Bipype Documentation

Release 1.0

Marian Siwiak, Albert Bogdanowicz, Agnieszka Hajduk, Michał Kr

December 04, 2016

Contents

Contents:

CHAPTER 1

Code

1.1 Bipype

1.1.1 Code documentation

parse_arguments(*args=None*)

1.2 Metatranscriptomics bipype

1.2.1 Module functions

auto_tax_read(*db_loc*)
Reads pickled `{KEGG GENES number:  set[KO identifiers]}` dict.

config_from_file(*_file*)
Reads parameters from configuration *_file*. Prepares target.txt and templates for SARTools.

Parameters **_file** – configuration file for metatranscriptomic pipeline

Returns

- ref_cond: reference condition defined by user
- all_conds: set of conditions (groups) from target.txt
- fastqs: list of fastq files on which analysis will be done

Return type (ref_cond, all_conds, fastqs)

connect_db(*db*)
Connects database

Parameters **db** – Path to SQL database

Returns Cursor object to database

dicto_reduce(*present*, *oversized*)
Removes all elements from dictionaries, which keys aren't present in both.

Parameters

- **present** – dict
- **oversized** – dict

Returns tuple of dicts

Return type (oversized, present)

> **Warning:** Order of parametres is opposite to results.

Example

```
>>> dict_1={'a':1,'c':3,'d':4}
>>> dict_2={'a':3,'b':4,'c':4}
>>> dicto_reduce(dict_1, dict_2)
({'a': 3, 'c': 4}, {'a': 1, 'c': 3})
```

fastq_to_fasta (*fastq*)
Runs fastq_to_fasta on fastq.

GLOBAL:

- path to fastq_to_fasta program: PATH_FQ2FA

get_kegg_name (*ko*)
Returns name assigned to given KO identifier (from kegg.jp)

Parameters **ko** – KO identifier (string)

Returns name assigned to ko (string)

get_ko_fc (*ko_dict*, *ref_cond*, *filepath*, *deseq=False*)
From given table file (SARTool), adds found fold changes to ko_dict.

Parameters

- **ko_dict** – `{KO_id:{cond1:value1, cond2:value2...}...}` dict
- **ref_cond** – reference condition (string)
- **filepath** – filepath to output table file from edgeR or DESeq2
- **deseq** – True, if filepath points to DESeq2 table file False, if filepath points to edgeR table file

Returns ko_dict with added fold changes from table file

get_kopathways (*database*)
Makes dictionaries from kopathways table from SQLite3 database.

Parameters **database** – Cursor object to SQLite3 database.

Returns

id -> pathways:

```
{KO identifier: set[KEGG_Pathway_ids]}
```

For example:

```
{
    'K01194': set(['ko00500','ko00600',...]),
    'K04501': set(['ko04390',...])
}
```

pathway -> ids mappings:

```
{KEGG_Pathway_id: set[KO identifiers]}
```

For example:

```
{ko12345: set([K12345, K12346,...]),...}
```

Return type Two dictionaries

get_pathways(*database*)
Make dictionary from pathways table from SQLite3 database.

Parameters **database** – Cursor object to SQLite3 database.

Returns

dictionary in following format:

```
{KEGG_Pathway_id:Name}
```

For example:

```
{
    'ko04060': 'Cytokine-cytokine receptor interaction',
    'ko00910': 'Nitrogen metabolism'
}
```

Return type dict

get_tables(*database*)
Prints all tables included in SQLite3 database.

Parameters **database** – Cursor object to SQLite3 database.

low_change(*ko_dict*, *all_conds*)
For every KO adds condition: 0, if condition is missing.

Parameters

- **ko_dict** – {KO_id:{cond1:value1, cond2:value2...}...} dict
- **all_conds** – list of conditions (list of strings)

Returns

suplemented ko_dict

For example:

```
low_change(
    {
        'K12345': {'pH5': 1.41, 'pH6': 1.73},
        'K23456': {'pH6': 2.0, 'pH8': 2.24}
    },
    ['pH5', 'pH6', 'pH8']
)
```

gives:

```
            {
                'K12345': {'pH5': 1.41, 'pH6': 1.73, 'pH8': 0.0},
                'K23456': {'pH5': 0.0, 'pH6': 2.0, 'pH8': 2.24}
            }
```

m8_to_ko (*file_*, *multi_id*)

Assigns and counts KEGG GENES identifiers from BLAST Tabular (flag: -m 8) output format file, for every KO from multi_id.

After mapping, writes data to output file.

Parameters

- **file_** – Path to BLAST Tabular (flag: -m 8) format file
- **multi_id** – Dict `{KEGG GENES identifier : set[KO identifiers]}`

Output file (outname) has following name:

```
outname = file_.replace('txt.m8', 'count')
```

and following format:

```
K00161  2
K00627  0
K00382  11
```

mapper (*ko_dict*, *ko_set*)

Assings every KO_id from ko_dict to KEGG_Pathway_id from *ko_set*

Parameters

- **ko_dict** – `{KO_id:{cond1:value1, cond2:value2...}...}` dict
- **ko_set** – `{KEGG_Pathway_id:set[KO identifiers]}` dict

Returns

Dict with structure:

```
{KEGG_Pathway_id:{KO_id:{cond1:value1, cond2:value2...}...}...}
```

Return type dict

mapper_write (*ko_path_dict*, *all_conds*, *out_dir*)

Writes file with KO and corresponding fold change, for every combination of condition & KEGG_Pathway_id.

Parameters

- **ko_path_dict** – `{KEGG_Pathway_id:{KO_id:{cond1:value1, cond2:value2...}...}...}`
- **all_conds** – list of conditions (list of strings)
- **out_dir** – relative output directory path

Output file has following path:

```
out_dir/condX/
        , following name:
KEGG_Pathway_id.txt
        , following header:
# KO KEGG_Pathway_id
```

```
            & following format:
    KO_id corresponding_fold_change
```

metatranscriptomics (*opts*)
Performs analyse of metagenomic data.

See also:

For more information please refer to:

- *run_fastq_to_fasta()*
- *run_rapsearch()*
- *run_ko_map()*
- *run_SARTools()*
- *run_pre_ko_remap()*
- *run_ko_remap()*
- *run_new_ko_remap()*
- *run_ko_csv()*

out_content (*filelist*, *kopath_values*, *path_names*, *method='DESeq2'*)
For every item in 'kopath_values' dictionary and for every file in 'filelist', writes to output file line with KOs, which are common for item.value and the set of KOs obtained from file.

Parameters

- **filelist** – List of paths to tab-delimited .txt files, where first column is a KO identifier.
- **kopath_values** – `{KEGG_Pathway_id:set[KO identifiers]}` dict.

  For example:

```
  {ko12345:set([K12345, K12346,...]),...}
```

- **path_names** – Dictionary in `{KEGG_Pathway_id:Name}` format.

  For example:

```
  {
      'ko04060': 'Cytokine-cytokine receptor interaction',
      'ko00910': 'Nitrogen metabolism'
  }
```

- **method** – Argument used only as a part of output file name

Output file has following name:

```
    (method+'_'+filename.replace('txt', 'path_counts.csv'))
where:
    filename = filepath.split('')[-1], if '' in filepath.
    filename = filepath.split('/')[-1],  if '/' in filepath.
    filename = filepath,                 in other cases.
```

anf following headline:

```
ko_path_id;ko_path_name;percent common;common KOs
```

Writes only lines with non-zero common KOs.

pickle_or_db(*pickle*, *db*)
Reads pickle or SQL database, than makes a dict.

If appropriate pickle (a dict) is available, it is read. In the other case function reads 'kogenes' table from SQL database and makes missing pickle. Eventually returns dict.

Parameters

- **pickle** – Path to pickled dict in following format: `{KEGG GENES identifier : set[KO identifiers]}`
- **db** – Cursor object to SQL database with 'kogenes' table `(KO identifier KEGG GENES identifier)`

Returns Dict in `{KEGG GENES identifier: set[KO identifiers]}` format.

Some information for Bipype's developers (delete this before final version): Code from this fuction was not a fuction in previous version and 'args' was hardcoded to: 'kogenes.pckl' & c (variable with db's cursor)

progress(*what*, *estimated_percentage=None*, *done=True*)
Prints specially formatted information about progress.

Parameters

- **what** – a string with name of operation which was just performed, and should be reported to standard output as don or failed,
- **estimated_percentage** – (int)

  Percent should be calculated as part of whole execution; first and last 5 percent should be reserved for programs which runs 'metatranscriptomics', for pre- and postprocessing,
- **done** – informs whether the operation from 'what' argument failed or was successfully done.

rapsearch2(*input_file*, *threads*)
Runs `rapsearch2` for *input_file* in fasta format.

Writes outputs in "m8/" directory.

GLOBALS:

- path to RAPSearch2 program: PATH_RAPSEARCH
- path to similarity search database: PATH_REF_PROT_KO

run_SARTools()
Runs SARTools in R.

HARDCODED:

R templates:

- edger: template_script_DESeq2.r
- deseq: template_script_edgeR.r

run_cat_pairing()
Merges fasta files with paired-end reads in cwd.

run_fastq_to_fasta(*fastqs*)
Runs *fastq_to_fasta()* for every .fastq in fastqs.

run_ko_csv(*ko_dict_deseq*, *ko_dict_edger*, *all_conds*, *kopath_keys*, *path_names*, *ref_cond*)
For given ko_dicts writes CSV files with pathways and foldchanges

Parameters

- **ko_dict** – {KO_id:{cond1:value1, cond2:value2...}...} dict
- **all_conds** – list of conditions (list of strings)
- **kopath_keys** – {KO identifier:set[KEGG_Pathway_ids]} dict
- **path_names** – {KEGG_Pathway_id:Name} dict
- **filepath** – output filepath

Output files have following format (and header):: KO_id;Gene_name;paths ids;paths names;FC vs cond1;FC vs cond2;...;

HARDCODED:

Output files paths:

- deseq: 'deseq.csv'
- edger: 'edger.csv'

run_ko_map()

Runs *m8_to_ko()* for every .m8 file in cwd.

GLOBALS:

- path to KO database: PATH_KO_DB
- pickle to dict from KO GENES table from KO database: PATH_KO_PCKL

run_ko_remap(*deseq_files*, *edger_files*, *kopath_values*, *path_names*)

Runs out_content(files, kopath_values, path_names (,'edgeR')) for files from *edger_paths* and *deseq_paths*.

Parameters

- **deseq_diles** – list of DESeq outputs paths
- **edger_files** – list of edgeR outputs paths
- **kopath_values** – {KEGG_Pathway_id: set[KO identifiers]} dict
- **path_names** – {KEGG_Pathway_id: Name} dict

run_new_ko_remap(*deseq_files*, *edger_files*, *kopath_values*, *all_conds*, *ref_cond*)

Runs *get_ko_fc()*, *low_change()*, *mapper()* and *mapper_write()* in appropriate way for files from *deseq_files* and *edger_files*.

Parameters

- **deseq_diles** – list of DESeq outputs paths
- **edger_files** – list of edgeR outputs paths
- **ref_cond** – Reference condition (group) - string
- **kopath_values** – {KEGG_Pathway_id:set[KO identifiers]} dict
- **all_conds** – list of conditions (list of strings)

Returns {KO_id:{cond1:value1, cond2:value2...}...} dict ko_dict_edger: {KO_id:{cond1:value1, cond2:value2...}...} dict

Return type ko_dict_deseq

HARDCODED:

Output directories paths:

- deseq: 'new_ko_remap/deseq/'
- edger: 'new_ko_remap/edger/'

run_pre_ko_remap()
Prepares args for *run_ko_remap()* or *run_new_ko_remap()*

Returns {KEGG_Pathway_id:Name} dict kopath_keys: {KO identifier:set[KEGG_Pathway_ids]} dict kopath_values: {KEGG_Pathway_id:set[KO identifiers]} dict edger_files: list of edgeR outputs paths deseq_diles: list of DESeq outputs paths

Return type path_names

HARDCODED:

Paths to files from SARTools:

- edger: *'edger/*[pn].txt'*
- deseq: *'deseq/*[pn].txt'*

GLOBALS:

- path to KO database: PATH_KO_DB

run_rapsearch(*threads*)
Runs *rapsearch2()* for every .tmp.fasta in cwd.

1.3 Refseq bipype

1.3.1 Module functions

MH(*mode*, *e*, *glob_out_dir*, *t*, *cat*, *pair*, *rap=False*, *presets='meta-large'*, *megan=False*)
Runs Megahit (also *rapsearch()* and *run_megan()*)

Parameters

- **mode** – if *mode* is other than 'run' program prints commands without executing them
- **e** – if true, then program updates todo list with *exist_check()* function.
- **t** – number of threads
- **cat** – name of current folder
- **pair** – tuple of paired_end reads
- **rap** – if true, program runs *rapsearch()* function.
- **presets** – argument for megahit
- **megan** – if true, program runs megan() function.

GLOBAL: PATH_MEGAHIT

MV(*mode*, *e*, *global_out_dir*, *k_mers*, *cat*, *pair*, *ins_len*, *rap=False*, *megan=False*)
Runs Velvet. Runs *gzip_MV()* function on folder with Velvet results.

The name of the folder is derived from *pair* argument:

```
pair_uni_name(pair) + '_velvh_out'
```

The function can also run *rapsearch()* and `megan()` functions when called with appropriate arguments.

Parameters

- **mode** – if *mode* is 'run' program runs velveth or velvetg or metavelvetg
- **e** – if true, then program changes todo list with *exist_check()*: function
- **k_mers** – k-length nucleotides reads list
- **cat** – name of current folder
- **pair** – tuple of paired_end reads
- **ins_len** – if *9999* given, then *ins_len* will be retrieved as an output of *ins_len_read()* function
- **rap** – if true, then program runs *rapsearch()* function.
- **megan** – if true, then program runs `megan()` function.

GLOBALS: PATH_VELVETH PATH_VELVETG PATH_METAVELVETG

SSU_read(*loc*, *headers_type='ITS'*)
Extracts from specially formatted FASTA file taxonomic data and returns them as hierarchically organised dict.

Headers of FASTA files should follow schemas presented by following examples:

•ITS format (used in UNITE database):

```
>DQ233785|uncultured ectomycorrhizal fungus|Fungi|Thelephora terrestris|Fungi; Basidiomycota
```

•16S format (alternative):

```
>AF093247.1.2007 Eukaryota;Amoebozoa;Mycetozoa;Myxogastria;;Hyperamoeba_sp._ATCC50750
```

Parameters

- **loc** – location - path to the FASTA file
- **headers_type** – tells which format of header is present in given file. If specified, indicates use of alternative format.

Returns

A dict with taxonomic data, where

- keys are sequence identifiers;
- values are lists of taxonomic terms, in order: from the most generic to the most specific one.

Example:

```
{
    'DQ482017':
    [
```

```
                    'Fungi',
                    'Basidiomycota',
                    'Agaricomycotina',
                    'Agaricomycetes',
                    'Incertaesedis',
                    'Thelephorales',
                    'Thelephoraceae',
                    'Tomentella',
                    'Tomentellasublilacina'
                ],
                'EF031133':
                [
                    'Fungi',
                    'Basidiomycota',
                    'Agaricomycotina',
                    'Agaricomycetes',
                    'Incertaesedis',
                    'Thelephorales',
                    'Thelephoraceae',
                    'Thelephora',
                    'Thelephoraterrestris'
                ]
            }
```

adapter_read(*filename*)

Replace 'NNNNNN' part of adp_2 by this part of the filename, which contains only letters that are symbols of nucleotides.

Example

input:

```
Amp18_BFp_B_p_GACGAC_L001_R1_001.fastq
```

output:

```
(
    'AGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGTAGATCTCGGTGGTCGCCGTATCATT',
    'AGATCGGAAGAGCACACGTCTGAACTCCAGTCACGACGACATCTCGTATGCCGTCTTCTGCTTG'
)
```

HARDCODED: Adapter sequences

adapter_read_bck(*adapter_file*, *filename*)

Function reads all lines of given *adapter_file* and separates words in each line. If first word in a line is equal to filename, then second and third words are returned as a tuple.

Parameters **filename**(*adapter_file,*) –

Returns tuple (fin_adap_1, fin_adap_2)

auto_tax_read(*db_loc*)

Read `{GI:TaxID} & {TaxID:scientific_name}` dicts from pickle.

Parameters **db_loc** – path to pickle file

Returns `{GI:TaxID}` Keys are integers, values are strings `{TaxID:scientific_name}` Keys and values are integers

Return type Two dicts

bam_idxstating(*mode*, *sorted_bam*, *idxstats*)
Launch samtools idxstats.

Parameters

- **mode** – if *mode* is 'run', launch the program.
- **sorted_bam** – path to file in BAM format, which is the input to samtools.
- **idxstats** – path to file, where the output from samtools will be written.

bam_indexing(*mode*, *sorted_bam*)
If mode is 'run' creates an index file *sorted_bam*.bam.bai for the *sorted_bam*.bam file.

Parameters

- **mode** – string type
- **sorted_bam** – BAM file

bam_make(*mode*, *sam*, *bam*)
If *mode* is 'run' and @SQ lines are present in the header, the function converts SAM to BAM

Parameters

- **mode** – string type
- **sam** – SAM file
- **bam** – BAM file

bam_sorting(*mode*, *bam*, *sorted_name*)
If *mode* is 'run' the function reads bam (file in BAM format), sort it by aligned read position and write it out to BAM file whose name is: *sorted_name*.

Parameters

- **mode** – string type
- **bam** – BAM file
- **sorted_name** – name of output file

bowtie2_run(*mode*, *proc*, *ref*, *out*, *inp1*, *inp2=False*)
If *mode* is 'run', launches Bowtie 2.

Parameters

- **mode** – should bowtie be runned?
- **proc** – [-p in Bowtie 2]

  Number of alignment threads to launch.
- **ref** – [-x in Bowtie 2]

  The basename of the index for the reference genome.
- **out** – [-S in Bowtie 2]

  File to write SAM alignments to.

- **inp1** – [-1 in Bowtie 2]

  Comma-separated list of files containing mate 1s (filename usually includes _1), for example:

```
-1 flyA_1.fq,flyB_1.fq
```

- **inp2** – [-2 in Bowtie 2]

  Comma-separated list of files containing mate 2s (filename usually includes _2), for example:

```
-2 flyA_2.fq,flyB_2.fq
```

  If inp2 is False: -2 argument will not be passed to Bowtie 2.

cat_read(*mode*, *fileext*, *paired_end=True*)

Returns a dict with paths to sequence files from current working directory.

The dict has following format:

```
{
    directory_path1: [
        file1_in_this_directory_path,
        file2_in_this_directory_path,
        file3_in_this_directory_path
    ],
    directory_path2: [
        file4_in_this_directory_path,
        file5_in_this_directory_path,
        file6_in_this_directory_path
    ]
}
```

Paths to files (values of dict) are relative.

Parameters

- **mode** – If mode is 'run', the gunzip command will be executed to extract compressed files.
- **fileext** – Extension of sequence files, which will be putted into the dict.
- **paired_end** – If True, only paired-end reads are included in the dict (default True).

create_krona_html(*krona_html_name*, *krona_xml_name*)

Runs script which creates Krona html file from krona xml file

create_krona_xml(*xml_string*, *out_xml*)

Writes xml_string into the file of name out_xml. If the file does not exist, creates a new file.

Parameters

- **xml_string** – a string with content to write
- **out_xml** – a string with path to file

cutadapt(*mode*, *e*, *cat*, *R1_file*, *R2_file*, *adapter_file*, *usearch_16S=False*, *usearch_ITS=False*, *threads=False*)

Performs preparation of reads to be used in usearch in three steps:

1. cutadapt - searches for the adapters in reads from input files, removes them, and creates output files with `.cutadapt.fastq` extension.

2.runs FLASH (software tool to merge paired-end reads) with those files and gets `.amplicons.cutadapt.flash.merged.fastq` files from output.

3.these results are input for fastq_to_fasta which converts file format from fastq to fasta

Optionally usearch can be run on files prepared this way.

Parameters

- **mode** – if mode is 'run', function operate on data
- **e** – if true, checks if a part of the workflow is actually done and omits these parts, to avoid duplicating this job.
- **cat** – name of folder with R_1 and R_2 files
- **R2_file** (*R1_file,*) – input files
- **adapter_file** – If 'use_filenames', the function will use adapters returned by *adapter_read(R1_file)*'. Else, function will use adapters returned by `adapter_read_bck(adapter_file, R1_file)`
- **usearch_16S** – if true runs `usearch(mode, e, '16S', outname_uni_fasta, usearch_16S, threads)` where *outname_uni_fasta* is cutadapt output.
- **usearch_ITS** –

  if true runs `usearch(mode, e, 'ITS', outname_uni_fasta, usearch_ITS, threads)` where *outname_uni_fasta* is cutadapt output.

  See Also: Follwoing functions may provide you with better background. - *adapter_read_bck()* - *adapter_read()*

GLOBALS: PATH_FQ2FA

deunique (*node*)

Extract name of a node from string representing taxonomic position of this node.

Example

input:

```
'lvl1_____lvl2_____lvl_3_____lvln'
```

output:

```
'lvln'
```

Parameters **node** – a string with a list of ancestors of the node and the node itself, separated by '_____'.

Returns A string with pure name of the node.

See also:

linia_unique() function

dict_prepare (*typ*, *indict*, *SSU*)

Runs files analysis and then, creates two dicts with summarized taxonomic data. The resulting dicts keep information about files from which given taxa comes.

Parameters

- **typ** – an "output type" - typically: 'ITS', '16S' or 'txt'.
- **indict** –

  A dict with file structure, where:

  - keys are directories,
  - values are lists of files.

  Example in description od output of *cat_read()* function.

- **SSU** –

  A dict with taxonomic data, where:

  - keys are sequence identifiers;
  - values are lists of taxonomic terms, in order: from the most generic to the most specific one.

  Example in description of output of *SSU_read()* function.

Returns

(all_dicts, all_tax_dict)

all_dicts: A dict, where

- keys are file identifiers,
- values are taxonomic trees with numbers of occurrences of particular species placed inside leafs

Example:

```
{
    'filename_1.ext':
    {
        'Bacillaceae': {'Anoxybacillus': {'subsum': 15}}
    },
    'filename_2.ext':
    {
        'Listeriaceae':
        {
            'Listeria':
            {
                'Lgrayi': {'subsum': 22},
                'Linnocua': {'subsum': 11}
            }
        }
    }
}
```

all_tax_dict: A dict, where

- keys are file identifiers,
- **values are dicts, where**
  - keys are names of taxa,
  - values are numbers of occurrences of particular taxon.

Example:

```
{
    'filename_1.ext':
    {
        'Bacillaceae': 5, 'Anoxybacillus': 5
    },
    'filename_2.ext':
    {
        'Listeriaceae': 19, 'Listeria': 19, 'Lgrayi': 16, 'Linnocua': 3
    }
}
```

Return type A tuple

dict_purify(*bac_dict*)
Removes from given dict values below the hardcoded threshold.

Parameters **bac_dict** – a dict.

Returns A new dict without entries, within values were lower than threshold.

HARDCODED: threshold = 10

dict_sum(*dicto*, *val*)
Sums the values in the given dict, using recurrence to handle nested dicts.

Parameters

- **dicto** – a dict, where values are of any type with defined + operation
- **val** – an int - initial value

Returns an int - sum of all values in given dict plus value given as second argument

Return type val

exist_check(*program*, *names*, *todo*)
Checks if a part of program is actually done.

Function tries to find output files from different parts of program and basing on the results, removes corresponding tokens from todo list.

In one special case (program == 'usearch_0') function checks size of the file specified by <names> argument in bits. If size == 0: todo=[] and PANIC MODE information is printed.

Parameters

- **program** – One of the following strings representing programs (de facto functions):
  - 'refseq' (&)
  - 'usearch' (@)
  - 'usearch_0' (@)
  - 'MV' (^)
  - 'rapsearch' (@)
  - 'cutadapt' (^)
- **names** – There are three possibilities:
  - (&) A dict in following format:

```
{file extension : path to file with corresponding extension}
```

- (@) A string (one path)
- (^) A list of paths

Paths are hypothetical - if one really exists, appropriate token is removed from todo list.

- **todo** – List of tokens representing different parts of function specified by 'program' argument.

Please, pay attention to mutual compatibility of arguments.

Result: todo: Checked list of tokens.

See also:

For more information please refer to descriptions of the following functions:

- *refseq_ref_namespace()* and *refseq_mapping()*
- *usearch()*
- *MV()*
- *rapsearch()*
- *cutadapt()*

fastq_dict (*seq_dict*, *root*, *file_*)

Adds *file_* to list available under *seq_dict[root]*.

If key *root* is not present, it will be created.

file_analysis (*typ*, *name*, *SSU=None*)

Using given SSU as database, performs 'statistical' analysis of taxonomy from file *name*. It counts occurrences of different species in given file and returns result in form of two dicts.

The threshold hardcoded in dict_purify is used, to remove neglectable entities.

Parameters

- **typ** – an "output type" - typically: 'ITS', '16S' or 'txt'.
- **name** – name of file to analise
- **SSU** –

  A dict with taxonomic data, where:

  - keys are sequence identifiers;
  - values are lists of taxonomic terms, in order: from the most generic to the most specific one.

  Explicit examples included in description of *SSU_read()* function.

Returns

If file of name <name> was not found: ('NA', 'NA')

If data of type '16S' was successfully analysed: (full_bac_dict, tax_bac_dict)

If data of type 'ITS' was successfully analysed: (full_fun_dict, tax_fun_dict)

full_bac_dict / full_fun_dict: a dict of dicts.

It represents taxonomic tree. Generally in this dict:

- keys are names of taxa,
- values are nested dicts of the same type.

Values in the deepest levels ("leafs") are counts of occurrences.

Example:

```
{
    'Bacillaceae':
    {
        'Anoxybacillus': {'subsum': 15}
    },
    'Listeriaceae':
    {
        'Listeria':
        {
            'Lgrayi': {'subsum': 22},
            'Linnocua': {'subsum': 11}
        }
    }
}
```

tax_bac_dict / tax_fun_dict: a dict, with taxonomic statistics:

- keys are names of taxa
- values are counts of occurrences

Example:

```
{
    'Bacillaceae': 15,
    'Anoxybacillus': 15,
    'Listeriaceae': 33,
    'Listeria': 33,
    'Lgrayi': 22,
    'Linnocua': 11
}
```

Return type A 2-tuple

HARDCODED: In '16S' analysis, if spec is 'Phaseolus_acutifolius_(tepary_bean)', then it is not counted neither as bacteria nor archaea.

TODO: There was a suggestion, that there might be something wrong with handling of 'name' argument, when passed in only file_analysis call present in this file. The issue is linked to "full path or basename?" question. It should be investigated, in fully functional testing environment.

graphlan_to_krona(*input_d*)

Modifies files given by *inupt_d* from Graphlan format to set of xml strings, which may be assembled into a input for Krona.

Parameters **input_d** –

A dict where:

- keys are names of directories,
- values are lists with filenames, which are in the directory.

Example:

```
{'dir_name_1': ['file_1','file_2'], 'dir_name_2': ['file_1']}
```

Explicit example in description of output of *cat_read()*.

Returns

A tuple –

xml_names: readable identifiers of files derived from filenames. Filenames comes from input_d dict. Example:

```
[identifier_1, identifier_2, ..., identifier_x]
```

xml_dict: Contains lists of numbers of occurrences of nodes, grouped by node:

- keys are names of nodes,
- values are lists of constant length, where every item on position *X* means: number of occurrences of nodes with prefix equal to *name of node* in file *X*. *file X* is that file, which is on position *X* on list xml_names.

Note, that for all i: len(xml_dict[i]) is equal to len(xml_names).

Example:

```
{node: [count_1, count_2, ... count_x], node_2: [count_1, count_2, ... count_x]}
```

tax_tree: A dict of dicts, etc. In form of nested dicts, it represents taxonomic tree. Example:

```
{a:{aa:{}, ab:{aba:{}, abb:{abba:{}}}}}
```

Explicit example in description of *total_tax_tree* in *tax_tree_graphlan()*,

name_total_count: A dict. Contains summed numbers of occurrences of nodes by file.

- keys are identifiers (derived from filenames - check `xml_names` description),
- values are sums.

Example:

```
{identifier_1: count_1, identifier_2: count_2}
```

Return type xml_names, xml_dict, tax_tree, name_total_count

gzip_MV(*MV_dir*)

Return archive with catalogs Sequences, Roadmaps, PreGraph, Graph2, LastGraph and files with statistics.

Parameters **MV_dir** – folder with Velvet results

humann(*mode*, *e*, *m8_dict*, *typ='m8'*)

Copies humann to the current directory, moves input (*.m8) files to the input directory, copies hmp_metadata.dat file to the input directory and runs humann.

GLOBALS:

- path to humann program: PATH_HUMANN
- path to data for humann: PATH_HUMANN_DATA

Parameters

- **mode** – if 'run', then humann will be copied to the current directory
- **e** – if true, then function checks existence of humann-0.99 results folder
- **m8_dict** – the similarty search results folders
- **typ** – default typ="m8", in that case new catalog humann-0.99 will be created in rapsearch result folder; in other case humann analysis results will be added in rapsearch result folder.

idx_map(*mode*, *file_*, *tax_name_dict*, *tax_id_dict*, *outfile*)

Parses and writes data from samtools idxstats output file.

Firstly, using `idx_reader(file_)`, creates `{GI:#_mapped_reads}` (a dict).

Secondly, replaces every GI (key) with TaxID if appropriate one is available in tax_id_dict.

Than, replaces every GI /TaxID (key) with scientific name if appropriate one is available in tax_name_dict.

Finally, writes data (sorted by numbers of mapped reads in decreasing order) to outfile in key;value format, where:

•key is GI/TaxID/scientific name

•value is number of mapped reads

Parameters

- **mode** – If (mode == 'run'), function do mentioned things. Elif (mode == 'test') function prints *file_* and outfile arguments. Else function do nothing.
- **file_** – Path to samtools idxstats output file. For more information refer to *idx_reader()* function.
- **tax_name_dict** – {TaxID:scientific_name} dict. For more information refer to *tax_name_reader()* function.
- **tax_id_dict** – {GI:TaxID} dict.__weakrefoffset__ For more information refer to *tax_id_reader()* function.
- **outfile** – Path to output file.

idx_reader(*file_path*)

Reads file in samtools idxstats output format.

Parameters **file_path** – path to samtools idxstats output file.

File is TAB-delimited with each line consisting of reference sequence name, sequence length, # mapped reads and # unmapped reads

Returns a dict with `{GI: #_mapped_reads}` (keys and values are integers).

idxstat_perling(*mode*, *idxstats*, *map_count*)

Counts and writes to *map_count* both:

- sum of all (numbers of mapped reads) in *idxstats*,
- sum of all (numbers of unmapped reads) in *idxstats*.

Function launches short "one-liner" in perl.

Parameters

- **mode** – If mode is 'run' the perl "one-liner" will be launched.

- **idxstats** – Path to samtools idxstats output file. Please refer to *idx_reader()* for more information.
- **map_count** – Path to file, where differences will be written.

Output: Output has following format:

```
#mapped - #unmapped
```

Example:

```
123 - 456
```

TODO: This should be rewritten to python (why to use perl here?)

input_locations(*mode*, *out_types*)

Generates lists of locations where input files are located. Lists are grouped by directories and later, by "output types".

As input files considered are only files, which meet all the following conditions:

•they are located in current working directory or in subdirectories of current working directory,

•have suffix of filename equal to *out_type*,

•if *out_type* is 'ITS' or '16S', filenames also have to contain 'usearch_' before the *out_type* in name.

Parameters

- **mode** – a parameter passed to *cat_read()* function. If mode is 'run', the gunzip command will be executed, to extract compressed files.
- **out_types** – a list with "output types", typically consisting of: 'ITS', '16S' or 'txt'.

Returns

A dict where keys are "output types", values are dicts describing file locations returned by *cat_read()* function.

Example:

```
{
    'ITS':
    {
        directory_path1:
            [file1_in_this_directory_path,
            file2_in_this_directory_path,
            file3_in_this_directory_path],
        directory_path2:
            [file4_in_this_directory_path,
            file5_in_this_directory_path,
            file6_in_this_directory_path]
    }
}
```

ins_len_read(*pair*, *cat*)

Return 200 if the read length is less then 200 or 500 in otherwise.

Parameters

- **pair** – tuple of paired_end read
- **cat** – name of folder with the sample files

Example

input:

```
('Amp15_BFk_B_p_CGTACG_L001_R1_001.fastq','Amp15_BFk_B_p_CGTACG_L001_R2_001.fastq'), 'catalog_na
```

output:

```
500
```

linia_unique(*linia*)

Creates list where every element includes information about previous elements, in order, separated by "____".

Example

input:

['1','2','3'],

output:

['1', '1_____2', '1_____2_____3']

Parameters **linia** – a list.

Returns A list.

See also:

deunique() function

name_total_reduction(*xml_names*, *file_total_count*)

Rewrites given <file_total_count> dict, to group total counts by names (datasets), instant of grouping by filenames.

Keep in mind, that this step - along with generation of xml_names in xml_name_parse - groups files referring to the same dataset under single name, and technically is vulnerable for some errors.

See also:

description of *xml_vals()* function

Parameters

- **xml_names** – readable identifiers of groups of files referring to common dataset; derived from filenames. Example:

```
['filename_1', 'filename_2']
```

- **file_total_count** – A dict within information about files are kept and values of every leaf are summed into values (usually representing total count of species inside particular file)

  Example:

```
{
    'filename_1.ext': 15,
    'filename_2.ext': 33,
    'filename_2_part_2.ext': 33
}
```

Returns

A dict, where keys are names as present in xml_names and values are summed total counts (usually of occurrences of different species).

Example:

```
{'filename_1': 15, 'filename_2': 66}
```

out_namespace (*curr*, *out_types*)

Generates paths to places, where krona xml and krona html files are or will be placed. Path composition varies, according to given arguments and always contains some string representation of *out_type*. Paths may start in current working directory or in directory specified by *curr*.

If 'txt' is one of output_types, the basename of output files will be 'humann-graphlan'; otherwise, the basename will be concatenation of elements from 'out_types' list, with underscore '_' as glue.

In case of multiple elements on <out_types> list, they will be sorted, so the output files will have consistent names regardless to parameters' order.

Examples of generated filenames, basing on <out_types>:

•for ['ITS']: 'ITS.krona', 'ITS.html'

•for ['ITS', '16S']: '16S_ITS.krona', '16S_ITS.html'

•for ['txt']: 'humann-graphlan.krona', 'humann-graphlan.html'

Parameters

- **curr** – A string - path to directory where the files are/will be placed. If curr is 'in_situ', the path will start in current working directory.
- **out_type** – A list that determines the basename of output files. In practice, we expect list of elements: 'ITS', '16S' or 'txt'.

Returns

A tuple –

out_xml: a string with path to file. The filename is derived from (out_type) and file has '.krona' extension.

out_html: a string with path to file. The filename is derived from (out_type) and file has '.html' extension.

Return type out_xml, out_html

pair_uni_name (*file_pair*)

Returns one name for given tuple containing pair of filenames.

Parameters **file_pair** – tuple of paired_end read

Example

input:

```
('Amp15_BFk_B_p_CGTACG_L001_R1_001.fastq','Amp15_BFk_B_p_CGTACG_L001_R2_001.fastq')
```

output:

```
    'Amp15_BFk_B_p_CGTACG_L001_001'
```

paired_end_match(*seq_dict*)

Returns a dict with paths to paired-end reads only. Files will be matched as pair if their names differs only by parts:

- R1 and R2, if the parts of filename are separated by underscores _
- 1 and 2, if the parts of filename are separated by dots .

Parameters **seq_dict** – A dict determining locations of paired-end sequences.

Example:

```
{directory_path:file_path}
```

Returns: a dict

The returned dict has following format:

```
{
    directory_path1: [
        (paired-end_read1_R1_path, paired-end_read1_R2_path),
        (paired-end_read2_R1_path, paired-end_read2_R2_path),
        (paired-end_read3_R1_path, paired-end_read3_R2_path)
    ],
    directory_path2: [
        (paired-end_read4_R1_path, paired-end_read4_R2_path)
    ]
}
```

Paths to files (both in input and output dict) are relative.

prepare_taxonomy_dicts(*opts*, *input_dict*)

Prepares dicts with taxonomy stats.

See also:

prepare_taxonomy_stats() for more information.

prepare_taxonomy_stats(*opts*)

Performs statistical analysis of taxonomy from appropriate files from current working directory: counts occurrences of different taxa and prepares the results to be presented in HTML format. Results will be converted to HTML (with the Krona program), but only when *opts.mode* is set to 'run'.

Parameters **opts** – A namespace, where:

opts.output_type: Allows to choice on which files the analysis will be performed and also determines basenames of output files.

One of: ['ITS', '16S', 'txt']

opts.mode: A mode in which the program will be run.

One of: ['test', 'run']

opts.out_dir: Indicates, where the output files should be located. To specify current working directory, use 'in_situ'.

One of: ['in_situ', a_string_with_path_to_dir]

opts.db_taxonomy_16S: Path to 16S database used in taxonomy classification (fasta with specially formatted headers)

opts.db_taxonomy_ITS: Path to ITS database used in taxonomy classification (fasta with specially formatted headers)

Input:

Analysis will be performed on files, meeting all the following conditions:

- files are located in current working directory or in subdirectories of current working directory,
- suffix of filename is equal to *opts.output_type*,
- if *opts.output_type* is 'ITS' or '16S', filenames contains 'usearch_' before the *opts.output_type* in name.

Output:

Output files will be placed in *opts.out_dir* directory, with basenames depending on *opts.output_type*.

Examples of filenames:

- for 'ITS': ('ITS.krona', 'ITS.html')
- for ['ITS', '16S']: ('ITS_16S.krona', 'ITS_16S.html')

prettify(*elem*)

Creates a string representation of an XML element by calling xml.etree.ElementTree.tostring() and then, parses this string again to obtain pretty-printed XML string, where indents are four spaces long.

Parameters **elem** – instance of class Element from xml.etree.ElementTree

Returns A string containing pretty-printed XML of <elem>

rapsearch(*mode*, *e*, *contig_loc*, *rap_out*, *KEGG=None*)

Runs RAPSearch with using KEGG databases for similarity search.

Parameters

- **mode** – if *mode* is 'run', then program runs rapsearch
- **e** – if e=True, then function runs exist_check function
- **contig_loc** – a query file
- **rap_out** – output file name
- **KEGG** – default is None, if KEGG= KO, then ko.pep.rapsearch.db is chosen as protein database

HARDCODED: if KEGG='masl'

GLOBALS:

- path to RAPSearch program: PATH_RAPSEARCH
- **paths to data for RAPSearch:** PATH_REF_PROT_MASL28, PATH_REF_PROT_KO, PATH_REF_PROT_ELSE

reconstruct(*mode*, *thr*, *e*, *pair*, *cat*, *prefix*, *rec_db_loc*)

Runs bwa. Find the SA coordinates of the input reads. Generate alignments in the SAM format given single-end reads. Repetitive hits will be randomly chosen.

Parameters

- **mode** – if 'run', then commands "bwa aln" and "bwa samse" were run
- **thr** – Number of threads (multi-threading mode)

- **e** – (!!!! Wasn't used !!!!)
- **pair** – tuple of paired_end read
- **cat** – name of folder with the sample files
- **prefix** – prefix for output files names
- **rec_db_loc** – input database with fasta files

refseq_mapping(*mode*, *e*, *directory*, *pair*, *postfix*, *refseq*, *tax_name_dict*, *tax_id_dict*, *threads*, *map_dir*, *refseq_2=False*)

Aligns reads to reference sequence(s) using Bowtie 2, than parses and writes data to files.

•Firstly, Bowtie 2 align reads to reference sequence (or sequences, if refseq_2 is not False and merge output SAM files).

•then, BAM files are made, sorted and indexed,

•the function launches 'samtools idxstats',

•in the next step, perl 'one-liner' counts and writes to file sums of mapped reads and unmapped reads,

•finally, *idx_map()* function is called: it parses data from samtools idxstats output file and writes data to *outfile* in `key;value` format, where:

–key is GI/TaxID/scientific name

–value is number of mapped reads.

Parameters

- **mode** – If *mode* is 'run', the function operates on data and prints information about it. Otherwise, prints information, without operating on data (it is a kind of test).
- **e** – If true, checks if a part of workflow is actually done and doesn't duplicate this jobs.

  For more information, please refer to `e` argument in *exist_check()* function.
- **directory** – Path to directory, where files will be written.
- **pair** – Tuple of paths to file (paired-end reads) OR a string with path to a sequence file.
- **postfix** – String added to the end of files basenames.

  Argument for *refseq_ref_namespace()*.
- **refseq** – The basename of the index for the reference genome.

  Argument for *bowtie2_run()*.
- **tax_name_dict** – A dict: `{TaxID:scientific_name}`

  For more information refer to *tax_name_reader()*.

  Argument for *idx_map()*.
- **tax_id_dict** – A dict: `{GI:TaxID}`

  For more information refer to *tax_id_reader()*.

  Argument for *idx_map()*.
- **threads** – Number of threads for Bowtie 2 calculations.
- **map_dir** – Directory where sums of mapped and unmapped reads will be written. For more information refer to *idxstat_perling()*.

- **refseq_2** – The basename of the index for the reference genome.

  Argument for *bowtie2_run()*.

  If selected, launches Bowtie 2 on it, then merges output with output from Bowtie 2 launched on 'refseq'.

  If refseq_2 == False, Bowtie 2 is working only on 'refseq' argument.

See also:

For more information take a look at following functions:

refseq_ref_namespace() exist_check() bowtie2_run() sam_merge() bam_make() bam_sorting() bam_indexing() bam_idxstating() idxstat_perling() idx_map()

refseq_ref_namespace(*directory*, *seq*, *postfix*, *out_dir='in_situ'*, *map_dir='in_situ'*)

Returns a dict within:

- keys are following file extensions: fastq, sam, sam2, bam, sorted, sorted.bam, idx_stats, tax_count, map_count
- values are paths to file with corresponding extension.

Filenames have following format:

```
sample_name + '_' + postfix + extension
```

where sample_name is basename of seq (if seq is a file) or output of pair_uni_name(seq) (if seq is a tuple)

Parameters

- **directory** – Path to directory, where file with .fastq extension will be written
- **seq** – Tuple of paired_end read or sequence file.
- **postfix** – String added to the end of file basenames.
- **out_dir** – Path to directory, where files with .sam, .sam2, .bam,.sorted, .sorted.bam, .idx_stats and .tax_count extensions will be written. If out_dir is 'in_situ' (default), out_dir=directory
- **map_dir** – Path to directory, where file with .map_count extension will be written. If map_dir is 'in_situ'(default),map_dir=out_dir.

run_megan(*out_dir*, *m8*, *contigs*, *tax_gi_file='/home/pszczesny/workingdata/mapping/gi_taxid_prot.dmp'*)

Makes and launches scripts for MEGAN

Parameters

- **out_dir** – output directory
- **m8** – list of m8 files
- **contigs** – contigs file
- **tax_gi_file** – taxGIFile

HARDCODED: default path to taxGIFile

sam_merge(*sam1*, *sam2*)

Merges two files in SAM format into one.

Parameters

- **sam1** – SAM file
- **sam2** – SAM file

Returns sam1 file in SAM format which contains now lines from both files. Function also removes sam2 file.

sample(*opts*)

Runs programs such as Velvet, bwa, Bowtie 2, humann; cuts adapters or run RAPSearch if these options were chosen.

Parameters **opts** – A namespace, where:

opts.to_calculate: Input sequences type (Plant or fungi). TODO (this description seems to be misleading)

opts.mode: A mode in which the program will be run. One of: ['test', 'run'].

opts.db_NCBI_taxonomy: Path to pickle file.

opts.reconstruct: Determines if *reconstruct()* function should be run.

opts.assembler: Assembly program One of: ['MH','MV'].

opts.MV: k-length nucleotids reads list or None.

opts.db_reconstruct: Input database with fasta files.

opts.threads: Number of threads (multi-threading mode).

opts.e: If true, check if file already exists.

opts.ins_len: 200 if read length is less than 200 and 500 in otherwsie.

opts.db_refseq_fungi: A list of paths to refseq databases to use in fungi analysis. Up to two paths are allowed.

Warning: If there are multiple databases with filenames like <your_path><second_part_of_name>, all will be loaded.

opts.db_refseq_plant: A list of paths to refseq databases to use in plants analysis. Up to two paths are allowed. Also check warning in opts.db_refseq_fungi description.

opts.our_dir: Directory where sums of mapped and unmapped reads will be written

opts.postfix: TODO

opts.cutadapt: List of adapter types: '16S', 'ITS' or 'both'.

opts.db_16S: database of 16S adapters, which will be -db parameter for Usearch program; if it is not empty, then cutadapt function's parameter usearch_16S will be True.__truediv__

opts.db_ITS: database of ITS adapters, which will be -db parameter for Usearch program; if it isn't empty,then cutadapt function's parameter usearch_ITS will be True.

GLOBALS: PATH_FQ2FA

HARDCODED: actions dependent on existence of 'seq' substring in some paths.

tax_id_reader()

Returns a dict with {GI:TaxID} from file.

Keys and values are integers.

File has following format:

```
13  9913
15  9915
16  9771
17  9771
```

where first column is a GI, second is a TaxID.

GLOBLAS: PATH_GI_TAX

tax_name_reader()
Returns a dict with {TaxID:scientific_name} from file.

Lines without scientific names are omitted.

Keys are integers, values are strings.

Example

File:

```
2   |       prokaryotes     |       prokaryotes <Bacteria>  |       in-part |
6   |       Azorhizobium    |               |       scientific name |
6   |       Azorhizobium Dreyfus et al. 1988        |               |       synonym |
6   |       Azotirhizobium  |               |       equivalent name |
7   |       ATCC 43989      |               |       type material   |
7   |       Azorhizobium caulinodans        |               |       scientific name |
```

Output:

```
{6:'Azotirhizobium', 7:'Azorhizobium caulinodans'}
```

GLOBALS: PATH_TAX_NAME

tax_tree_extend(*tax_tree*, *linia*)
Recursively adds elements form list *linia* into the tree *tax_tree*.

Parameters

- **tax_tree** – tree represented as dict of dicts.
- **linia** – a list in order: from the oldest ancestor to the youngest descendant. Elements should be formatted to contain information about ancestors (like it does *linia_unique()* function).

Returns Extended tree in form of dict of dicts.

tax_tree_graphlan(*input_d*)
Reads and interprets information about taxonomic relations, from files in Graphlan-like format.

Parameters **input_d** – A dict where keys are names of directories, values are lists with file-names, which are in the directory.

Example:

```
{'dir_name_1': ['file_1','file_2'], 'dir_name_2': ['file_1']}
```

Explicit example in description of output of *cat_read()*.

Example of featured file:

```
Listeriaceae.Listeria.Lgrayi
Listeriaceae.Listeria.Linnocua
```

Returns

A tuple –

total_tax_tree: A dict of dicts, etc. In form of nested dicts, represents phylogenetic trees of entities described by files featured in the input. This is a total tree - nodes from all files are merged here.

Example:

```
{
    'Bacillaceae':
    {
        'Bacillaceae_____Anoxybacillus': {}
    },
    'Listeriaceae':
    {
        'Listeriaceae_____Listeria':
        {
            'Listeriaceae_____Listeria_____Lgrayi': {},
            'Listeriaceae_____Listeria_____Linnocua': {},
        }
    }
}
```

per_file_tax_tree: A dict of dicts, etc. Keys are filenames. In form of nested dicts, represents phylogenetic trees of entities, described by files featured in the input.

Example:

```
{
    'annot_1.txt':
    {
        'Bacillaceae':
        {
            'Bacillaceae_____Anoxybacillus': {}
        }
    }
    'annot_2.txt':
    {
        'Listeriaceae':
        {
            'Listeriaceae_____Listeria':
            {
                'Listeriaceae_____Listeria_____Lgrayi': {},
                'Listeriaceae_____Listeria_____Linnocua': {},
            }
        }
    }
}
```

multi_flat_tax_tree: A dict of dicts, where:

- keys are filenames,
- values are dicts, where:

- keys are names of nodes,
- values are numbers of nodes, with names starting from "node_name"

Example:

```
{
    'annot_1.txt':
    {
        'Bacillaceae': 8,
        'Bacillaceae_____Anoxybacillus': 8
    },
    'annot_2.txt':
    {
        'Listeriaceae': 16,
        'Listeriaceae_____Listeria': 16,
        'Listeriaceae_____Listeria_____Lgrayi': 8,
        'Listeriaceae_____Listeria_____Linnocua': 8
    }
}
```

Return type total_tax_tree, per_file_tax_tree, multi_flat_tax_tree

taxa_read(*read_mode*, *db_loc=None*)

Returns `{GI:TaxID} & {TaxID:scientific_name}` dicts.

Parameters

- **read_mode** –

  if read_mode is equal to 'manual', then: *tax_id_reader()* and *tax_name_reader()* functions will be used and data is read from text files. Refer to mentioned functions for more information.

  otherwise: *auto_tax_read()* function (with *db_loc* as an argument) will be used and data will be read from a pickle file. Refer to mentioned function for more information.

- **db_loc** – path to pickle file, used when *read_mode* is other than 'manual'

Returns

- `{GI:TaxID}` Keys are integers, values are strings.
- `{TaxID:scientific_name}` Keys and values are integers.

Return type Two dicts

tree_of_life(*full_dict*)

Rewrites the data from a dict containing information arranged by type and then by file into two dicts:

•first represents full taxonomic tree

•second presents how many species are inside particular files

Parameters **full_dict** – a dict of dicts, with structure like:

```
{
    'output_type_1':
    {
        'filename_1.ext':
        {
            'Bacillaceae': {'Anoxybacillus': {'subsum': 15}}
```

```
                    }
                }
            }
```

Longer example available in description of *xml_format()*.

Returns

A tuple (full_tree, file_total_count)

full_tree: A dict without information about type and file.

Example:

```
{
    'Bacillaceae':
    {
        'Anoxybacillus': {}
    },
    'Listeriaceae':
    {
        'Listeria':
        {
            'Lgrayi': {}, 'Linnocua': {}
        }
    }
}
```

file_total_count: A dict within information about files are kept and values of every leaf are summed into values representing total count of species inside particular file.

Example:

```
{'filename_1.ext': 15, 'filename_2.ext': 33}
```

tuple_to_dict (*tuple_dict*)

Input dict has tuples as keys and int type values. In the dict returned by function:

- keys are elements from input dict's tuples,
- values are dicts.

Parameters **tuple_dict** – a dict of tuples

Example:

```
{('first','second','third'): 10, ('first','second'): 30, ...}
```

Example will show why this function can be useful:

input:

```
{
    ('Animalia','Mammalia','Canis lupus familiaris'): 10,
    ('Bacteria','Enterobacteriaceae','Escherichia coli'): 20,
    ('Animalia','Mammalia','Felis catus'): 30
}
```

output:

```
    {
        'Animalia':
        {
            'Mammalia':
            {
                'Canis lupus familiaris': {'subsum': 10},
                'Felis catus': {'subsum': 30}
            }
        }
        'Bacteria':
        {
            'Enterobacteriaceae':
            {
                'Escherichia coli': {'subsum': 20}
            }
        }
    }
```

tuple_to_xml_dict(*tuple_dict*)

Input dict has tuples as keys and int type values. In dict returned by function all elements from input dict's tuples are single keys but their current values are the sum of the values of all tuples, in which they were.

Parameters **tuple_dict** – tuple to be modified

Example:

```
            {
                ('first','second','third'): 10,
                ('first','third','fifth': 30)
                ...
            }
```

Example

input:

```
    {
        ('Animalia','Mammalia','Canis lupus familiaris'): 10,
        ('Bacteria','Enterobacteriaceae','Escherichia coli'): 20,
        ('Animalia','Mammalia','Felis catus'): 30
    }
```

output:

```
    {
        'Animalia': 40,
        'Felis catus': 30,
        'Enterobacteriaceae': 20,
        'Mammalia': 40,
        'Escherichia coli': 20,
        'Bacteria': 20,
        'Canis lupus familiaris': 10
    }
```

txt_dict_clean(*dicto*)

Cleans given file structure by removing files which do not contain words "graplan" or "tree". Emptied directories are also removed.

Parameters **dicto** –

A dict where:

- keys are names of directories,
- values are lists with filenames, which are in the directory.

Example:

```
{'dir_name_1': ['file_1','file_2'], 'dir_name_2': ['file_1']}
```

Explicit example in description of output of *cat_read()*.

Returns A dict in the same format, as given one (without unwanted files)

update_dict (*tax_tree*, *curr_tax*)

Updates the dict with keys form another one. If during walking the dict, the 'subsum' string is found, recurrence stops and nodes beneath are ignored. The dicts have to contain only other dicts or 'subsum'!

Parameters

- **tax_tree** – a dict to update
- **curr_tax** – a dict to be merged into the *tax_tree*

Returns updated dict

Return type tax_tree

Example

input (tax_tree, curr_tax):

```
(
    {
        'Listeriaceae':
        {
            'Listeria': { 'Lgrayi': {} }
        }
    },
    {
        'Listeriaceae':
        {
            'Listeria': { 'Linnocua': {'subsum': 11} }
        }
    }
)
```

output:

```
{
    'Listeriaceae':
    {
        'Listeria':
        {
            'Lgrayi': {},
            'Linnocua': {}
        }
    }
}
```

usearch (*mode*, *e*, *search_type*, *infile*, *database*, *threads*)

Launches Usearch with -usearch_local command.

HARDCODED:

- **Usearch options:**
  - evalue = 0.01
  - id = 0.9
  - strand = both

Parameters

- **mode** – if *mode* is 'run' Usearch will be launched.
- **e** – if true, checks if a part of workflow is actually done and doesn't duplicate this jobs.

  For more information, please refer to e argument in *exist_check()* function.
- **search_type** – &
- **infile** – a name of input file for usearch. Also determines name of [-blast6out in Usearch] output file.
- **database** – [-db in Usearch]
- **threads** – [-threads in Usearch] Number of threads used in calculations.

GLOBALS: PATH_USEARCH

xml_counts_graphlan (*per_file_tax_tree*, *xml_names*, *multi_flat_tax_tree*)

Groups and/or sums numbers of nodes (obtained from multi_flat_tax_tree):

1. by node_name - grouping
2. by identifier (obtained from filename) - summing

Parameters

- **per_file_tax_tree** – A dict of dicts, etc. Top-level keys are filenames. Taxonomic tree represented by nested dicts. Similar to tax_tree. Example included in description of *tax_tree_graphlan()*.
- **xml_names** – readable identifiers of files derived from filenames. Example:

```
['filename_1', 'filename_2']
```

- **multi_flat_tax_tree** –

  A dict of dicts, where:
  - keys are filenames,
  - **values are dicts, where:**
    - keys are names of nodes,
    - values are numbers of nodes with names starting from "node_name"

  Explicit example in description of *tax_tree_graphlan()*.

Returns

A tuple – xml_dict: A dict. Contains lists of numbers of occurrences of nodes by node.

- keys are names of nodes.

- values are lists of constant length, where every item on position *X* means: number of occurrences of nodes with prefix equal to *name of node* in file *X*. *file X* is that file, which is on position *X* on list xml_names.

Example:

```
{node: [count_1, count_2], node_2: [count_1, count_2]}
```

name_total_count: A dict. Contains summed numbers of occurrences of nodes by file.

- keys are identifiers (derived from filenames - look for xml_names),
- values ale sums.

Example:

```
{identifier_1: count_1, identifier_2: count_1}
```

Return type xml_dict, name_total_count

xml_format (*full_dict*, *tax_dict*)

Creates a set of dicts, which are useful to generate xml files from.

Parameters

- **full_dict** – a dict of dicts with structure like:

```
{
    'output_type_1':
    {
        'filename_1.ext':
        {
            'Bacillaceae': {'Anoxybacillus': {'subsum': 15}}
        },
        'filename_2.ext':
        {
            'Listeriaceae':
            {
                'Listeria':
                {
                    'Lgrayi': {'subsum': 22},
                    'Linnocua': {'subsum': 11}
                }
            }
        }
    },
    'output_type_2':
    {
        'filename_2.ext':
        {
            'Bacillaceae': {'Anoxybacillus': {'subsum': 15}}
        }
    }
}
```

"output type" typically will be 'ITS', '16S'.

- **tax_dict** –

  A dict, where:

  - keys are file identifiers,

- **values are dicts, where:**
  - keys are names of taxa,
  - values are numbers of occurrences of particular taxon.

Example:

```
{
    'filename_1.ext':
    {
        'Bacillaceae': 5, 'Anoxybacillus': 5
    },
    'filename_2.ext':
    {
        'Bacillaceae': 1, 'Anoxybacillus': 1,
        'Listeriaceae': 19, 'Listeria': 19,
        'Lgrayi': 16, 'Linnocua': 3
    }
}
```

Returns

(xml_names, xml_dict, tax_tree, name_total_count)

xml_names: readable identifiers of groups of files referring to common dataset; derived from filenames.

Example:

```
['filename_1', 'filename_2']
```

xml_dict:

a dict, where:

- keys are names of taxa,
- values are lists with counters of occurences of particular taxon in subsequent groups of files. Order on this list is defined by order of names in xml_names list.

Example:

```
{
    'Anoxybacillus': [5, 1],
    'Bacillaceae': [5, 1],
    'Lgrayi': [0, 16],
    'Linnocua': [0, 3],
    'Listeria': [0, 19],
    'Listeriaceae': [0, 19]
}
```

tax_tree: a dict without information about type and file.

Example:

```
{
    'Bacillaceae':
    {
        'Anoxybacillus': {}
    },
    'Listeriaceae':
    {
```

```
                        'Listeria':
                        {
                            'Lgrayi': {}, 'Linnocua': {}
                        }
                    }
                }
```

name_total_count: a dict, where keys are names as present in xml_names and values are summed total counts (usually of occurrences of different species).

Example:

```
                {'filename_1': 5, 'filename_2': 20}
```

Return type A tuple

xml_name_parse(*full_dict*)

Creates a list of simplified filenames (full filenames comes from *full_dict*). The list is guaranteed to not have any duplicates.

If filename contains sequence of nucleotides from set {A, C, T, G}, separated from other parts of the name by underscore (_), then simplified filename will be substring of filename from beginning to this sequence (inclusive). This set of nucleotides in most cases will represent "index" from naming schema for fastq files, so since this step, many files will belong to one alias.

Otherwise, the first chunk of filename will be used. In this case, filename will be splitted with dot '.' as delimiter. It also indicates grouping files with similar filenames since this step.

Keep in mind, that this step groups files referring to the same dataset under single name, and technically is vulnerable for errors: If filenames given to this functions do not follow appropriate naming schema or when these filenames are not named in prefix-code convention, then some false-positives might be generated in some of functions which use results of xml_name_parse().

Example

Following filenames:

```
'16S_ArchV3V4_M_BF_02_TAGCTT_L001_001.amplicons.cutadapt.flash.merged.fastq.extendedFrags.fasta.
'16S_ArchV3V4_M_BF_02_TAGCTT_L001_001.cutadapt.amplicons.cutadapt.flash.merged.fastq.extendedFra
'Amp45_BFp_B_CAAAAG_L001_R12.fasta.usearch_ITS'
'meta-velvetg.contigs.fa.usearch_ITS'
```

will become:

```
'16S_ArchV3V4_M_BF_02_TAGCTT' # note: this groups first two files
'Amp45_BFp_B_CAAAAG'
'meta-velvetg'
```

Parameters **full_dict** – a dict of dicts, with structure like:

```
            {
                'output_type_1':
                {
                    'filename_1.ext':
                    {
                        'Bacillaceae': {'Anoxybacillus': {'subsum': 15}}
                    }
```

```
        }
    }
```

Longer example avaliable in description of *xml_format()*.

Returns

A list with simplified filenames. The list is guaranteed to not have any duplicates.

Example:

```
['filename_1', 'filename_2']
```

xml_names_graphlan(*input_d*)

Extracts values from given dict, puts the values on a list, and then removes from every value those parts that are prefixes and suffixes common for all items on the list.

Parameters **input_d** – A dict where: keys are names of directories, values are lists with filenames, which are in the directory.

Example

{'dir_name_1': ['file_1','file_2'], 'dir_name_2': ['file_1']}

Explicit example in description of output of *cat_read()*.

Returns A list with all filenames which were given on input, trimmed by common prefixes and suffixes.

xml_prepare(*xml_names*, *xml_dict*, *tax_tree*, *name_total_count*, *unit='reads'*)

Generates XML string, dedicated to use with Krona. XML is generated with use of default python xml module.

Parameters

- **xml_names** – a list of readable identifiers of datasets.

  Example:

```
['filename_1', 'filename_2']
```

- **xml_dict** – a dict with lists of occurrences of nodes by node. Keys are names of nodes. Values are lists of constant length, where every item on position *X* means: number of occurrences of node in dataset on position *X* on list xml_names.

  Note, that for all i: len(xml_dict[i]) is equal to len(xml_names).

  Example:

```
{node_1: [count_1, count_2, ... count_x],
node_2: [count_1, count_2, ... count_x]}
```

- **tax_tree** – a dict of dicts, etc. In form of nested dicts, represents phylogenetic tree. Example:

```
{a:{aa:{}, ab:{aba:{}, abb:{abba:{}}}}}
```

- **name_total_count** – a dict with numbers of occurrences of nodes by dataset. Keys are names of dataset from xml_names, values are counts. Example:

```
{identifier_1: count_1, identifier_2: count_2}
```

- **unit** – A string. Defines units to be used in the Krona.

Returns A single string with generated XML.

HARDCODED: Number of levels to iterate in tax_tree is hardcoded to 9. As original comment states: 'however it ignores all absent levels! which is actually kind of cool'

xml_vals (*xml_names*, *tax_dict*)

Reformats information extracted from tax_dict into another dict, to allow use of this data in xml-generation process.

Parameters

- **xml_names** – readable identifiers of groups of files referring to common dataset, derived from filenames.

  Example:

```
['filename_1', 'filename_2']
```

- **tax_dict** –

  A dict, where:

  - keys are file identifiers,
  - **values are dicts, where:**
    - keys are names of taxa,
    - values are numbers of occurrences of particular taxon.

  Example:

```
{
    'filename_1.ext':
    {
        'Bacillaceae': 5, 'Anoxybacillus': 5
    },
    'filename_2.ext':
    {
        'Bacillaceae': 1, 'Anoxybacillus': 1
        'Listeriaceae': 19, 'Listeria': 19, 'Lgrayi': 16, 'Linnocua': 3
    }
}
```

Returns: dict:

A dict, where:

- keys are names of taxa,
- values are lists with counters of occurrences of particular taxon in subsequent groups of files. Order on this list is defined by order of names in xml_names list.

Example:

```
{
    'Anoxybacillus': [5, 1],
    'Bacillaceae': [5, 1],
    'Lgrayi': [0, 16],
    'Linnocua': [0, 3],
    'Listeria': [0, 19],
    'Listeriaceae': [0, 19]
}
```

Warning: Keep in mind, that this step - along with generation of xml_names in *xml_name_parse()* - groups files referring to the same dataset under a single name, and technically is vulnerable for some errors: If filenames given to these functions do not follow appropriate naming schema or when these filenames are not named in prefix-code convention, then some false-positives might be generated when grouping results and this will influence results.

CHAPTER 2

CLI

2.1 Command line options

2.1.1 General

Performance and I/O related options.

Help

-h, --help

Show help message and exit.

Adjusting the number of used threads

-t <threads>, **--threads** <threads>

Number of threads to be used.
Default: 8

Mode of running

-m <mode>, **--mode** <mode>

A mode in which the program will be run.
Available modes: test, run
Default: test

Directory with input files

-r <root_directory>, **--root_dir** <root_directory>

Root of the directory tree to be searched.
Default: ~/bipype

Directory for output files

-o <output_directory>, **--out_dir** <output_directory>

Indicates, where the output files should be located.
To specify current working directory, use 'in_situ'.
Otherwise type path to chosen directory.
Default: in_situ

Insert length

--ins_len <insert_length>

Lenght of instert.
Be advised - better use it for single run.
Default: 9999

Processed file postfix

-postfix <postfix>

Alphanumerical postfix of processed file
Default: (empty string)

Continuation of work, after aborting or using manually improved partial results

-e

Tells bipype, to use existing files with partial results, so parts which are already done (for example by previous, killed instance of program) will be incorporated into pipe. It also creates possibility of manually improving existing files.
Default: False

2.1.2 Taxonomy stats

Options related to prepare_taxonomy_stats results.

Output type

-ot <output_type_list>, **--output_type** <output_type_list>

Simultaneously defines input type!
Allows to choice on which files (for example ITS, 16S, map_count) the analysis will be performed and also determines basenames of output files. Names of input files should end with (respectively) .usearch_ITS or .usearch_16S, .map_count.
Default: ['ITS, 16S']

2.1.3 Input cleaning

Methods for cleaning input from noise.

Initial cleaning

-ic <method>, **--initial_cleaning** <method>

Initial cleaning method
Available methods: usearch, fastx
Default: (empty string)

Adapters cutting

--cutadapt <adapter_file> <search_options>

Location of file with adapters to be used by cutadapt (possible use of "use_filenames" to determine adapters from hardcode), and list of usearches to be run on created files - possible options are 16S, ITS, both. Please note, that mapping options -16S, -ITS are completely irrelevant if you use cutadapt. Other note - this is !!!IMPORTANT!!! to present location of file with adapters as first option of this argument
Default: (empty string)

2.1.4 Mappings

Mappings to be done during the run.

ITS usearch

-ITS

Perform usearch on ITS database

16S usearch

-16S

Perform usearch on 16S database

RefSeq

-refseq <kingdom>

Map samples on refseq.
Available kingdoms: p, f, b
p states for plantae, f for fungi, b - both
Default: f

2.1.5 Contig reconstruction

Methods of contig reconstruction to be used during the run.

MetaVelvet

-MV <parameters>

Parameters for MetaVelvet.
Parameters format:
[initial_k_mer_size, final_k-mer_size, step]

Reconstruct

-reconstruct <option>

Reconstruct relating to database.
Available options: database_loc, prefix

Humann

-humann

Mapping rapsearch using humann (default: None)

Rapsearch

-rapsearch <database>

Perform RAPsearch on selected protein database
Available databases: masl28,rap_prot,rap_KO
Default: rap_prot

2.1.6 Databases

Locations of databases.

16S for usearch

--db_16S <database>

16S database to use in usearch (bowtie indexed)
Default: ${PATH_X16S_DB}

ITS for usearch

--db_ITS <database>

ITS database to use in usearch (bowtie indexed)
Default: ${PATH_ITS_DB}

Fungi for refseq

--db_refseq_fungi <databases>

Refseq database to use for fungi analysis. All files found under your_database_path/*.some_suffix path will be loaded and treated as subparts of your database.
Up to two paths are allowed (paths should be separated with space).
Default: ${PREF_PATH_REF_FUNGI}

Plant for refseq

--db_refseq_plant

Refseq database to use for plants analysis. All files found under your_database_path/*.some_suffix path will be loaded and treated as subparts of your database.
Up to two paths are allowed.
Default: ${PREF_PATH_REF_PLANT_1} ${PREF_PATH_REF_PLANT_2}

Taxonomy database

--db_NCBI_taxonomy

Location of cPickled database with mappings of NCBI_tax_id, NCBI tax names and NCBI tax ids.
Default: ${PATH_NCBI_TAXA_DB}

Database for reconstruction

--db_reconstruct

This database will be passed as parameter to bwa program.
Format: Fasta file.
Default: ${PATH_RECONSTRUCT_DB}

Database for ITS taxonomy statistics

--db_taxonomy_16S

This database is used to create a taxonomy tree for results visualisations.
By default it is specially formatted FASTA file.
For more informations check "databases formatting" chapter.
Default: ${PATH_16S_DATABASE}

Database for ITS taxonomy statistics

--db_taxonomy_ITS

This database is used to create a taxonomy tree for results visualisations.
By default it is specially formatted FASTA file (with headers like in UNITE database).
For more informations check "databases formatting" chapter.
Default: ${PATH_ITS_DATABASE}

2.2 Configuration File

All commands may be presented in a configuration file fed to bipype with @ prefix. A file with configuration should contain all desired commands and their options (if applicable) one per line.

2.2.1 Example 1

`bipype @fancy_filename`

Content of file with fancy filename:

```
-e
--mode run
--threads 4
--output_type 'ITS'
```

So, this is equivalent to

`bipype -e --mode run --threads 4 --output_type 'ITS'`

2.2.2 Example 2

```
bipype @another_fancy_filename
```

Content of file with another fancy filename

```
--mode run
--threads 4
```

So, this is equivalent to

```
bipype -e --mode run --threads 4 --output_type 'ITS'
```

CHAPTER 3

DESCRIPTION

3.1 DESCRIPTION OF PROGRAM OPTIONS

Bipype is a very useful program, which prepare a lot of types of bioinformatics analyses. There are three input options: amplicons, WGS (whole genome sequences) and metatranscriptomic data. If amplicons are input data, then bipype does reconstruction and pairs merging. After that biodiversity is searching. There are two types of searching depending on the amplicons types (ITS or 16S). If WGS are chosen, then bipype finds the SA coordinates of the input reads and generates alignments in the SAM format given single-end reads, aligns reads to reference sequence(s). All of these analyses will be shown with Krona program, which allows to show hierarchical data with pie charts.

bipype stands for bioinformatics-python-pipe.

3.1.1 Proposed analysis

Below are listed a few analysis which could be performed with bipype program. As input for all following analysis FASTA files should be given.

Metagenomics

output: krona.html

- **Analysis 1**

```
bipype -m run -rapsearch rap_KEGG -humann
bipype -m run --order prepare_taxonomy_stats -ot txt
```

- **Analysis 2**

This analysis can be done in the same way as presented in case of analysis of amplicons.

Amplicons

- **Both (16S and ITS)**

output: 16S_ITS.krona

```
bipype -m run --cutadapt use_filenames ITS 16S
bipype -m run --order taxonomy_stats -ot ITS 16S
```

- **Only ITS**

output: ITS.krona

```
bipype -m run --cutadapt use_filenames ITS
bipype -m run --order taxonomy_stats -ot ITS
```

- **Only 16S**

output: 16S.krona

```
bipype -m run --cutadapt use_filenames 16S
bipype -m run --order taxonomy_stats -ot  16S
```

Metatranscriptomics

3.1.2 More detailed description of bipype workflow

Bipype consists of three major, parts:

- **sample**, which performs most analysis and runs other programs
- **taxonomy_stats**, which generates taxonomy stats in Krona format
- **metatranscriptomics** TODO

Sample

General schema of workflow

1. Samples mapping
2. Alignment reads to reference sequence(s)
3. Determining the presence/absence and abundance of microbial pathways

Samples mapping

This step checks to which species and taxonomic group given sample belongs. It searches databases for fungi, plants or both, accordingly to value of <to_calculate> option:

- p - plant,
- f - fungi,
- b - both

Taxonomy database loading

By default the taxonomy database is loaded from path defined by <db_NCBI_taxonomy> option. This should by a "pickle" file and by default it is:

${PATH_NCBI_TAXA_DB}

Databases will be loaded only if mode is "run".

What if mentioned, default taxonomy database doesn't exists?

Then, there are loaded other, also default databases:

Database	default path
GI → TaxID	${PATH_GI_TAX}
TaxID → scientific_name	${PATH_TAX_NAME}

Formatting of these databases are described in appendix "databases formatting".

How to replace default taxonomy database?

To load your own database, you need to supersede the <db_NCBI_taxonomy> option with the path to file, where pickled dict with database is saved.

One of possible ways to create this database:

1. Modify databases: "GI → TaxID" and "TaxID → scientific_name", (discussed below), and then
2. Load modified databases and pickle them to new file (here: 'new_db'), for example with use of the following python script:

```
#!/usr/bin/python
from refseq_bipype import taxa_read
import cPickle as pickle
with open('new_db', 'wb') as output_file:
        list_with_dicts = taxa_read("manual")
        pickle.dump(list_with_dicts, output_file)
```

Getting input files

Finds in current working directory and subdirectories fastq files, which are "paired_end". If mode is run, also unpacks compressed archives in search of input files.

Mapping

A list of path to refseq databases is given in db_refseq_fungi and db_refseq_plant parameters for plant and fungi analysis. Up to two paths are allowed, so for one path program will add "False", as a second element.

Reconstruct option:

If that option was chosen, bwa (Burrows-Wheeler Alignment Tool) program is run. The program finds the SA coordinates of the input reads and generates alignments in the SAM format given single-end reads. Repetitive hits will be randomly chosen. If mode "run" was chosen, commands "bwa aln", "bwa samse" and "samtools mpileup" were run. Bwa aln command:

aln	bwa aln [-n maxDiff] [-o maxGapO] [-e maxGapE] [-d nDelTail] [-i nIndelEnd] [-k maxSeedDiff] [-l seedLen] [-t nThrds] [-cRN] [-M misMsc] [-O gapOsc] [-E gapEsc] [-q trimQual] <in.db.fasta> <in.query.fq> > <out.sai>

Find the SA coordinates of the input reads. Maximum *maxSeedDiff* differences are allowed in the first *seedLen* subsequence and maximum *maxDiff* differences are allowed in the whole sequence. Parameter t (number of threads) is taken from reconstruct function (thr parameter), all others will be default.

Bwa samse command:

samse	bwa samse [-n maxOcc] <in.db.fasta> <in.sai> <in.fq> > <out.sam>

Generate alignments in the SAM format given single-end reads. Repetitive hits will be randomly chosen. All parameters will be default. Also SAMtools program is run. For example, we have reference sequences in ref.fa, indexed by samtools faidx, and position sorted alignment file aln.bam, the following command lines call SNPs and short INDELs:

samtools mpileup -uf ref.fa aln.bam | bcftools view -cg - | vcfutils.pl vcf2fq > cns.fq

Alignment reads to reference sequence(s)

After choosing fungi or plant, program runs refseq_mapping function with appropriate parameters for this type of input. Biopype uses Bowtie 2 program to align reads to reference sequence(s). Firstly, Bowtie 2 aligns reads to reference sequence (or sequences, if refseq_2 is not False and merge output SAM files). Secondly, BAM files are made, sorted and indexed. Thirdly, Function launches 'samtools idxstats'. Finally, idx_map() function is called: parse data from samtools idxstats output file and writes data to outfile in key; value format, where: GI/TaxID/scientific name

→ number of mapped reads. For fungies and plants, different basenames of the indexes for the reference genome are used as refseq parameter for refseq_mapping function.

There are values for adapters‘ list (cutadapt parameter): ‘both’, ‘ITS’, ‘16S’. If list of adapters types isn‘t empty, then program searches for the adapters in reads from input files, removes them, when it finds them and writes to output files which have .cutadapt.fastq extension. Then function runs FLASH (software tool to merge paired-end reads) with cutadapt.fastq files as input and .amplicons.cutadapt.flash.merged.fastq files as output. These results are input for fastq_to_fasta which converts file format from fastq to fasta. If Velvet wasn‘t choosen, then rapsearch option is available. RAPSearch will be run with default KEGG=None, if ‘rap_prot’ option is in to_calculate option. RAPSearch will be runned with options:

-z 10 -e 0.001 -b 100 -v 100 -g T -a T 1, if mode is ‘run’. If ‘rap_KEGG’ or ‘rap_KO’ are in to_calculate list, then all files .fastq will be changed for .fasta. If rap_KEGG is in to_calculate list, then rapsearch will be runned with KEGG=’masl’. In other case, rapsearch will be runned with KEGG=’KO’. With this parameter RAPSearch will be runned with options: -z 12 -v 20 -b 1 -t n -a t 1, if mode is ‘run’.

Determining the presence/absence and abundance of microbial pathways

HUMAnN If ‘humann’ is in to_calculate list, then program checks if m8 files exist. In that case humann function will be runned and new catalog humann-0.99 will be created in rapsearch result folder. This function copies HUMAnN program to the current directory, moves input (*.m8) files to the input directory, copies hmp_metadata.dat file to the input directory and runs HUMAnN. HUMAnN is a pipeline for efficiently and accurately determining the presence/absence and abundance of microbial pathways in a community from metagenomic data.

In that case, human function will be runned and analysis results will be added in rapsearch result folder. If ‘16S’ or ‘ITS’ are in to_calculate list (or both of them), then usearch function will be runned for these types of sequences. Usearch function runned USEARCH command, if mode=’run’:

-usearch_local [katalog z USEARCH] -db [input file] -evalue 0.01 -id 0.9 -blast6out [output file] -strand both -threads [threads (integer)]

Preparing taxonomy stats

Performs statistical analysis of taxonomy from appropriate files from current working directory: counts occurrences of different taxa and prepares the results to be presented in HTML format.

Results will be converted to HTML (with the Krona program), but only when <mode> is set to ‘run’.

Takes following options: output_type, mode, out_dir.

Input

Analysis will be performed on files, which simultaneously:

- are located in current working directory or in subdirectories,
- have suffixes of filenames equal to value of <output_type> option,
- have suffixes of filenames equal to ‘usearch_’ + value of <output_type> option, but only if <output_type> is ‘ITS’ or ‘16S’.

Output

Output files will be placed in <out_dir> directory, with basenames depending on <output_type>.

Examples:

Given <output_type>	Filename of krona file	Filename of html file
ITS	ITS.krona	ITS.html
ITS, 16S	ITS_16S.krona	ITS_16S.html

Databases

Bipype gets two separate databases with taxonomy to use in this step - one for ITS and one for 16S sequences. Custom paths to these databases can be passed with –db_taxonomy_16S and –db_taxonomy_ITS parameters.

Databases for this step should be specially formatted FASTA files, or - especially if you want to allow faster loading of this data - files including only header lines from these FASTA files.

For more information, refer to “databases formatting” chapter.

CHAPTER 4

Appendixes

4.1 Database formatting

4.1.1 16S taxonomic database

This database should be given as FASTA file.
Format of headers for 16S should be like in following example:

Example:

```
>AF093247.1.2007 Eukaryota;Amoebozoa;Mycetozoa;Myxogastria;;Hyperamoeba_sp._ATCC50750
```

4.1.2 ITS taxonomic database

This database should be given as FASTA file.
Format of headers for ITS is the same as one used in UNITE database:

Example:

```
>DQ233785|uncultured ectomycorrhizal fungus|Fungi|Thelephora terrestris|Fungi; Basidiomycota; Agaric
```

4.1.3 Database "GI → TaxID"

This database should be prepared in tsv (tab-separed values) format.
First column is a GI, second is a TaxID.

Example:

```
13   9913
15   9915
16   9771
17   9771
```

4.1.4 Database "TaxID → scientific_name"

From file formated as in example below:

```
2    |       prokaryotes     |       prokaryotes <Bacteria>  |       in-part |
6    |       Azorhizobium    |               |       scientific name |
6    |       Azorhizobium Dreyfus et al. 1988        |               |       synonym |
6    |       Azotirhizobium  |               |       equivalent name |
7    |       ATCC 43989      |               |       type material   |
7    |       Azorhizobium caulinodans        |               |       scientific name |
```

Only following data will be extracted:

TaxID	scientific_name
6	Azotirhizobium
7	Azorhizobium caulinodans

4.2 Paths constants

4.2.1 For refseq_bipype.py

Constant	Path
PATH_GI_TAX	/home/pszczesny/workingdata/refseq/db/tax_id/gi_taxid_nucl.dmp
PATH_TAX_NAME	/home/pszczesny/workingdata/refseq/db/tax_id/names.dmp
PATH_RAPSEARCH	/home/pszczesny/soft/RAPSearch2.12_64bits/bin/rapsearch
PATH_REF_PROT_MASL28	/home/pszczesny/storage/workingdata/rapsearch/masl28
PATH_REF_PROT_KO	/home/pszczesny/soft/KEGG/ko.pep.rapsearch.db
PATH_REF_PROT_ELSE	/home/pszczesny/workingdata/refseq/protein/refseq_protein
PATH_USEARCH	/home/pszczesny/soft/usearch
PATH_FQ2FA	/usr/local/bioinformatics/fastx_toolkit/bin/fastq_to_fasta
PATH_HUMANN	/home/pszczesny/soft/humann-0.99
PATH_HUMANN_DATA	/home/pszczesny/soft/humann-0.99/input/hmp_metadata.dat
PATH_BOWTIE2	/home/pszczesny/bin/bowtie2
PATH_VELVETH	/home/pszczesny/soft/Velvet/velveth
PATH_VELVETG	/home/pszczesny/soft/Velvet/velvetg
PATH_METAVELVETG	/home/pszczesny/soft/MetaVelvet/meta-velvetg
PATH_MEGAHIT	/home/pszczesny/soft/megahit/megahit

4.2.2 Bipype

Constant	Path
PREF_PATH_REF_FUNGI	/home/pszczesny/workingdata/refseq/db/fungi/fungi_refseq
PREF_PATH_REF_PLANT_1	/home/pszczesny/workingdata/refseq/db/plant/plant_refseq_1
PREF_PATH_REF_PLANT_2	/home/pszczesny/workingdata/refseq/db/plant/plant_refseq_2
PATH_X16S_DB	/home/public/EMIRGE/SSU/SSU_candidate_db.udb
PATH_ITS_DB	/home/pszczesny/workingdata/ITS/unite.udb
PATH_NCBI_TAXA_DB	/home/pszczesny/soft/bipype/NCBI_tax_id_name.pickle
PATH_RECONSTRUCT_DB	/home/pszczesny/storage/workingdata/mcra/mcra10k.fasta
PATH_16S_DATABASE	/home/pszczesny/soft/bipype/SSU_candidate_db.fasta
PATH_ITS_DATABASE	/home/pszczesny/soft/bipype/UNITE_public_from_27.01.13.fasta

4.2.3 Metatranscriptomics

Constant	Path
PATH_KO_DB	/home/pszczesny/soft/KEGG/ko.db
PATH_KO_PCKL	/home/pszczesny/soft/bipype/kogenes.pickle

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

b

m

r

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

X