physcraper Documentation

McTavish Lab

Dec 01, 2018

Contents:

CHAPTER 1

API Documentation

Physcraper module

class physcraper.**AlignTreeTax**(*newick*, *otu_dict*, *alignment*, *ingroup_mrca*, *workdir*, *schema=None*, *taxon_namespace=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 `DnaCharacterMatrix` object
- **ingroup_mrca**: OToL identifier of the group of interest, either subclade as defined by user or of all tiplabels in the phylogeny
- **workdir**: the path to the corresponding working directory
- **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: a unique identifier (otu plus either "tiplabel of phylogeny" or for newly found sequences PS_number.
  - 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.

          If the year is different from the 20th century, it tells us something about the initial status: * 1800 = never blasted, not yet considered to be added * 1900 = never blasted and not added - see status for more information * this century = blasted and added.
        * '^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.ott_mrca**: 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_names():** removes taxa, that are not found in both, the phylogeny and the aln and changes their names????

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

dump(*filename=None*)
writes pickled files from att class

prune_short(*min_seqlen_perc=0.75*)
Prunes sequences from alignment if they are shorter than 75%, 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

Parameters **min_seqlen_perc** – minimum length of seq

Returns prunes aln and tre

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(*taxon_missingness=0.75*)
It removes bases at the start and end of alignments, if they are represented by less than 75% of the sequences in the alignment. Ensures, that not whole chromosomes get dragged in. It's cutting the ends of long sequences.

Used in prune_short() has test: test_trim.py

Parameters **taxon_missingness** – defines how many sequences need to have a base at the start/end of an alignment

write_files(*treepath='physcraper.tre'*, *treeschema='newick'*, *alnpath='physcraper.fas'*, *alnschema='fasta'*)
Outputs both the streaming files and a ditechecked

write_labelled(*label*, *treepath=None*, *alnpath=None*, *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 filenname (including path) for phylogeny
- **alnpath** – optional: full filenname (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_otus(*filename*, *schema='table'*)

Writes out OTU dict as json.

Parameters

- **filename** – filename
- **schema** – either table or json format

Returns writes out otu_dict to file

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.

Is only used within func align_query_seqs.

class physcraper.**ConfigObj**(*configfi*, *interactive=None*)

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.

  The file needs to have a heading of the format: [blast] and then somewhere below that heading a string e_value_thresh = value

- **interactive**: defaults to True, is used to interactively update the local blast databases

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.seq_len_perc**: value from 0 to 1. Defines how much shorter new seq can be compared to input
- **self.get_ncbi_taxonomy**: Path to sh file doing something...
- **self.ncbi_dmp**: path to file that has gi numbers and the corresponding ncbi tax id's
- **self.phylesystem_loc**: defines which phylesystem for OpenTree datastore is used. The default is api, but can run on local version too.
- **self.ott_ncbi**: file containing OTT id, ncbi and taxon name (??)
- **self.id_pickle**: path to pickle file
- **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.unmapped**: used for OToL original tips that can not be assigned to a taxon
  - keep: keep the unmapped taxa and asign them to life
  - remove: remove the unmapped taxa from aln and tre
- **optional self.objects**:
  - if blastloc == local:
    * self.blastdb: this defines the path to the local blast database
    * self.ncbi_parser_nodes_fn: path to 'nodes.dmp' file, that contains the hierarchical information
    * self.ncbi_parser_names_fn: path to 'names.dmp' file, that contains the different ID's

class physcraper.**FilterBlast**(*data_obj*, *ids*, *settings=None*)

Takes the Physcraper Superclass and filters the ncbi blast results to only include a subset of the sequences.

They can be filtered by number or by rank and number. The feature can be useful for non-population-level studies, e.g. analyses which require a single representative per taxon (threshold = 1) or to check the monophyly of taxa without having to deal with over-representation of few taxa (e.g. threshold = 4, which allows to get a good overview of what is available without having some taxa being represented by high numbers of sequences). The second option (downtorank), which is optional, allows to filter according to taxonomic levels, e.g. getting a number of representative sequences for a genus or lineage it can also be used to deal with subspecies.

Existing self objects are:

self.sp_d: dictionary

key = species name/id value = list with otu_dict entry

self.sp_seq_d: dictionary

key = species name/id value = dictionary (Is overwritten every 'round')

key = otuID value = seq.

self.filtered_seq: dictionary. Is used as the self.new_seqs equivalent from Physcraper, just with fewer seqs. Is overwritten every 'round'

key = otuID, val = seq.

self.downtorank: optional string defining the level of taxonomic filtering, e.g. "species", "genus"

add_all(*key*)

It adds all seq to filtered_seq dict as the number of sequences present is smaller than the threshold value.

It is only used, when sequences selection happens via blasting.

Note: has test, test_add_all.py

Parameters **key** – key of self.sp_d (taxon name)

Returns self.filtered_seq

add_setting_to_self(*downtorank*, *threshold*)
Add FilterBlast items to self.

Parameters

- **downtorank** – rank which defines your level of Filtering
- **threshold** – number, defining how many seq per rank do you want to keep

Returns

count_num_seq(*tax_id*)
Counts how many sequences there are for a tax_name, excluding sequences that have not been added during earlier cycles.

Function is only used in how_many_sp_to_keep().

Parameters **tax_id** – key from self.sp_seq_d

Returns dict which contains information of how many seq are already present in aln, how many new seq have been found and if the taxon is a new taxon or if seq are already present

get_name_for_blastfiles(*key*)
Gets the name which is needed to write/read the blast files in 'loop_for_write_blast files'.

The name needs to be retrieved before the actual loop starts. I use the taxonomic names here, as this is the measure of which information goes into which local filter blast database. The function is only used within 'loop_for_write_blast_files' to generate the filenames.

Parameters **key** – key of self.sp_d (taxon name)

Returns name used for blast file

how_many_sp_to_keep(*threshold*, *selectby*)
Uses the sp_seq_d and places the number of sequences according to threshold into the self.filterdseq_dict.

This is essentially the key function of the Filter-class, it wraps up everything.

Parameters

- **threshold** – threshold value, defined in input
- **selectby** – mode of sequence selection, defined in input

Returns nothing specific, it is the function, that completes the self.filtered_seq, which contains the filtered sequences that shall be added.

loop_for_write_blast_files(*key*)
This loop is needed to be able to write the local blast files for the filtering step correctly.

Function returns a filename for the filter blast, which were generated with 'get_name_for_blastfiles()'.

Note: has test,test_loop_for_blast.py

Parameters **key** – key of self.sp_d (taxon name)

Returns name of the blast file

make_sp_seq_dict()
Uses the sp_d to make a dict with species names as key1, key2 is gb_id/sp.name and value is seq

This is used to select representatives during the filtering step, where it selects how many sequences per species to keep in the alignment. It will only contain sp that were not removed in an earlier cycle of the program.

Note: has test, test_sp_seq_d.py

return: self.sp_seq_d

replace_new_seq()
Function to replace self.new_seqs and self.new_seqs_otu_id with the subset of filtered sequences.

This is the final step in the FilterBlast class, from here it goes back to PhyScraper.

Returns subsets of self.new_seqs and self.new_seqs_otu_id

select_seq_by_length(*taxon_id*, *threshold*, *count*)
This is another mode to filter the sequences, if there are more than the threshold.

This one selects new sequences by length instead of by score values. It is selected by "selectby='length'". Count is the return value from self.count_num_seq(tax_id)["seq_present"], that tells the program how many sequences for the taxon are already available in the aln.

Parameters

- **taxon_id** – key from self.sp_seq_d
- **threshold** – threshold - max number of sequences added per taxon - defined in input
- **count** – self.count_num_seq(tax_id)["seq_present"]

Returns self.filtered_seq

select_seq_by_local_blast(*seq_d*, *fn*, *threshold*, *count*)
Selects number of sequences from local_blast to fill up sequences to the threshold. Count is the return value from self.count_num_seq(tax_id)["seq_present"], that tells the program how many sequences for the taxon are already available in the aln.

It will only include species which have a blast score of mean plus/minus sd. Uses the information returned by read_local_blast_query() to select which sequences shall be added in a filtered run.

Note: has test,test_select_seq_by_local_blast.py

Parameters

- **seq_d** – is the value of self.sp_d (= another dict)
- **fn** – refers to a filename to find the local blast file produced before, which needs to be read in by read_local_blast_query()
- **threshold** – threshold
- **count** – self.count_num_seq(tax_id)["seq_present"]

Returns self.filtered_seq

sp_dict(*downtorank=None*)
Takes the information from the Physcraper otu_dict and makes a dict with species name as key and the corresponding seq information from aln and blast seq, it returns self.sp_d.

This is generated to make information for the filtering class more easily available. self.sp_d sums up which information are available per taxonomic concept and which have not already been removed during either the remove_identical_seq steps or during a filtering run of an earlier cycle.

Note: has test, test_sp_d.py

Parameters **downtorank** – string defining the level of taxonomic filtering, e.g. "species", "genus"

Returns self.sp_d

write_otu_info(*downtorank=None*)
Writes different output tables to file: Makes reading important information less code heavy.

1. table with taxon names and sampling.
2. a file with all relevant GenBank info to file (otu_dict).

It uses the self.sp_d to get sampling information, that's why the downtorank is required.

Parameters **downtorank** – hierarchical filter

Returns writes output to file

class physcraper.**IdDicts**(*config_obj*, *workdir*, *mrca=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
- **mrca**: mrca as defined by input, can be a class

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
- **self.mrca_ott**: user defined list of mrca OTT-ID's
- **self.mrca_ncbi**: set, which is fed by self.get_ncbi_mrca()
- **Optional**:
  - depending on blasting method:

- self.ncbi_parser: for local blast, initializes the ncbi_parser class, that contains information about rank and identifiers
- self.otu_rank: for remote blast to store the rank information

find_name(*sp_dict=None*, *acc=None*)
Find the taxon name in the sp_dict (= otu_dict entry) or of a Genbank accession number. If not already known it will ask ncbi using the accession number

Parameters

- **sp_dict** – otu_dict entry
- **acc** – Genbank accession number

Returns ncbi taxon id

get_ncbi_mrca()
get the ncbi tax ids from a list of mrca.

get_ncbiid_from_tax_name(*tax_name*)
Get the ncbi_id from the species name using ncbi web query.

Parameters **tax_name** – species name

Returns corresponding ncbi id

get_rank_info_from_web(*taxon_name*)
Collects rank and lineage information from ncbi, used to delimit the sequences from blast, when you have a local blast database or a Filter Blast run

map_acc_ncbi(*gb_id*)
get the ncbi taxon id's for a Genbank identifier input.

Finds different identifiers and information from a given gb_id and fills the corresponding self.objects with the retrieved information.

Parameters **gb_id** – Genbank ID to

Returns ncbi taxon id

ott_id_to_ncbiid(*ott_id*)
Find ncbi Id for ott id. Is only used for the mrca list thing.

physcraper.**OtuJsonDict**(*id_to_spn*, *id_dict*)
Make otu json dict, which is also produced within the openTreeLife-query reads input file into the var sp_info_dict, translates using an IdDict object using web to call Open tree, then ncbi if not found.

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 tipnames and the corresponding species names. It then tries to get the different identifier from the OToL project or if not from ncbi.

Parameters

- **id_to_spn** – user file, that contains tipname and corresponding sp name for input files.
- **id_dict** – uses the id_dict generates 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'

class physcraper.**PhyscraperScrape**(*data_obj*, *ids_obj*)
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.otu_by_gi**: dictionary that contains ????:
  - key:
  - value:
- **self._to_be_pruned**: list that contains ????
- **self.mrca_ncbi**: ncbi identifier of mrca
- **self.tmpfi**: path to a file or folder???
- **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.blacklist**: list of gi_id of sequences that shall not be added or need to be removed. Supplied by user.
- **self.acc_list_mrca**: list of all gi_ids available on GenBank for a given mrca. Used to limit possible seq to add.
- **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.

- **self.OToL_unmapped_tips()**: function that either removes or maps unmapped taxa from OToL studies

OToL_unmapped_tips()
Assign names or remove tips from aln and tre that were not mapped during initiation of ATT class.

align_query_seqs(*papara_runname='extended'*)
runs papara on the tree, the alignment and the new query sequences

Parameters **papara_runname** – possible file extension name for papara

Returns writes out files after papara run/aligning seqs

calculate_bootstrap()
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

dump(*filename=None*)
writes out class to pickle file

Parameters **filename** – optional filename

Returns writes out file

est_full_tree()
Full raxml run from the placement tree as starting tree

find_otudict_gi()
Used to find seqs that were added twice. Debugging function.

generate_streamed_alignment()
runs the key steps and then replaces the tree and alignment with the expanded ones

get_sp_id_of_otulabel(*label*)
Get the species name and the corresponding ncbi id of the otu.

Parameters **label** – otu_label = key from otu_dict

Returns ncbi id of corresponding label

local_blast_for_unpublished(*query*, *taxon*)
Run a local blast search if the data is unpublished.

Parameters

- **query** – query sequence
- **taxon** – taxon.label used as identifier for the sequences

Returns xml files with the results of the local blast

place_query_seqs()
runs raxml on the tree, and the combined alignment including the new query seqs. Just for placement, to use as starting tree.

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_unpublished_blast_query()
Reads in the blast files generated during local_blast_for_unpublished() and adds seq to self.data.gb_dict and self.new_seqs.

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_alien_aln_tre()
Sometimes there were alien entries in self.tre and self.aln.

This function ensures they are properly removed.

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

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.

run_blast_wrapper(*delay=14*)
generates the blast queries and saves them depending on the blasting method to different file formats

Parameters **delay** – number that determines when a previously blasted sequence is reblasted - time is in days

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_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

seq_dict_build(*seq*, *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

write_query_seqs()
writes out the query sequence file

write_unpubl_blastdb(*path_to_local_seq*)
Adds local sequences into a local blast database, which then can be used to blast aln seq against it and adds sequences that were found to be similar to input. If this option is used, it queries against local database first and only in "2" round it goes back to blasting against GenBank

Parameters **path_to_local_seq** – path to the local seqs that shall be added

Returns writes local blast databases for the local sequences

class physcraper.**Settings**(*seqaln*, *mattype*, *trfn*, *schema_trf*, *workdir*, *threshold=None*, *selectby=None*, *downtorank=None*, *spInfoDict=None*, *add_unpubl_seq=None*, *id_to_spn_addseq_json=None*, *configfi=None*, *blacklist=None*, *shared_blast_folder=None*, *delay=None*, *trim=None*)
A class to store all settings for PhyScraper.

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

physcraper.**deep_debug**(*msg*)
short debugging command

physcraper.**generate_ATT_from_files**(*seqaln*, *mattype*, *workdir*, *treefile*, *otu_json*, *schema_trf*, *ingroup_mrca=None*)
Build an ATT object without phylesystem.

If no ingroup mrca ott_id is provided, will use all taxa in tree to calc mrca. otu_json should encode the taxon names for each tip.

Note: has test -> owndata.py

Parameters

- **seqaln** – path to sequence alignment
- **mattype** – string containing format of sequence alignment
- **workdir** – path to working directory
- **treefile** – path to phylogeny
- **otu_json** – path to jsonfile containing the translation of tipnames to taxon names
- **schema_trf** – string defining the format of the input phylogeny
- **ingroup_mrca** – optional - OToL ID of the mrca of the clade of interest

Returns object of class ATT

physcraper.**generate_ATT_from_phylesystem**(*aln*, *workdir*, *study_id*, *tree_id*, *phylesystem_loc='api'*, *ingroup_mrca=None*)

gathers together tree, alignment, and study info - forces names to otu_ids. Outputs AlignTreeTax object.

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

Input can be either a study ID and tree ID from OpenTree # TODO: According to code it cannot be either, but must be both

Alignment need to be a Dendropy DNA character matrix!

Parameters

- **aln** – dendropy alignment object
- **workdir** – path to working directory
- **study_id** – OToL study id of the corresponding phylogeny which shall be updated
- **tree_id** – OToL corresponding tree ID as some studies have several phylogenies
- **phylesystem_loc** – access the github version of the OpenTree data store, or a local clone
- **ingroup_mrca** – OToL identifier of the mrca of the clade that shall be updated (can be subset of the phylogeny)

Returns object of class ATT

physcraper.**get_dataset_from_treebase**(*study_id*, *phylesystem_loc='api'*)

Function is used to get the aln from treebase, for a tree that OpenTree has the mapped tree.

physcraper.**get_mrca_ott**(*ott_ids*)

finds the mrca of the taxa in the ingroup of the original tree. The blast search later is limited to descendants of this mrca according to the ncbi taxonomy

Used in the functions that generate the ATT object.

Parameters **ott_ids** – list of all OToL identifiers for tiplabels in phylogeny

Returns OToL identifier of most recent common ancestor or ott_ids

physcraper.**get_ncbi_tax_id**(*handle*)

Get the taxon ID from ncbi.

Parameters **handle** – NCBI read.handle

Returns ncbi_id

physcraper.**get_ncbi_tax_name**(*handle*)
Get the sp name from ncbi.

Parameters **handle** – NCBI read.handle

Returns ncbi_spn

physcraper.**get_nexson**(*study_id*, *phylesystem_loc*)
Grabs nexson from phylesystem

physcraper.**get_ott_ids_from_otu_dict**(*otu_dict*)
Get the ott ids from an otu dict object

physcraper.**get_ott_taxon_info**(*spp_name*)
get ottid, taxon name, and ncbid (if present) from Open Tree Taxonomy. ONLY works with version 3 of Open tree APIs

Parameters **spp_name** – species name

Returns

physcraper.**get_raw_input**()
Asks for yes or no user input.

Returns user input

physcraper.**is_number**(*s*)
test if string can be coerced to float

physcraper.**standardize_label**(*item*)
Make sure that the tipnames are unicode.

Function is only used if own files are used for the OtuJsonDict() function.

Parameters **item** – original tipname

Returns tipname in unicode

CHAPTER 2

README

CHAPTER 3

physcraper

Continual gene tree updating. Uses a tree from Open tree of Life (or your own tree) and an alignment to search for and adds homologous sequences to phylogenetic inference. The tool is under current development in the McTavish Lab. Please contact ejmctavish, gmail if you need any help!

3.1 Requirements

see also here

3.1.1 Dependencies (need to be in path):

- PaPaRa http://sco.h-its.org/exelixis/web/software/papara/index.html
- Raxml http://sco.h-its.org/exelixis/web/software/raxml/index.html
- BLAST+ https://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastDocs&DOC_TYPE=Download

3.1.2 Python packages:

These will all be installed if you install physcraper using python setup.py install

(but note, if you are using virtualenv there are some weird interactions with setuptools and python 2.7.6)

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

3.1.3 Databases

The tool uses several databases, which can automatically be downloaded/updated from ncbi. If the tool wants to access the site, you will be asked for input (yes or no), as the tool will then access ncbi, which is a US government website!

3.2 Tutorial

For a description of which settings need to be changed and how to set-up a run, see here.

3.3 Example runs and datasets

There is a full example python script with comments in `docs/example.py`. Some more example files can be found in `docs/example_scripts/`.

If you want to try running physcraper use the testdata which is in `tests/data/tiny_test_example/`

More information will follow soon.

3.4 Documentation

The Documentation about the different classes can be found here.

3.5 Tests

There are some tests here and here, which test the major functionality of the code. If you want to test if the code works on your machine, please run `python tests/testfilesetup.py` and then `sh tests/run_test.sh`, `sh ws-tests/run_ws-tests.sh`.

CHAPTER 4

How to start

4.1 Short introduction into what the tool actually does

PhyScraper is a command-line tool written in python to automatically update phylogenies. As input it needs a phylogeny, the corresponding alignment and the information about the tip names and the corresponding species names. This can either be a file provided by the user or if you have uploaded your tree to Open Tree of Life the corresponding study ID. PhyScraper will take every input sequence and blasts it against the ncbi GenBank database. Sequences that are similar to the input sequence will be added to the alignment, if they are a different species and/or they are longer than existing sequences or differ in at least one point mutation. Then it will place the newly found sequences onto the tree, which is then used as a starting tree for a full RAxML run. In the next round, every newly added sequence will be blasted and this continues until no new sequence were found. After a certain time threshold (currently 14 days), the existing sequences will be blasted again to check if new sequences can be found. After the single-gene datasets are updated, the data can be concatenated. Either, the user specifies which sequences are combined or the tool decides randomly which sequences to combine if there are more than a single sequence for a taxon in one of the alignments.

4.2 Short tutorial:

4.2.1 Before you can start

1. install the dependencies:

- PaPaRa - alignment tool
- RAxML - tree estimation program
  - Make sure you do `make -f Makefile.PTHREADS.gcc` from within the RAxML folder to enable multi-core calculation
- BLAST+ - it's needed for filter runs and when using local BLAST databses.

make sure the programms are accessible from everywhere, thus add them to your PATH using the command line:

- UNIX: `export PATH=$PATH:/path/to/my/program`

- Windows: `set PATH=%PATH%;C:\path\to\my\program`
- MAC: `export PATH=$PATH:~/path/to/program`

(! set PATH=%PATH%: it takes the current path and sets PATH to it.)

2. download PhyScraper using the command line:

- as a normal package: `git clone https://github.com/McTavishLab/physcraper.git`
- as a git repository: `git clone 'git@github.com:McTavishLab/physcraper.git'`

3. install python requirements and dependencies:

run from within the physcraper main folder:

- `python setup.py install`
- `pip install -r requirements.txt`

4. decide for a BLASTing method:

Depending on the size of your tree to be updated, there are things to consider.

- **web BLAST service**: If the tree is not too large and/or you have enough time, you can run the tool with the main settings, that uses the web BLAST service. The web service is not intended for large amounts of queries and if there are too many searchs being submitted by a user, the searches are being slowed down. Another down side is, that the species name retrieval can be difficult sometimes. Advantage is that it is the most up to date database to blast against.
- **Amazon cloud service**: If you do not have a fast computer, there are options to pay for a pre-installed cloud service using amazon.
- **local blast database**: This is the **recommended method**, as it is the fastest and does not heavily depend on good internet connection. Especially, if the trees are bigger and/or you have a relatively fast computer, this might be the best option. Ncbi regularly publishes the databases, that can easily be downloaded and initiated.
  - Install a local Blast database:

    General information about the BLAST database can be found here: ftp://ftp.ncbi.nlm.nih.gov/blast/documents/blastdb.html.

    In Linux to install the BLAST database do the following (for Windows and MAC please use google to figure it out, there should be plenty of information.):

    * `open a terminal`
    * `cd /to/the/folder/of/your/future/blastdb`
    * `sudo apt-get install ncbi-blast+` # if not already installed earlier
    * `wget 'ftp://ftp.ncbi.nlm.nih.gov/blast/db/nt.*'` # this downloads all nt-compressed files
    * `update_blastdb nt`
    * `cat *.tar.gz | tar -xvzf - -i` # macOS `tar` does not support the `-i` flag, you need to use homebrew to `brew install gnu-tar` and replace the `tar` command by `gtar`

    * `blastdbcmd -db nt -info`

      The last command shows you if it worked correctly. 'nt' means, we are making the nucleotide database. The database needs to be update regularly, the program will check the dates of your databases and will ask you to update the databases after 60 days. If your databases are older, you will be asked for input, if you want to update the databases. Interactive input does not work on remote machines, to stop the program from asking, change the following line in your analysis file from `conf = ConfigObj(configfi)` to `conf = ConfigObj(configfi, interactive=False)`. If you want to update the databases earlier go back to step 1.

- install the taxonomy database:

  install ncbi taxonomy database to retrieve taxon information from BLAST searches into the same directory as your blastdb from the step before.

  * `cd /to/the/folder/of/your/blastdb`
  * `wget 'ftp://ftp.ncbi.nlm.nih.gov/blast/db/taxdb.tar.gz'` # Download the taxdb archive
  * `gunzip -cd taxdb.tar.gz | (tar xvf - )` # Install it in the BLASTDB directory

- install the taxonomic rank database:

  * `wget 'ftp://ftp.ncbi.nlm.nih.gov/pub/taxonomy/taxdump.tar.gz'`
  * `gunzip -cd taxdump.tar.gz | (tar xvf - names.dmp nodes.dmp)`
  * move files into `tests/data/`

- updating the databases:

  The databases need to be update regularly, the program will check the dates of your databases and will ask you to update the databases after 60 days. If your databases are older, you will be asked for input, if you want to update the databases. Interactive input does not work on remote machines, to stop the program from asking, change the following line in your analysis file from `conf = ConfigObj(configfi)` to `conf = ConfigObj(configfi, interactive=False)`.

  If you want to update the databases earlier:

```
* blast db: repeat the steps listed under 'Install a local Blast database'
* taxonomy db: run `update_blastdb taxdb`
* rank db: repeat the steps listed under 'install the taxonomic rank database'
```

4.2.2 Set up a run

1. edit major settings in the config file

There is an example config file in `tests/data/localblast.config`

- BLAST settings:
  - **email**: Please specify your email address, this is recommended/required by ncbi.
  - **e_value_thresh**: This is the e-value that can be retrieved from BLAST searches and is used to limit the BLAST results to sequences that are similar to the search input sequence. It is a parameter that describes how many hits can be expected by chance from a similar-sized database during BLAST searches. Small e-value indicate a significant match. In general, shorter sequences have lower e-values, because shorter sequences have a higher probability to occur in the database by chance. For more information please

refer to the ncbi BLAST resources (e.g. url{https://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastDocs}). We used an e-value of 0.001 for all example datasets.

- **unmapped**: It is used when working with trees from OToL. Sometimes not all sequences of the study were mapped to OToL species identifiers.
  - **keep**: Unmapped tips will be kept and the is set to unknown_NAME_OF_THE_MRCA.
  - **remove**: Unmapped tips will be removed from the phylogeny and the alignment.
- **hitlist_size**: This specifies the amount of sequences being returned from a BLAST search. If your phylogeny does not contain a lot of nodes, it can be a low value, which will speed-up the analysis. If the sampled lineage contains many sequences with low sequence divergence it is better to increase it to be able to retrieve all similar sequences. It is not advised to have a really low hitlist size, as this will influence the number of sequences that will be added to your alignment. Low hitlist sizes might not return all best-matches but only the first 10 even though there might be more best-matches in the database citep{shah_misunderstood-2018}. Furthermore, for example, if the hitlist size is 10, but the phylogeny which shall be updated is sparsely sampled, this might result in an updated phylogeny, that has only the parts of the phylogeny updated, that were present in the input phylogeny. Lineages that were not present might never be added, as the 10 best hits all belong to the lineages already present.
- **location**: This defines which kind of BLAST service is used
  - **remote**: either ncbi (default) or amazon cloud service (website needs to be defined using `url_base`)
  - **local**: will look for a local database under the path specified under localblastdb, `num_threads` defines how many cores shall be used for the blasting (the more the faster).
- **gb_id_filename**: If you plan to run different settings for the same phylogeny or several runs with similar phylogenies, where there might be overlap between BLAST searches, set it to true und specify the blast-subdir to be equal among runs (see next section). This will share BLAST searches between runs and thus speeds up the run time of the BLAST search.

- ncbi_parser settings: Only need to be specified for local blast searches
  - **nodes_fn**: Path to the nodes file, which was downloaded as part of the local blast installation.
  - **names_fn**: Path to the names file, which was downloaded as part of the local blast installation.
- Physcraper settings:
  - **seq_len_perc**: Here you can specify the minimum percentage length of newly found sequences to be added in comparison to the original alignment.

2. write your analysis file

1. standard run

   This is explaining how to set up a "standard run", which will add all sequences, that are similar and long enough to the alignment, as long as they are no subsequences of an already existing sequence. Optional arguments are explained in the following section.

   Depending if you have uploaded your tree prior to analysis to the OpenTreeofLife website (OToL), there are two main options:

   Specified paths have to start either from your root directory (e.g. `/home/USER/physcraper/path/to/file`), or can be relative from within the physcraper main folder (`./path/to/file`).

   (a) using OpenTreeOfLife study ID:

   There is an example file in `docs/example.py` it is based on the wrapper function `standard_run()`

To obtain the study and tree ID's for an OToL run, either go to the website and query your lineage or you can run `find_studies.py` by typing in the terminal `python ./path/to/file/find_studies.py LINEAGENAME`. It will give you a studyID and a treeID, if there is a study available.

```
   * **study_id**: the ID of the corresponding study from OToL
   * **tree_id**: the ID of the corresponding tree from OToL
   * **seqaln**: give the path to your alignment file, must be a single gene␣
↪alignment
   * **mattype**: file format of your alignment - currently supported: "fasta",␣
↪"newick", "nexus", "nexml", "phylip"
   * **workdir**: path to your working directory, the folder where the␣
↪intermediate and result files shall be stored.
   * **configfi**: path to your config-file, which edited in step 1.
   * **otu_jsonfi**: path to the otu json file, this will contain all the␣
↪information of the sequences retrieved during the run. Usually, does not need␣
↪to be edited.


b) using your own files:

  There is an example file in ``tests/tiny_standard_ownfile.py``\ , it comes with␣
↪a tiny sample dataset in ``tests/data/tiny_example``. The corresponding wrapper␣
↪function to use in your file setup is ``own_data_run()``.
```

```
* **seqaln**: give the path to your alignment file, must be a single gene␣
↪alignment
* **mattype**: file format of your alignment - currently supported: "fasta",␣
↪"newick", "nexus", "nexml", "phylip"
* **trfn**: give the path to the file containing the corresponding phylogeny, all␣
↪tips must be represented in the alignment file as well.
* **schema_trf**: file format of your phylogeny file - currently supported:␣
↪"fasta", "newick", "nexus", "nexml", "phylip"
* **id_to_spn**: path to a comma-delimited file where tip labels correspond to␣
↪species names: example file can be found in `tests/data/tiny_test_example/test_
↪nicespl.csv`
* **workdir**: path to your working directory, the folder where intermediate and␣
↪result files shall be stored.
* **configfi**: path to your config-file, which was edited in step 1.
* **otu_jsonfi**: path to the otu json file, this will contain all the␣
↪information of the sequences retrieved during the run. Usually, does not need␣
↪to be edited.
```

2. filter run:

   This explains how to set up a filtered run. Here, the sequences retrieved from the BLAST search/during a standard run, are being filtered according to some user input. This is particularly of use, if there is no need to have every single sequence available being represented in your phylogeny.

   Beside the standard definition, there are more input options. Currently supported are:

- **threshold**: This defines the maximum number of sequences per taxon (e.g. species) to be retrieved.

  If your input dataset already contains more sequences, there will be no additional sequences added, but also not removed. (If the removal of sequences that were already part of the initial phylogeny is a function someone would like to have, this should be easy to implement. Just ask.)

- **downtorank**: This defines the rank which is used to determine the maximum number of sequences per

taxon.

It can be set to None and then for all taxons, there will be the maximum number of threshold sequences retrieved. If it is set to species, there will no more than the maximum number of sequences randomly choosen from all sequences available for all the subspecies. It can be set to any ranks defined in the ncbi taxonomy browser.

- **selectby**: This defines how to select the representative sequences.
  - **blast**: All sequences belonging to a taxon will be used for a filtering blast search.

    A sequence already present in the phylogeny, or a randomly chosen sequence, will be used to blast against all other sequences from the locus with the same taxon name. From the sequences that pass the filtering criterium, sequences will be randomly selected as representative. The filtering criterium is that they need to be within the mean +/- standard deviation of sequence similarity in relation to the queried sequence. See below for the explanation of the similarity value.

    If the taxon is likely monophyletic the distances will be similar and thus all sequences will fall within the mean and standard deviation of sequence similarity. If there are a few outlier sequences only, this seems to be likely a misidentification or mis-labeling in GenBank, outlier sequences will not be added, as they are most likely outside the allowed range of mean +/- SD. If the taxon is likely not monophyletic and sequences diverge a lot from each other, the mean and SD will be larger and allows to randomly pick sequences, that represent the divergence.

    As value for sequence similarity, we use bit-scores. Bit-scores are log-scaled scores and a score is a numerical value that describes the overall quality of an alignment (thus from the blasted sequence against the other available sequences). Higher numbers correspond to higher similarity. While scores are depending on database size, the rescaled bit-scores do not. Check out https://www.ncbi.nlm.nih.gov/BLAST/tutorial/Altschul-1.html for more detail.
  - **length** Instead of randomly choosing between sequences that are within the criteria of the blast search using sequence divergence as a criteria, here the longest sequences will be selected.
- **blacklist**: a list of sequences, that shall not be added or were identified to be removed later. This needs to be formatted as a python list containing the GenBank identifiers (e.g. `[gi number, gi number]`). Please not, that it must be the Genbank identifiers and not the accession numbers.

(a) using OpenTreeofLife study ID: There is an example file in `tests/filter_OTOL.py`. The corresponding function to use in your file setup is `filter_OTOL()`.

(b) using your own files: There is an example file in `tests/tiny_filter_ownfile.py`. The corresponding function to use in your file setup is `filter_data_run()`.

3. Use a local folder as sequence database:

   Instead of using GenBank as the source of new sequences, we can specify a folder which contains sequences in fasta format and this folder will be used as a sequence database. Then before running a standard or filter run, sequences from that folder can be added to the alignment/phylogeny if the folder contains sequences that are similar to the sequences already present in the alignment. This is intended to be used for newly sequenced material, which is not yet published on GenBank. To use this you need to specify:

```
* add_unpubl_seq = path to folder with the sequences
* id_to_spn_addseq_json = path to file which translates the sequences names to␣
↪species names
```

3. start to update your phylogeny:

This should be straight forward - type in your physcraper main folder:

```
python ./path/to/file/analysis-file.py
```

And now you just need to wait...

4. more hidden features that can be changed:

There are some more features that can be changed if you know where, we will change the code hopefully soon, to make that easier adjustable.

- time lapse for blasting: at the moment this is set to be 14 days. If you want to adjust the timing change `run_blast_wrapper()` in the wrapper to `run_blast_wrapper(delay = your_value)`
- trim method: by default sequences will be trimmed from the alignment if it has not at least 75% of the total sequence length. This can be changed in `./physcraper/__init__.py`, in the function `trim()` the value for `taxon_missingness`.
- change the most recent common ancestor (mrca): often phylogenies include outgroups, and someone might not be interested in updating that part of the tree. This can be avoided by defining the most recent common ancestor. It requires the OpenTreeOfLife identifier for the group of interest.

  You can get that ID by two different approaches:

  1. run `python scripts/get_ottid.py name_of_your_ingroup`
  2. by going to Open Tree of Life and type in the name of the lineage and get the OTT ID at the right side of the page. That number needs to be provided analysis file, as following:

     The identifying number need to be entered here:
  3. in an OToL run: within the function `standard_run()`/`filter_OTOL()` in your analysis file in the field for `ingroup_mrca`.
  4. in an own data run: provide ID within the function `own_data_run()`/`filter_data_run()` in your analysis file in the field for `ingroup_mrca`.

     Another aspect which needs to be considered, if your group of interest is not monophyletic and you limit the search to the mrca of the group, closely related sequences that belong for example to a different genus will not be added.
- sharing blast result files across runs:
  1. give the path to the folder in the wrapper function of your analysis file.
  2. in your config file: change the gb_id_filename setting to True.

  Be careful! If you have different hitlist_size defined, your blast files have different numbers of sequences saved. Sharing the folder across those different settings is not recommended!

5. Concatenate different single-gene PhyScraper runs:

After the single-gene PhyScraper runs were updated, the data can be combined, see for example `tests/data/concat_runs.py`. Either the program randomly decides which sequences to concatenate if there are more sequences available for one loci or the user can specify a file, which sequences shall be concatenated. An example file can be found at `tests/data/concatenation_input.csv`.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

R

S

T

W