonName', '^ot:ottId', '^ot:originalLabel', '^user:TaxonName', '^physcraper:status', '^physcraper:last_blasted'

physcraper.opentree_helpers.**bulk_tnrs_load**(*filename*)

Read in outputs from OpenTree Bulk TNRS, translates to a Physcraper OTU dictionary. :param filename: input json file

physcraper.opentree_helpers.**check_if_ottid_in_synth**(*ottid*)

Web call to check if OTT id in synthetic tree. NOT USED.

physcraper.opentree_helpers.**conflict_tree**(*inputtree*, *otu_dict*)
Write out a tree with labels that work for the OpenTree Conflict API

physcraper.opentree_helpers.**count_match_tree_to_aln**(*tree*, *dataset*)
Assess how many taxa match between multiple genes in an alignment data set and input tree.

physcraper.opentree_helpers.**debug**(*msg*)
short debugging command

physcraper.opentree_helpers.**deconcatenate_aln**(*aln_obj*, *filename*, *direc*)
Split out separate concatended alignments. NOT TESTED

physcraper.opentree_helpers.**generate_ATT_from_phylesystem**(*alnfile*, *aln_schema*, *workdir*, *configfile*, *study_id*, *tree_id*, *search_taxon=None*, *tip_label='^ot:originalLabel'*)
Gathers together tree, alignment, and study info; forces names to OTT ids.

Study and tree ID's can be obtained by using python ./scripts/find_trees.py LINEAGE_NAME

Spaces vs underscores kept being an issue, so all spaces are coerced to underscores when data are read in.

Parameters **aln** – dendropy :class:'DnaCharacterMatrix

<dendropy.datamodel.charmatrixmodel.DnaCharacterMatrix>' alignment object. :param workdir: Path to working directory. :param config_obj: Config class containing the settings. :param study_id: OpenTree study id of the phylogeny to update. :param tree_id: OpenTree tree id of the phylogeny to update, some studies have several phylogenies. :param phylesystem_loc: Access the GitHub version of the OpenTree data store, or a local clone. :param search_taxon: optional. OTT id of the MRCA of the clade that shall be updated. :return: Object of class ATT.

physcraper.opentree_helpers.**get_citations_from_json**(*synth_response*, *citations_file*)
Get ciattions for studies in an induced synthetic tree repsonse. :param synth_response: Web service call record :param citations_file: Output file

physcraper.opentree_helpers.**get_dataset_from_treebase**(*study_id*)
Given a phylogeny in OpenTree with mapped tip labels, this function gets an alignment from the corresponding study on TreeBASE, if available. By default, it first tries getting the alignment from the supertreebase repository at https://github.com/TreeBASE/supertreebase. If that fials, it tries getting the alignment directly form TreeBASE at https://treebase.org If both fail, it exits with a message.

physcraper.opentree_helpers.**get_max_match_aln**(*tree*, *dataset*, *min_match=3*)
Select an alignment from a DNA dataset

physcraper.opentree_helpers.**get_mrca_ott**(*ott_ids*)
Finds the MRCA of taxa in the ingroup of the original tree. The BLAST search later is limited to descendants of this MRCA according to the NCBI taxonomy.

Only used in the functions that generate the ATT object.

Parameters **ott_ids** – List of all OTT ids for tip labels in phylogeny

Returns OTT id of most recent common ancestor

physcraper.opentree_helpers.**get_nexson**(*study_id*)
Grabs nexson from phylesystem.

physcraper.opentree_helpers.**get_ott_taxon_info**(*spp_name*)
Get OTT id, taxon name, and NCBI id (if present) from the OpenTree Taxonomy. Only works with version 3 of OpenTree APIs

Parameters **spp_name** – Species name

Returns

physcraper.opentree_helpers.**get_ottid_from_gbifid**(*gbif_id*)
Returns a dictionary mapping GBIF ids to OTT ids. ott_id is set to 'None' if the GBIF id is not found in the Open Tree Taxanomy

physcraper.opentree_helpers.**get_tree_from_study**(*study_id*, *tree_id*, *label_format='ot:originallabel'*)
Create a dendropy Tree object from OpenTree data. :param study_id: OpenTree Study Id :param tree_id: OpenTree tree id :param label_format: One of 'id', 'name', "ot:originallabel", "ot:ottid", "ot:otttaxonname". defaults to "ot:originallabel"

physcraper.opentree_helpers.**get_tree_from_synth**(*ott_ids*, *label_format='name'*, *citation='cites.txt'*)
Wrapper for OT.synth_induced_tree that also pulls citations

physcraper.opentree_helpers.**ottids_in_synth**(*synthfile=None*)
Checks if OTT ids are present in current synthetic tree, using a file listing all current OTT ids in synth (v12.3) :param synthfile: defaults to taxonomy/ottids_in_synth.txt

physcraper.opentree_helpers.**root_tree_from_synth**(*tree*, *otu_dict*, *base='ott'*)
Uses information from OpenTree of Life to suggest a root. :param tree: dendropy `Tree :param otu_dict: a dictionary of tip label metadata, inculding an '^ot:ottId'attribute 'param base: either `synth or *ott.* If *synth* will use OpenTree synthetic tree relationships to root input tree, if *ott* will use OpenTree taxonomy.

physcraper.opentree_helpers.**scraper_from_opentree**(*study_id*, *tree_id*, *alnfile*, *workdir*, *aln_schema*, *configfile=None*)
Pull tree from OpenTree to create a physcraper object.

physcraper.opentree_helpers.**set_verbose**()
Set output verbosity

Physcraper run Configuration object generator

class physcraper.configobj.**ConfigObj**(*configfile=None*, *run=True*)
To build the class the following is needed:

- **configfi**: a configuration file in a specific format, e.g. to read in self.e_value_thresh.

During the initializing process the following self objects are generated:

- **self.e_value_thresh: the defined threshold for the e-value during Blast searches,** check out: https://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastDocs&DOC_TYPE=FAQ
- **self.hitlist_size**: the maximum number of sequences retrieved by a single blast search
- **self.minlen**: value from 0 to 1. Defines how much shorter new seq can be compared to input
- **self.trim_perc: value that determines how many seq need to be present before the beginning** and end of alignment will be trimmed
- **self.maxlen**: max length for values to add to aln
- **self.get_ncbi_taxonomy**: Path to sh file doing something...
- **self.ott_ncbi**: file containing OTT id, ncbi and taxon name (??)
- **self.email**: email address used for blast queries
- **self.blast_loc**: defines which blasting method to use:
  - either web-query (=remote)
  - from a local blast database (=local)

- **self.num_threads**: number of cores to be used during a run
- **self.url_base**:
  - if blastloc == remote: it defines the url for the blast queries.
  - if blastloc == local: url_base = None
- **self.delay**: defines when to reblast sequences in days
- **optional self.objects**:
  - if blastloc == local:
    * self.blastdb: this defines the path to the local blast database
    * self.ncbi_nodes: path to 'nodes.dmp' file, that contains the hierarchical information
    * self.ncbi_names: path to 'names.dmp' file, that contains the different ID's

check_taxonomy()
Locates a taxonomy directory in tthe phyysraper repo, or if not avail (often because module was pip installed), genertes one.

config_str()
Write out the current config values. DOES NOT INCUDE SOME HIDDEN CONFIGUREABLE ATTRIBUTES

read_config(*configfi*)
Reads configfile, and sets configuration params. any params not listed will be set to dafault values in set_default() * **configfile**: path to input file.

set_defaults()
In the absence of an input configuration file, sets default values.

set_local()
Checks that all appropriate files etc are in place for local blast db.

write_file(*direc*, *filename='run.config'*)
writes config params to file * **direc**: path to write file * **filename**: filename to use. Default = run.config

physcraper.configobj.**is_number**(*inputstr*)
Test if string can be coerced to float

Link together NCBI and Open Tree identifiers and names, with Gen Bank information for updated sequences

class physcraper.ids.**IdDicts**(*configfile=None*)
Class contains different taxonomic identifiers and helps to find the corresponding ids between ncbi and OToL

To build the class the following is needed:

- **config_obj**: Object of class config (see above)
- **workdir**: the path to the assigned working directory

During the initializing process the following self objects are generated:

- **self.workdir**: contains path of working directory
- **self.config**: contains the Config class object
- **self.ott_to_ncbi**: dictionary
  - key: OToL taxon identifier
  - value: ncbi taxon identifier
- **self.ncbi_to_ott**: dictionary

- key: OToL taxon identifier
- value: ncbi taxon identifier

- **self.ott_to_name**: dictionary
  - key: OToL taxon identifier
  - value: OToL taxon name
- **self.acc_ncbi_dict**: dictionary
  - key: Genbank identifier
  - value: ncbi taxon identifier
- **self.spn_to_ncbiid**: dictionary
  - key: OToL taxon name
  - value: ncbi taxon identifier
- **self.ncbiid_to_spn**: dictionary
  - key: ncbi taxon identifier
  - value: ncbi taxon name

user defined list of mrca OTT-ID's #TODO this is flipped form the dat aobj .ott_mrca. On purpose?

#reomved mrca's from ida, and put them into scrape object

- **Optional**:
  - depending on blasting method:
  - **self.ncbi_parser: for local blast,** initializes the ncbi_parser class, that contains information about rank and identifiers

entrez_efetch(*gb_id*)

Wrapper function around efetch from ncbi to get taxonomic information if everything else is failing. Also used when the local blast files have redundant information to access the taxon info of those sequences.

It adds information to various id_dicts.

Parameters **gb_id** – Genbank identifier

Returns read_handle

get_ncbiid_from_acc(*acc*)
checks local dicts, and then runs eftech to get ncbi id for accession

get_tax_seq_acc(*acc*)
Pulls the taxon ID and the full sequences from NCBI

uses ncbi databases to easily retrieve taxonomic information.

parts are altered from https://github.com/zyxue/ncbitax2lin/blob/master/ncbitax2lin.py

class physcraper.ncbi_data_parser.**Parser**(*names_file*, *nodes_file*)
Reads in databases from ncbi to connect species names with the taxonomic identifier and the corresponding hierarchical information. It provides a much faster way to get those information then using web queries. We use those files to get independent from web requests to find those information (the implementation of it in BioPython was not really reliable). Nodes includes the hierarchical information, names the scientific names and ID's. The files need to be updated regularly, best way to always do it when a new blast database was loaded.

get_downtorank_id(*tax_id*, *downtorank='species'*)
Recursive function to find the parent id of a taxon as defined by downtorank.

get_id_from_name(*tax_name*)
Find the ID for a given taxonomic name.

get_id_from_synonym(*tax_name*)
Find the ID for a given taxonomic name, which is not an accepted name.

get_name_from_id(*tax_id*)
Find the scientific name for a given ID.

get_rank(*tax_id*)
Get rank for given ncbi tax id.

match_id_to_mrca(*tax_id*, *mrca_id*)
Recursive function to find out if tax_id is part of mrca_id.

physcraper.ncbi_data_parser.**get_acc_from_blast**(*query_string*)
Get the accession number from a blast query. :param query_string: string that contains acc and gi from local blast query result :return: gb_acc

physcraper.ncbi_data_parser.**get_gi_from_blast**(*query_string*)
Get the gi number from a blast query. Get acc is more difficult now, as new seqs not always have gi number, then query changes.

If not available return None.

Parameters **query_string** – string that contains acc and gi from local blast query result

Returns gb_id if available

physcraper.ncbi_data_parser.**get_ncbi_tax_id**(*handle*)
Get the taxon ID from ncbi. ONly used for web queries

Parameters **handle** – NCBI read.handle

Returns ncbi_id

physcraper.ncbi_data_parser.**get_ncbi_tax_name**(*handle*)
Get the sp name from ncbi. Could be replaced by direct lookup to ott_ncbi.

Parameters **handle** – NCBI read.handle

Returns ncbi_spn

physcraper.ncbi_data_parser.**get_tax_info_from_acc**(*gb_id*, *ids_obj*)
takes an accession number and returns the ncbi_id and the taxon name

physcraper.ncbi_data_parser.**load_names**(*names_file*)
Loads names.dmp and converts it into a pandas.DataFrame. Includes only names which are accepted as scientific name by ncbi.

physcraper.ncbi_data_parser.**load_nodes**(*nodes_file*)
Loads nodes.dmp and converts it into a pandas.DataFrame. Contains the information about the taxonomic hierarchy of names.

physcraper.ncbi_data_parser.**load_synonyms**(*names_file*)
Loads names.dmp and converts it into a pandas.DataFrame. Includes only names which are viewed as synonym by ncbi.

physcraper.ncbi_data_parser.**strip**(*inputstr*)
Strips of blank characters from string in pd dataframe.

Work in progress to pull apart the linked tree and taxon objects from the alignemnt based ATT object

class physcraper.treetaxon.**TreeTax**(*otu_json*, *treefrom*, *schema='newick'*)
wrap up the key parts together, requires OTT_id, and names must already match.

write_labelled(*label*, *path*, *norepeats=True*, *add_gb_id=False*)
output tree and alignment with human readable labels Jumps through a bunch of hoops to make labels unique.

NOT MEMORY EFFICIENT AT ALL

Has different options available for different desired outputs

Parameters

- **label** – which information shall be displayed in labelled files: possible options: '^ot:ottTaxonName', '^user:TaxonName', "^ot:originalLabel", "^ot:ottId", "^ncbi:taxon"
- **treepath** – optional: full file name (including path) for phylogeny
- **alnpath** – optional: full file name (including path) for alignment
- **norepeats** – optional: if there shall be no duplicate names in the labelled output files
- **add_gb_id** – optional, to supplement tiplabel with corresponding GenBank sequence identifier

Returns writes out labelled phylogeny and alignment to file

physcraper.treetaxon.**generate_TreeTax_from_run**(*workdir*, *start_files='output'*, *tag=None*)
Build an Tree + Taxon object from the outputs of a run. :return: object of class TreeTax

5.11 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. (This page modified from https://github.com/pyOpenSci/cookiecutter-pyopensci)

You can contribute in many ways:

5.11.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/McTavishLab/physcraper/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

Physcraper could always use more documentation, whether as part of the official Physcraper docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/McTavishLab/physcraper/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.11.2 Get Started!

Ready to contribute? Here's how to set up Physcraper for local development.

(1) Fork the Physcraper repo on GitHub https://github.com/McTavishLab/physcraper

(2) Clone your fork locally:

```
$ git clone git@github.com:your_name_here/physcraper.git
```

(3) Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv venv-physcraper
$ cd physcraper/
$ pip install -e .
```

(4) Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

(5) When you're done making changes, check that your changes pass flake8 and the

tests, including testing other Python versions with tox:

```
$ pytest tests
```

(6) Use Pylint to check your code. Move to the "bin" or "physcraper" directory to use the `.pylintrc` config file, then run:

```
$ pylint insert_name_of_module_here.py
```

(7) Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

(8) Submit a pull request through the GitHub website.

Extra: Count the number of functions in any given module

```
from inspect import getmembers, isfunction
foos = [o for o in getmembers(physcraper) if isfunction(o[1])]
len(foos)
```

5.11.3 Updating the README

The `README.Rmd` file generates the `README.md` file, which in turn generates what is shown on Physcraper's home page at its GitHub repository, and PyPI's description.

To update `README.md` from `README.Rmd` file, you need R and the `rmarkdown` package installed to run:

```
R -e 'rmarkdown::render("README.Rmd")'
```

The `index.rst` file that lives in the `docs/source/` folder controls the home page at readthedocs, which is updated automatically as you push to GitHub.

To update any of these, you have to modify, as needed, `README.Rmd` and `docs/source/index.rst`, as well as the following `.md` files living in the docs/mds/ folder:

- intro-badges.md
- intro-logo.md
- intro-part1.md
- citation.md
- license.md
- contact.md
- updating-the-readme.md, *aka, this file*

To create new sections, you just need to create new `.md` files in `docs/mds/` and make sure to add them to `README.Rmd` and `docs/source/index.rst`

5.11.4 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in "README.rst"
3. The pull request should work for Python 3.6, 3.7 and 3.8, and for PyPy. Check https://travis-ci.org/McTavishLab/physcraper/pull_requests and make sure that the tests pass for all supported Python versions.

5.11.5 Code of Conduct

Please note that the Physcraper project is released with a Contributor Code of Conduct. By contributing to this project you agree to abide by its terms.

5.12 Credits

5.12.1 Citation

If you use Physcraper, please cite:

- Sánchez-Reyes, L.L., M. Kandziora, & E.J McTavish. (2021). *Physcraper: a Python package for continually updated phylogenetic trees using the Open Tree of Life*. BMC Bioinformatics 22, 355. doi: doi.org/10.1186/s12859-021-04274-6.
- Open Tree of Life, B. Redelings, L.L. Sanchez Reyes, K.A. Cranston, J. Allman, M.T. Holder, & E.J. McTavish. (2019). *Open Tree of Life Synthetic Tree (Version 12.3)*. Zenodo. doi: 10.5281/zenodo.3937741

5.12.2 License

Physcraper is made available through the GNU General Public License v3.0

5.12.3 Development Lead

Emily Jane McTavish

https://github.com/snacktavish

ejmctavish@ucmerced.edu

5.12.4 Coauthors

Luna Luisa Sanchez Reyes

https://github.com/LunaSare

Martha Kandziora

https://github.com/mkandziora

5.12.5 Contributors

Matthias Bussonnier

https://github.com/Carreau

5.13 Changelog

All notable changes to the Physcraper project are documented here.

5.13.1 next

- progress bars
- aminoacid alignments
- cleaner screen print

5.13.2 0.6.0

- add citation to README
- add links to installation instructions and demonstration usage (documentation and examples) to README
- Pylinting code
- Multilocus script
- Trapping supertreebase error
- Trapping DendroPy error when reading in alignments with alternative states

5.13.3 0.5.0

- New contributor Luna L. Sanchez Reyes
- Fixing lack of ingroup
- Adding documentation
- Rerooting functions

5.14 FAQs

5.14.1 Frequently asked questions

How does Physcraper handle paralogs?

Physcraper allows multiple tips to be mapped to the same taxon, with an added unique identifier that allows linking the sequence back to the original one on GenBank. This means that newly added homologous DNA sequences can include both orthologs and paralogs and that a phylogenetic analysis can be performed. While we expect that most curated alignments are composed of ortholog sequences, the algorithm used to find new sequences is unable to distinguish paralogs, so it is likely that – if existing, they will be automatically added to the dataset. Users should check the output phylogenetic trees to detect paralogs and filter them appropriately if needed.

I have to learn to use OpenTree to use Physcraper, *is the learning curve worth it*?

We think that this decision depends on the goals of the user.

To our knowledge, existing tools that automatize dataset construction for phylogenetics focus on assembling an alignment *de novo* and/or mining and filtering homolog sequences. The main goal of Physcraper is to construct upon the knowledge contained in existing expertly-curated and peer-reviewed homology hypothesis (alignments), as well as establish a framework for interoperability between biological databases and phylogenetic knowledge. To achieve that, understanding the OpenTree Taxonomy and associated tools is key.

In this sense Physcraper offers the unique advantage to automatically connect a phylogenetic tree to other databases and services, including alignment databases such as TreeBASE, and the conflicting service provided by OpenTree, which permits to rapidly identify regions in the tree that:

- have been enriched in a tree with phylogenetic information,
- are coherent with other phylogenetic estimates, as well as
- conflict with other phylogenetic estimates.

Users can update phylogenetic trees using any other existing tool. If they want to connect phylogenetic data to other biological databases, particularly in a reproducible way, matching their taxon names to the OpenTree Taxonomy allows them to do this programmatically and automatically, instead of by hand.

For this goal, having a minimum familiarity with the OpenTree tools is needed.

We realize that this might initially discourage some users, but we believe that the benefits brought by connecting taxonomic data with the OpenTree services will encourage users to familiarize with the OpenTree services, and to adopt the use of Physcraper.

How does Physcraper handle polytomies in a starting tree?

The Physcraper starting tree is a phylogeny whose tip labels must have been standardized to the OpenTree Taxonomy (as described in the Introduction section: Mapping names to taxa). Original tip labels of the starting tree must be identical to taxon labels on the starting alignment. However, not all taxon labels in the alignment have to be present in the tree and visceversa.

Physcraper makes use of the starting tree in four main ways:

1. to delimit a taxon for the GenBank search (a search taxon),
2. to be used as starting tree for the phylogenetic reconstruction software of choice,
3. to standardize the taxon names from the starting alignment, and
4. to compare the updated phylogenetic relationships with the original ones.

Physcraper does not really "handle" polytomies. The goal of the software is to use the existing phylogenetic information that has been generated, reviewed, published and curated by experts in the field.

If a starting tree contains polytomies, these can only affect the outcome of the analysis if the starting tree is used for the case (1) delimiting a taxon for the GenBank search. To delimit the search taxon from the starting tree, a known outgroup is necessary. The outgroup can be user defined. If the outgroup is not defined by the user, Physcraper will attempt to root the starting tree following the OpenTree Taxonomy. If succesful, it will take the tip labels from the earliest diverging branch with the least number of tips. These will be used as outgroup. However, if the starting tree has polytomies around the early diverging branches, the automatic rooting is problematic and can have multiple solutions.

How does Physcraper use the starting alignment?

Physcraper uses all unique DNA sequences in the input alignment to mine a genetic database using the BLAST algorithm, with the goal of increasing the lineage sampling of the alignment within a given biological group.

Python Module Index

p

Index

I

L

M

O

P

R

S

T

W