AMAS_JGLAHE

Generated by Doxygen 1.9.8

Chapter 1

AMAS

Alignment manipulation and summary statistics

If you are using this program, please cite this publication:
Borowiec, M.L. 2016. AMAS: a fast tool for alignment manipulation and computing of summary statistics. PeerJ 4:e1660.

1.1 This fork: AMAS_JGLAHE

A standalone version of the main repo that's been modified to fit my needs, notably:

- a `metapartition` command -> collates discontinuous metapartitions within a superalignment and concatenates them into a new superalignment of contiguous metapartitons.
- less restrictions on partition file formatting -> accepts partition files for RAxML(-NG) and IQ-TREE2 (best_↩ scheme, best_scheme.nex and best_model.nex).
- minor formatting changes for pipeline inegration.

1.2 Installation

Use `AMAS.py` in the `amas` directory as a stand-alone program or clone it if you have `git installed` on your system.

If your system doesn't have a Python version 3.4 or newer (`AMAS` will work under Python 3.0 but you may noy be able to use it with multiple cores), you will need to `download and install it`. On Linux-like systems (including Ubuntu) you can install it from the command line using

It may be possible to use this version as a module, but only through manual configuration.

1.3 Command line interface

AMAS can be run from the command line. Here is the general usage (you can view this in your command line with python3 AMAS.py -h):

```
usage: AMAS <command> [<args>]

The AMAS commands are:
  concat            Concatenate input alignments.
  convert           Convert to other file format.
  replicate         Create replicate data sets for phylogenetic jackknife.
  split             Split alignment according to a partitions file.
  summary           Write alignment summary.
  remove            Remove taxa from alignment.
  translate         Translate DNA alignment into protein alignment.
  trim              Remove columns from alignment.
  metapartitions    Runs `split' and concatenates the output.

Use AMAS <command> -h for help with arguments of the command of interest

positional arguments:
  command     Subcommand to run

optional arguments:
  -h, --help  show this help message and exit
```

To show help for individual commands, use AMAS.py <command> -h or AMAS.py <command> --help.

1.3.1 Examples

For every AMAS.py run on the command line you need to specify action with concat, convert, replicate, split, or summary for the input to be processed. Additionally, you need to provide three arguments required for all commands. The order in which the arguments are given does not matter:

1) input file name(s) with -i (or in long version: --in-files),

2) format with -f (--in-format),

3) and data type with -d (--data-type).

The options available for the format are fasta, phylip, nexus (sequential), phylip-int, and nexus-int (interleaved). Data types are aa for protein alignments and dna for nucleotide alignments.

For example:

```
python3 AMAS.py concat -i gene1.nex gene2.nex -f nexus -d dna
```

If you have many files that you want to input in one run, you can use multiple cores of your computer to process them in parallel. The summary command supports -c or --cores with which you can specify the number of cores to be used:

```
python3 AMAS.py summary -f phylip -d dna -i *phy -c 12
```

In the above, we specified 12 cores. Note that this won't improve computing time if you're working with only one or very few files. The parallel processing is only used for the file parsing step and calculating alignment summaries.

In addition to overall alignment summaries, you can also print statistics calculated on a sequence (taxon) by sequence basis. Use -s or --by-taxon flag to turn it on. AMAS in this mode will print out one file with overall alignment summaries and a file with taxon summaries for each input alignment.

IMPORTANT! AMAS is fast and powerful, but be careful: it assumes you know what you are doing and will not prevent you overwriting a file. It will, however, print out a warning if this has happened. AMAS was also written to work with aligned data and some of the output generated from unaligned sequences won't make sense. Because of computing efficiency AMAS by default does not check if input sequences are aligned. You can turn this option on with -e or --check-align.

1.3.1.1 Concatenating alignments

For example, if you want to concatenate all DNA phylip files in a directory and all of them have the `.phy` extension, you can run:

```
python3 AMAS.py concat -f phylip -d dna -i *phy
```

By default the output will be written to two files: `partitions.txt`, containing partitions from which your new alignment was constructed, and `concatenated.out` with the alignment itself in the fasta format. You can change the default names for these files with `-p` (`--concat-part`) and `-t` (`--concat-out`), respectively, followed by the desired name. The output format is specified by `-u` (`--out-format`) and can also be any of the following: `fasta`, `phylip`, `nexus` (sequential), `phylip-int`, or `nexus-int` (interleaved).

Below is a command specifying the concatenated file output format as nexus with `-u nexus`:

```
python3 AMAS.py concat -f fasta -d aa -i *fas -u nexus
```

Alignments to be concatenated need not have identical sets of taxa before processing: the concatenated alignment will be populated with missing data where a given locus is missing a taxon. However, if every file to be concatenated includes only unique names (for example species name plus sequence name: `D_melanogaster_NW_001845408.1` in one alignment, `D_melanogaster_NW_001848855.1` in other alignment etc.), you will first need to trim those names so that sequences from one taxon have equivalents in all files.
In addition to the name, you can also specify the format of the partitions output file. By default, the format is the following:

```
AA = 1-605
AK = 606-1200
28S = 1201-1800
```

RAxML:

```
python3 AMAS.py concat -f phylip -d dna -i *phy --part-format raxml
DNA, AA = 1-605
DNA, AK = 606-1200
DNA, 28S = 1201-1800
```

Nexus:

```
python3 AMAS.py concat -f phylip -d dna -i *phy --part-format nexus
#NEXUS
Begin sets;
    charset AA = 1-605;
    charset AK = 606-1200;
    charset 28S = 1201-1800;
End;
```

Partitions can also be written by codon positions using the `-n` or `--codons` flag, either for alignments containing first and second or all three positions. In the above example, supplying `-n 123` would result in:

```
AA_pos1 = 1-605\3
AA_pos2 = 2-605\3
AA_pos3 = 3-605\3
AK_pos1 = 606-1200\3
AK_pos2 = 607-1200\3
AK_pos3 = 608-1200\3
28S_pos1 = 1201-1800\3
28S_pos2 = 1202-1800\3
28S_pos3 = 1203-1800\3
```

1.3.1.2 Getting alignment statistics

This is an example of how you can summarize two protein fasta alignments by running:

```
python3 AMAS.py summary -f fasta -d aa -i my_aln.fasta my_aln2.fasta
```

By default `AMAS` will write a file with the summary of the alignment in `summary.txt`. You can change the name of this file with `-o` or `--summary-out`. You can also summarize a single or multiple sequence alignments at once.

The statistics calculated include the number of taxa, alignment length, total number of matrix cells, overall number of undetermined characters, percent of missing data, AT and GC contents (for DNA alignments), number and proportion of variable sites, number and proportion of parsimony informative sites, and counts of all characters present in the relevant alphabet.

1.3.1.3 Converting among formats

To convert all nucleotide fasta files with a `.fas` extension in a directory to nexus alignments, you could use:

```
python3 AMAS.py convert -d dna -f fasta -i *fas -u nexus
```

In the above, the required options are combined with `convert` command to convert the input files and `-u nexus` which indicates the output format.

`AMAS` will not overwrite over input here but will create new files instead, automatically appending appropriate extensions to the input file's name: `-out.fas`, `-out.phy`, `-out.int-phy`, `-out.nex`, or `-out.int-nex`.

1.3.1.4 Splitting alignment by partitions: TODO update

If you have a partition file, you can split a concatenated alignment and write a file for each partition:

```
python3 AMAS.py split -f nexus -d dna -i concat.nex -l partitions.txt -u nexus
```

In the above one input file `concat.nex` was provided for splitting with `split` and partitions file `partitions.txt` with `-l` (same as `--split-by`). For splitting you should only use one input and one partition file at a time. This is an example partition file:

```
AApos1&2  =  1-604\3, 2-605\3
AApos3  =  3-606\3
28SAutapoInDels=7583, 7584, 7587, 7593
```

If this was the `partitions.txt` file from the example command above, `AMAS` would write three output files called `concat_AApos1&2.nex`, `concat_AApos3.nex`, and `concat_28SautapoInDels.nex`. The partitions file will be parsed correctly as long as there is no text prior to the partition name (`CHARSET AApos1&2` or `DNA, AApos1&2` will not work) and commas separate ranges or individual sites in each partition.

Sometimes after splitting you will have alignments with taxa that have only gaps `-` or missing data `?`. If you want to these to not be included in the output , add `-j` or `--remove-empty` to the command line.

1.3.1.5 Translating a DNA alignment into aligned protein sequences

You can translate a nucleotide alignment to amino acids with AMAS using one of the `NCBI translation tables`. For example, to correctly translate an insect mitochondrial gene alignment that begins at a second codon position:

```
python3 AMAS.py translate -f nexus -d dna -i concat.nex --code 5 --reading-frame 2 --out-format phylip
```

`--code` and `--reading-frame` are the same as `-b` and `-k` and are both set to 1 (the standard genetic code and the first character of the alignment corresponds to the first codon position) by default. When translating, `AMAS` will contract gaps `-` and missing `?`, such that `---` becomes `-` in the translated alignment. A warning will be printed if stop codons are found and these are indicated as asterisks `*` in the output. See `AMAS.py translate -h` for more info.

1.3.1.6 Creating replicate data sets

With `AMAS` you can create concatenated alignments from a proportion of randomly chosen alignments that can be used for, for example, a phylogenetic jackknife analysis. Say you have 1000 phylip files, each containing a single aligned locus, and you want to create 200 replicate phylip alignments, each built from 100 loci randomly chosen from all the input files. You can do this by specifying `replicate` command and following it with `-r` or `--rep-aln` followed by the number of replicates (in this case `200`) and number of alignments (`100`). Remember to supply the output format with `-u` if you want it to be other than `fasta`:

```
python3 AMAS.py replicate -r 200 100 -d dna -f phylip -i *phy -u phylip
```

1.3.1.7 Removing taxa/sequences from alignment

It is possible to remove taxa from alignments:
```
python3 AMAS.py remove -x species1 species2 -d dna -f nexus -i *nex -u nexus-int -g no_species12_
```

The above will process all `nexus` files in the directory and remove taxa called `species1` and `species2`. The argument `-x` (the same as `--taxa-to-remove`) is followed by the names of sequences to be removed. Note that `AMAS` converts spaces into underscores and strips any quotes present in input sequence names before processing, so you may need to modify your names to remove accordingly. The argument `-g` (the same as `--out-prefix`) specifies a prefix to be added to output file names. The default prefix is 'reduced_'. You may want to realign your files after taxon removal.

1.3.1.8 Checking if input is aligned

By specifying optional argument `-e` (`--check-align`), you can make `AMAS` check if your input files contain only aligned sequences. This option is disabled by default because it can substantially increase computation times in files with many taxa. Enabling this option also provides an additional check against misspecified input file format.

1.3.1.9 TODO Metapartitions

1.4 TODO AMAS as a Python module

Using `AMAS` inside your Python pipeline gives you much more flexibility in how the input and output are being processed. All the major functions of the command line interface can recreated using `AMAS` as a module. Following installation from `pip` use:
```
pydoc amas.AMAS
```

To access detailed documentation for the classes and functions available.

You can import `AMAS` to your script with:
```
from amas import AMAS
```

The class used to manipulate alignments in `AMAS` is `MetaAlignment`. This class has to be instantiated with the same, named arguments as on the command line: `in_files`, `data_type`, `in_format`. You also need to supply the number of cores to be used with `cores`. MetaAlignment holds one or multiple alignments and its `in_files` option must be a list, even if only one file is being read.
```
meta_aln = AMAS.MetaAlignment(in_files=["gene1.phy"], data_type="dna",in_format="phylip", cores=1)
```

Creating MetaAlignment with multiple files is easy:
```
multi_meta_aln = AMAS.MetaAlignment(in_files=["gene1.phy", "gene1.phy"], data_type="dna",
      in_format="phylip", cores=2)
```

Now you can call the various methods on your alignments. `.get_summaries()` method will compute summaries for your alignments and produce headers for them as atuple with first element being the header and the second element a list of lists with the statistics:
```
summaries = meta_aln.get_summaries()
```

The header is different for nucleotide and amino acid data. You may choose to skip it and print only the second element of the tuple, that is a list of summary statistics:
```
= summaries[1]
```

`.get_parsed_alignments()` returns a list of dictionaries where each dictionary is an alignment and where taxa are the keys and sequences are the values. This allows you to, for example, print only taxa names in each alignment or do other manipulation of the sequence data:
```
# get parsed dictionaties
```

```
aln_dicts = multi_meta_aln.get_parsed_alignments()

# print only taxa names in the alignments:
for alignment in aln_dicts:
    for taxon_name in alignment.keys():
        print(taxon_name)
```

Similar to the above example, it is also easy to get translated amino acid alignment as a list of dictionaries (one per input alignment):

```
# get parsed dictionaties
aln_dicts = multi_meta_aln.get_translated(2, 1) # 2: vertebrate mitochondrial genetic code and 1: reading
      frame starting at first character
```

To split alignment use `.get_partitioned("your_partitions_file")` on a `MetaAlignment` with a single input file. `.get_partitioned()` returns a list of dictionaries of dictionaries, with `{ partition_name : { taxon : sequence } }` structure for each partition:

```
partitions = meta_aln.get_partitioned("partitions.txt")
```

`AMAS` uses `.get_partitions("your_partitions_file")` to parse the partition file:

```
parsed_parts = meta_aln.get_partitions("partitions.txt")
print(parsed_parts)
```

`.get_replicate(no_replicates, no_loci)` gives a list of parsed alignments (dictionaries), each a replicate constructed from the specified number of loci:

```
replicate_sets = multi_meta_aln.get_replicate(2, 2)
```

To concatenate multiple alignments first parse them with `.get_parsed_alignments()`, then pass to `.get_concatenated(your_parsed_alignments)`. This will return a tuple where the first element is the `{ taxon : sequence }` dictionary of concatenated alignment and the second element is the partitions dict with `{ name : range }`.

```
parsed_alns = multi_meta_aln.get_parsed_alignments()
concat_tuple = multi_meta_aln.get_concatenated(parsed_alns)
concatenated_alignments = concat_tuple[0]
concatenated_partitions = concat_tuple[1]
```

Removing taxa from alignments is very easy:

```
spp_to_remove = ["taxon1", "taxon2", "taxon3"]
reduced_alns = multi_meta_aln.remove_taxa(spp_to_remove)
```

To print to file or convert among file formats use one of the `.print_format(parsed_alignment)` methods called with a parsed dictionary as an argument. These methods include `.print_fasta()`, `.print_nexus()`, `.print_nexus_int()`, `print_phylip()`, and `.print_phylip_int()`. They return an apporpriately formatted string.

```
for alignment in concatenated_alignments:
    nex_int_string = meta_aln.print_nexus_int(alignment)
    print(nex_int_string)
```

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

Chapter 6

Namespace Documentation

6.1 amas Namespace Reference

Namespaces

- namespace AMAS

Variables

- str __author__ = 'Marek Borowiec'
- str __email__ = 'petiolus@gmail.com'
- str __version__ = '1.02'
- __all__ = dir()

6.1.1 Variable Documentation

6.1.1.1 __all__

```
amas.__all__ = dir()  [private]
```

Definition at line 6 of file __init__.py.

6.1.1.2 __author__

```
str amas.__author__ = 'Marek Borowiec'  [private]
```

Definition at line 3 of file __init__.py.

6.1.1.3 __email__

```
str amas.__email__ = 'petiolus@gmail.com'  [private]
```

Definition at line 4 of file __init__.py.

6.1.1.4 __version__

```
str amas.__version__ = '1.02'  [private]
```

Definition at line 5 of file __init__.py.

6.2 amas.AMAS Namespace Reference

Classes

- class Alignment
- class AminoAcidAlignment
- class DNAAlignment
- class FileHandler
- class FileParser
- class MetaAlignment
- class ParsedArgs

Functions

- proportion (x)
- main ()
- run ()

6.2.1 Detailed Description

```
This stand-alone program allows manipulations of multiple sequence
alignments. It supports sequential FASTA, PHYLIP, NEXUS, and interleaved PHYLIP
and NEXUS formats for DNA and aino acid sequences. It can print summary statistics,
convert among formats, and concatenate alignments.

Current statistics include the number of taxa, alignment length, total number
of matrix cells, overall number of undetermined characters, percent of missing
data, AT and GC contents (for DNA alignments), number and proportion of
variable sites, number and proportion of parsimony informative sites,
and counts of all characters present in the relevant (nucleotide or amino acid) alphabet.
```

6.2.2 Function Documentation

6.2.2.1 main()

```
amas.AMAS.main ( )
```

Definition at line 2361 of file AMAS.py.

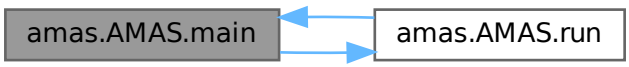
```
02361 def main():
02362
02363     # initialize parsed arguments and meta alignment objects
02364     kwargs = run()
02365     meta_aln = MetaAlignment(**kwargs)
02366
02367     if meta_aln.command == "summary":
02368         meta_aln.write_summaries(kwargs["summary_out"])
02369     if meta_aln.by_taxon_summary:
02370         print("Printing taxon summaries")
02371         meta_aln.write_taxa_summaries()
```

```
02372     if meta_aln.command == "convert":
02373         meta_aln.write_out("convert", kwargs["out_format"])
02374     if meta_aln.command == "concat":
02375         meta_aln.write_out("concat", kwargs["out_format"])
02376         meta_aln.write_partitions(kwargs["concat_part"], kwargs["part_format"], kwargs["data_type"],
      kwargs["codons"])
02377     if meta_aln.command == "replicate":
02378         meta_aln.write_out("replicate", kwargs["out_format"])
02379     if meta_aln.command == "split":
02380         meta_aln.write_out("split", kwargs["out_format"])
02381     if meta_aln.command == "remove":
02382         meta_aln.write_out("remove", kwargs["out_format"])
02383     if meta_aln.command == "translate":
02384         meta_aln.write_out("translate", kwargs["out_format"])
02385     if meta_aln.command == "trim":
02386         meta_aln.write_out("trim", kwargs["out_format"])
02387 
02388     if meta_aln.command == "metapartitions":
02389         # `metapartitions` is essentially `split` + `concat`. Currently you can't set an out_format:
02390         # it's automatically set to match the in_format because the intermediate `split` outputs
      become
02391         # the 'new' in_files for the `concat` operation, and then calling either:
02392         #  -> AminoAcidAlignment(Alignment.__init__(self, in_file, in_format, data_type))
02393         #  -> DNAAlignment(Alignment.__init__(self, in_file, in_format, data_type))
02394         # through MetaAlignment.get_alignment_object(alignment, self.in_format, self.data_type)
02395         meta_aln.write_out("metapartitions", kwargs["in_format"])
02396         meta_aln.write_partitions(kwargs["concat_part"], kwargs["part_format"], kwargs["data_type"],
      "none")
02397 
02398         # meta_aln.write_out("translate", kwargs["out_format"])
02399 
```

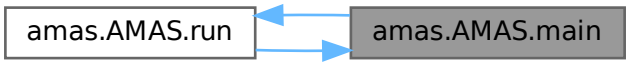
References amas.AMAS.run().

Referenced by amas.AMAS.run().

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.2.2 proportion()

```
amas.AMAS.proportion (
              x )
```

Definition at line 43 of file AMAS.py.

```
00043 def proportion(x):
00044     # needed to prevent input of invalid floats in trim mode
00045     x = float(x)
00046     if x < 0.0 or x > 1.0:
00047         raise argparse.ArgumentTypeError("%r not in range [0.0, 1.0]" % (x,))
00048     return x
00049
```

6.2.2.3 run()

```
amas.AMAS.run ( )
```

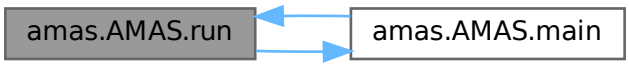
Definition at line 2400 of file AMAS.py.

```
02400 def run():
02401
02402     # initialize parsed arguments
02403     config = ParsedArgs()
02404     # get arguments
02405     config_dict = config.get_args_dict()
02406     return config_dict
02407
```

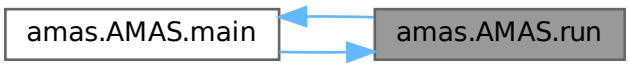
References amas.AMAS.main().

Referenced by amas.AMAS.main().

Here is the call graph for this function:



Here is the caller graph for this function:

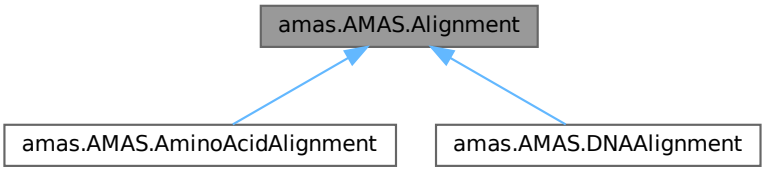


Chapter 7

Class Documentation

7.1 amas.AMAS.Alignment Class Reference

Inheritance diagram for amas.AMAS.Alignment:



Public Member Functions

- __init__ (self, in_file, in_format, data_type)
- __str__ (self)
- get_aln_input (self)
- get_parsed_aln (self)
- summarize_alignment (self)
- summarize_alignment_by_taxa (self)
- get_char_summary (self)
- get_taxon_char_summary (self)
- append_count (self, char_dict)
- matrix_creator (self)
- get_column (self, i)
- all_same (self, site)
- get_sites_no_missing_ambiguous (self)
- get_site_no_missing_ambiguous (self, column)
- replace_missing (self, column)
- get_trim_selection (self, trim_fraction, parsimony_check)
- get_variable (self)
- get_parsimony_informative (self)

- get_prop_variable (self)
- get_prop_parsimony (self)
- get_name (self)
- get_taxa_no (self)
- get_alignment_length (self)
- get_matrix_cells (self)
- get_missing (self)
- get_missing_percent (self)
- get_missing_from_parsed (self)
- get_missing_from_seq (self, seq)
- get_missing_percent_from_seq (self, seq)
- get_counts (self)
- get_counts_from_parsed (self)
- get_counts_from_seq (self, seq)
- check_data_type (self)

Public Attributes

- in_file
- in_format
- data_type
- parsed_aln
- length
- matrix
- no_missing_ambiguous
- variable_sites
- prop_variable
- parsimony_informative
- prop_parsimony
- missing_records
- all_matrix_cells
- missing
- check

7.1.1 Detailed Description

```
Base class: Gets in parsed sequences as input and summarizes their stats.
Based on the data type, the subclasses AminoAcidAlignment & DNAAlignment define the attributes:
`alphabet`, `missing_ambiguous_chars`, `missing_chars`, `non_alphabet`
```

Definition at line 805 of file AMAS.py.

7.1.2 Constructor & Destructor Documentation

7.1.2.1 __init__()

```
amas.AMAS.Alignment.__init__ (
                self,
                in_file,
                in_format,
                data_type )
```

Definition at line 811 of file AMAS.py.

```
00811     def __init__(self, in_file, in_format, data_type):
00812         # initialize alignment class with parsed records and alignment name as arguments,
00813         # create empty lists for list of sequences, sites without
00814         # ambiguous or missing characters, and initialize variable for the number
00815         # of parsimony informative sites
00816         self.in_file = in_file
00817         self.in_format = in_format
00818         self.data_type = data_type
00819
00820         self.parsed_aln = self.get_parsed_aln()
00821
```

7.1.3 Member Function Documentation

7.1.3.1 __str__()

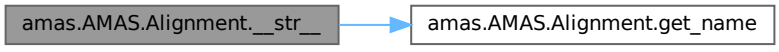
```
amas.AMAS.Alignment.__str__ (
                self )
```

Definition at line 822 of file AMAS.py.

```
00822     def __str__(self):
00823         #  purpose of override? (originally returned method object)
00824         return self.get_name()
00825
```

References amas.AMAS.Alignment.get_name().

Here is the call graph for this function:



7.1.3.2 all_same()

```
amas.AMAS.Alignment.all_same (
                self,
                site )
```

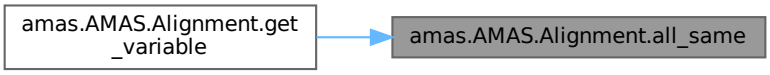
Definition at line 937 of file AMAS.py.

```
00937     def all_same(self, site):
00938         # check if all elements of a site are the same
00939         return not site or site.count(site[0]) == len(site)
```

```
00940
```

Referenced by amas.AMAS.Alignment.get_variable().

Here is the caller graph for this function:



7.1.3.3 append_count()

```
amas.AMAS.Alignment.append_count (
                self,
                char_dict )
```

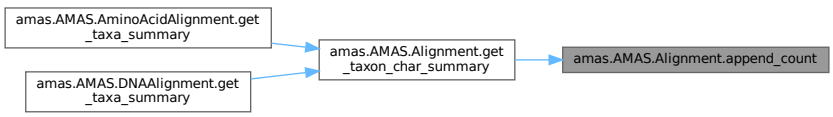
Definition at line 919 of file AMAS.py.

```
00919     def append_count(self, char_dict):
00920         count_list = []
00921         for char in self.alphabet:
00922             if char in char_dict.keys():
00923                 count_list.append(char_dict[char])
00924             else:
00925                 count_list.append(0)
00926         return count_list
00927
```

References amas.AMAS.AminoAcidAlignment.alphabet, and amas.AMAS.DNAAlignment.alphabet.

Referenced by amas.AMAS.Alignment.get_taxon_char_summary().

Here is the caller graph for this function:



7.1.3.4 check_data_type()

```
amas.AMAS.Alignment.check_data_type (
                self )
```

Definition at line 1074 of file AMAS.py.

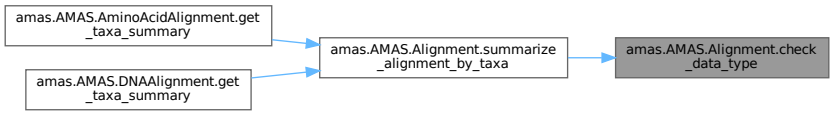
```
01074     def check_data_type(self):
01075         # check if the data type is correct; only one seq to save on computation
01076         seq = next(iter(self.parsed_aln.values()))
01077         self.check = any(char in self.non_alphabet for char in seq)
01078         if self.check is True:
```

```
01079             print(
01080                 "WARNING: found non-" + self.data_type + " characters. "
01081                 "Are you sure you specified the right data type?"
01082             )
01083
01084
```

References amas.AMAS.Alignment.parsed_aln.

Referenced by amas.AMAS.Alignment.summarize_alignment_by_taxa().

Here is the caller graph for this function:



7.1.3.5 get_alignment_length()

```
amas.AMAS.Alignment.get_alignment_length (
                self )
```

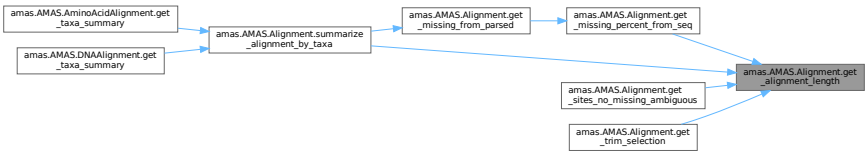
Definition at line 1011 of file AMAS.py.

```
01011     def get_alignment_length(self):
01012         # get alignment length by just checking the first seq length
01013         # this assumes that all sequences are of equal length
01014         return len(next(iter(self.parsed_aln.values())))
01015
```

References amas.AMAS.Alignment.parsed_aln.

Referenced by amas.AMAS.Alignment.get_missing_percent_from_seq(), amas.AMAS.Alignment.get_sites_no_missing_ambiguous(), amas.AMAS.Alignment.get_trim_selection(), and amas.AMAS.Alignment.summarize_alignment_by_taxa().

Here is the caller graph for this function:



7.1.3.6 get_aln_input()

```
amas.AMAS.Alignment.get_aln_input (
              self )
```

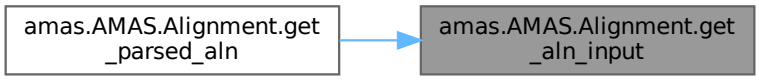
Definition at line 826 of file AMAS.py.

```
00826     def get_aln_input(self):
00827         # open and parse input file
00828         aln_input = FileParser(self.in_file)
00829         return aln_input
00830
```

References amas.AMAS.FileHandler.in_file, amas.AMAS.FileParser.in_file, and amas.AMAS.Alignment.in_file.

Referenced by amas.AMAS.Alignment.get_parsed_aln().

Here is the caller graph for this function:



7.1.3.7 get_char_summary()

```
amas.AMAS.Alignment.get_char_summary (
              self )
```

Definition at line 899 of file AMAS.py.

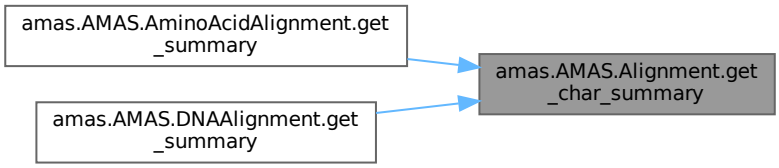
```
00899     def get_char_summary(self):
00900         # get summary of frequencies for all characters
00901         characters = []
00902         counts = []
00903         add_to_chars = characters.append
00904         add_to_counts = counts.append
00905         char_count_dicts = self.get_counts()
00906         for char in self.alphabet:
00907             add_to_chars(char)
00908             if char in char_count_dicts.keys():
00909                 add_to_counts(str(char_count_dicts[char]))
00910             else:
00911                 add_to_counts("0")
00912         return characters, counts
00913
```

References amas.AMAS.AminoAcidAlignment.alphabet, amas.AMAS.DNAAlignment.alphabet, and amas.AMAS.Alignment.get_coun

Referenced by amas.AMAS.AminoAcidAlignment.get_summary(), and amas.AMAS.DNAAlignment.get_summary().

Here is the call graph for this function:

Here is the caller graph for this function:



7.1.3.8 get_column()

```
amas.AMAS.Alignment.get_column (
              self,
              i )
```

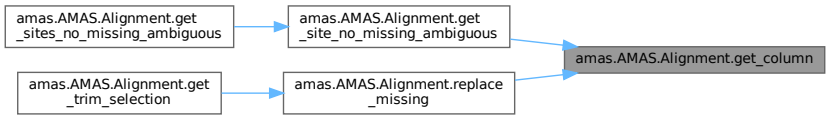
Definition at line 933 of file AMAS.py.

```
00933     def get_column(self, i):
00934         # get site from the character matrix
00935         return [row[i] for row in self.matrix]
00936
```

References amas.AMAS.Alignment.matrix.

Referenced by amas.AMAS.Alignment.get_site_no_missing_ambiguous(), and amas.AMAS.Alignment.replace_missing().

Here is the caller graph for this function:



7.1.3.9 get_counts()

```
amas.AMAS.Alignment.get_counts (
              self )
```

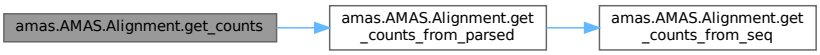
Definition at line 1052 of file AMAS.py.

```
01052     def get_counts(self):
01053         # get counts of each character in the used alphabet for all sequences
01054         counters = [Counter(chars) for taxon, chars in self.get_counts_from_parsed()]
01055         all_counts = sum(counters, Counter())
01056         counts_dict = dict(all_counts)
01057         return counts_dict
01058
```

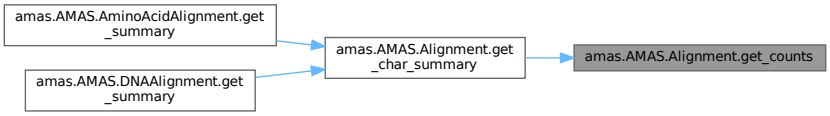
References amas.AMAS.Alignment.get_counts_from_parsed().

Referenced by amas.AMAS.Alignment.get_char_summary().

Here is the call graph for this function:



Here is the caller graph for this function:



7.1.3.10 get_counts_from_parsed()

```
amas.AMAS.Alignment.get_counts_from_parsed (
              self )
```

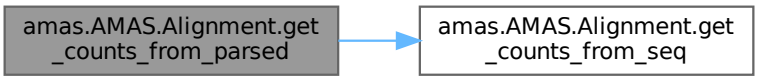
Definition at line 1059 of file AMAS.py.

```
01059     def get_counts_from_parsed(self):
01060         # get counts of all characters from parsed alignment
01061         # return a list of tuples with taxon name and counts
01062         return sorted(
01063             [
01064                 (taxon, self.get_counts_from_seq(seq))
01065                 for taxon, seq in self.parsed_aln.items()
01066             ]
01067         )
01068
```

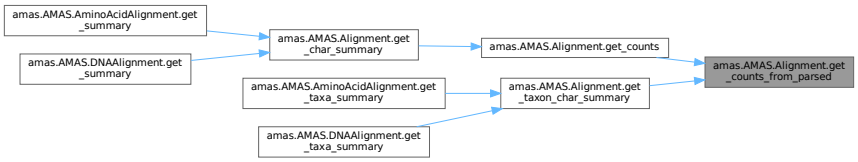
References amas.AMAS.Alignment.get_counts_from_seq(), and amas.AMAS.Alignment.parsed_aln.

Referenced by amas.AMAS.Alignment.get_counts(), and amas.AMAS.Alignment.get_taxon_char_summary().

Here is the call graph for this function:



Here is the caller graph for this function:



7.1.3.11 get_counts_from_seq()

```
amas.AMAS.Alignment.get_counts_from_seq (
                self,
                seq )
```

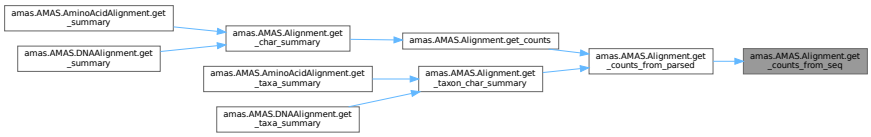
Definition at line 1069 of file AMAS.py.

```
01069     def get_counts_from_seq(self, seq):
01070         # get all alphabet chars count for individual sequence
01071         char_counts = {char : seq.count(char) for char in self.alphabet}
01072         return char_counts
01073
```

References amas.AMAS.AminoAcidAlignment.alphabet, and amas.AMAS.DNAAlignment.alphabet.

Referenced by amas.AMAS.Alignment.get_counts_from_parsed().

Here is the caller graph for this function:



7.1.3.12 get_matrix_cells()

```
amas.AMAS.Alignment.get_matrix_cells (
                self )
```

Definition at line 1016 of file AMAS.py.

```
01016     def get_matrix_cells(self):
01017         # count all matrix cells
01018         self.all_matrix_cells = len(self.parsed_aln.values()) * int(self.length)
01019         return self.all_matrix_cells
01020
```

7.1.3.13 get_missing()

```
amas.AMAS.Alignment.get_missing (
                self )
```

Definition at line 1021 of file AMAS.py.

```
01021     def get_missing(self):
01022         # count missing characters from the list of missing for all sequences
01023         self.missing = sum(count for taxon, count, percent in self.missing_records)
01024         return self.missing
01025
```

7.1.3.14 get_missing_from_parsed()

```
amas.AMAS.Alignment.get_missing_from_parsed (
            self )
```

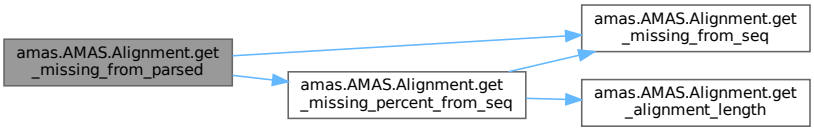
Definition at line 1031 of file AMAS.py.

```
01031     def get_missing_from_parsed(self):
01032         # get missing count and percent from parsed alignment
01033         # return a list of tuples with taxon name, count, and percent missing
01034         self.missing_records = sorted(
01035             [
01036                 (taxon, self.get_missing_from_seq(seq), self.get_missing_percent_from_seq(seq))
01037                 for taxon, seq in self.parsed_aln.items()
01038             ]
01039         )
01040         return self.missing_records
01041
```

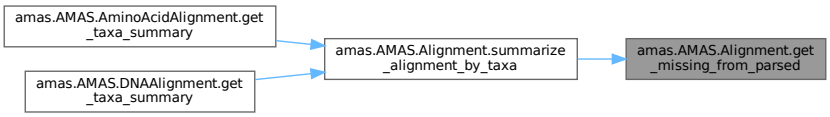
References amas.AMAS.Alignment.get_missing_from_seq(), amas.AMAS.Alignment.get_missing_percent_from_seq(), amas.AMAS.Alignment.missing_records, and amas.AMAS.Alignment.parsed_aln.

Referenced by amas.AMAS.Alignment.summarize_alignment_by_taxa().

Here is the call graph for this function:



Here is the caller graph for this function:



7.1.3.15 get_missing_from_seq()

```
amas.AMAS.Alignment.get_missing_from_seq (
            self,
            seq )
```

Definition at line 1042 of file AMAS.py.

```
01042     def get_missing_from_seq(self, seq):
01043         # count missing characters for individual sequence
01044         missing_count = sum(seq.count(char) for char in self.missing_chars)
01045         return missing_count
01046
```

References amas.AMAS.AminoAcidAlignment.missing_chars, and amas.AMAS.DNAAlignment.missing_chars.

Referenced by amas.AMAS.Alignment.get_missing_from_parsed(), and amas.AMAS.Alignment.get_missing_percent_from_seq().

Here is the caller graph for this function:

7.1.3.16 get_missing_percent()

```
amas.AMAS.Alignment.get_missing_percent (
              self )
```

Definition at line 1026 of file AMAS.py.

```
01026     def get_missing_percent(self):
01027         # get missing percent
01028         missing_percent = round((self.missing / self.all_matrix_cells * 100), 3)
01029         return missing_percent
01030
```

References amas.AMAS.Alignment.all_matrix_cells, and amas.AMAS.Alignment.missing.

7.1.3.17 get_missing_percent_from_seq()

```
amas.AMAS.Alignment.get_missing_percent_from_seq (
              self,
              seq )
```

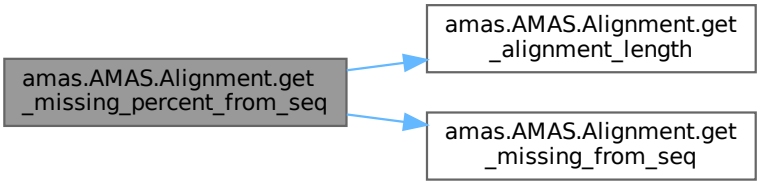
Definition at line 1047 of file AMAS.py.

```
01047     def get_missing_percent_from_seq(self, seq):
01048         # get missing percent from individual sequence
01049         missing_seq_percent = round((self.get_missing_from_seq(seq) / self.get_alignment_length() *
      100), 3)
01050         return missing_seq_percent
01051
```

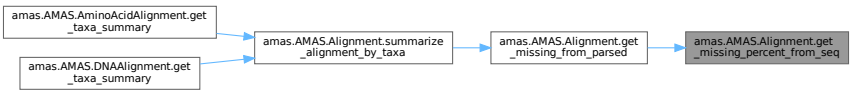
References amas.AMAS.Alignment.get_alignment_length(), and amas.AMAS.Alignment.get_missing_from_seq().

Referenced by amas.AMAS.Alignment.get_missing_from_parsed().

Here is the call graph for this function:



Here is the caller graph for this function:



7.1.3.18 get_name()

```
amas.AMAS.Alignment.get_name (
              self )
```

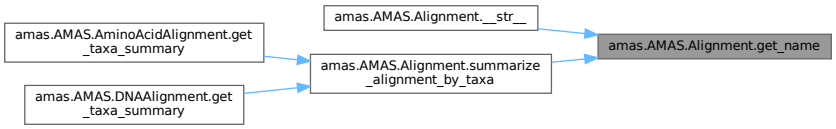
Definition at line 1002 of file AMAS.py.

```
01002     def get_name(self):
01003         # get input file name
01004         in_filename = path.basename(self.in_file)
01005         return in_filename
01006
```

References amas.AMAS.FileHandler.in_file, amas.AMAS.FileParser.in_file, and amas.AMAS.Alignment.in_file.

Referenced by amas.AMAS.Alignment.__str__(), and amas.AMAS.Alignment.summarize_alignment_by_taxa().

Here is the caller graph for this function:



7.1.3.19 get_parsed_aln()

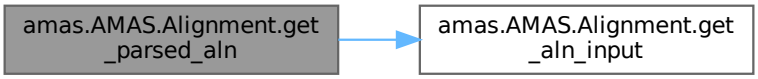
```
amas.AMAS.Alignment.get_parsed_aln (
              self )
```

Definition at line 831 of file AMAS.py.

```
00831     def get_parsed_aln(self):
00832         # parse according to the given format
00833         aln_input = self.get_aln_input()
00834         if self.in_format == "fasta":
00835             parsed_aln = aln_input.fasta_parse()
00836         elif self.in_format == "phylip":
00837             parsed_aln = aln_input.phylip_parse()
00838         elif self.in_format == "phylip-int":
00839             parsed_aln = aln_input.phylip_interleaved_parse()
00840         elif self.in_format == "nexus":
00841             parsed_aln = aln_input.nexus_parse()
00842         elif self.in_format == "nexus-int":
00843             parsed_aln = aln_input.nexus_interleaved_parse()
00844
00845         return parsed_aln
00846
```

References amas.AMAS.Alignment.get_aln_input(), amas.AMAS.Alignment.in_format, and amas.AMAS.MetaAlignment.in_format.

Here is the call graph for this function:



7.1.3.20 get_parsimony_informative()

```
amas.AMAS.Alignment.get_parsimony_informative (
              self )
```

Definition at line 978 of file AMAS.py.

```
00978     def get_parsimony_informative(self):
00979         # if the count for a unique character in a site is at least two,
00980         # and there are at least two such characters in a site without missing
00981         # or ambiguous characters, consider it parsimony informative
00982         parsimony_informative = 0
00983         for site in self.no_missing_ambiguous:
00984             unique_chars = set(site)
00985             pattern = [base for base in unique_chars if site.count(base) >= 2]
00986             no_patterns = len(pattern)
00987 
00988             if no_patterns >= 2:
00989                 parsimony_informative += 1
00990         return parsimony_informative
00991 
```

References amas.AMAS.Alignment.no_missing_ambiguous.

7.1.3.21 get_prop_parsimony()

```
amas.AMAS.Alignment.get_prop_parsimony (
              self )
```

Definition at line 997 of file AMAS.py.

```
00997     def get_prop_parsimony(self):
00998         # get proportion of parsimony informative sites to all sites
00999         prop_parsimony = self.parsimony_informative / int(self.length)
01000         return round(prop_parsimony, 3)
01001 
```

References amas.AMAS.Alignment.length, and amas.AMAS.Alignment.parsimony_informative.

7.1.3.22 get_prop_variable()

```
amas.AMAS.Alignment.get_prop_variable (
              self )
```

Definition at line 992 of file AMAS.py.

```
00992     def get_prop_variable(self):
00993         # get proportion of variable sites to all sites
00994         prop_variable = self.variable_sites / int(self.length)
00995         return round(prop_variable, 3)
00996 
```

References amas.AMAS.Alignment.length, and amas.AMAS.Alignment.variable_sites.

7.1.3.23 get_site_no_missing_ambiguous()

```
amas.AMAS.Alignment.get_site_no_missing_ambiguous (
               self,
               column )
```

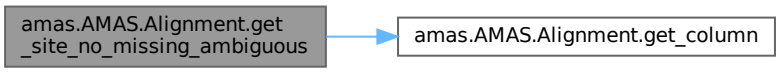
Definition at line 946 of file AMAS.py.

```
00946     def get_site_no_missing_ambiguous(self, column):
00947         site = self.get_column(column)
00948         return [char for char in site if char not in self.missing_ambiguous_chars]
00949
```

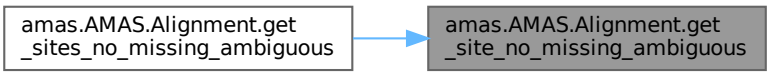
References amas.AMAS.Alignment.get_column(), amas.AMAS.AminoAcidAlignment.missing_ambiguous_chars, and amas.AMAS.DNAAlignment.missing_ambiguous_chars.

Referenced by amas.AMAS.Alignment.get_sites_no_missing_ambiguous().

Here is the call graph for this function:



Here is the caller graph for this function:



7.1.3.24 get_sites_no_missing_ambiguous()

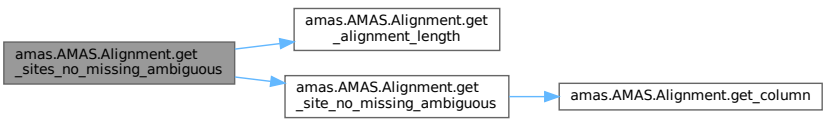
```
amas.AMAS.Alignment.get_sites_no_missing_ambiguous (
               self )
```

Definition at line 941 of file AMAS.py.

```
00941     def get_sites_no_missing_ambiguous(self):
00942         # get each site without missing or ambiguous characters
00943         no_missing_ambiguous_sites = [self.get_site_no_missing_ambiguous(column) for column in
      range(self.get_alignment_length())]
00944         return no_missing_ambiguous_sites
00945
```

References amas.AMAS.Alignment.get_alignment_length(), and amas.AMAS.Alignment.get_site_no_missing_ambiguous().

Here is the call graph for this function:



7.1.3.25 get_taxa_no()

```
amas.AMAS.Alignment.get_taxa_no (
              self )
```

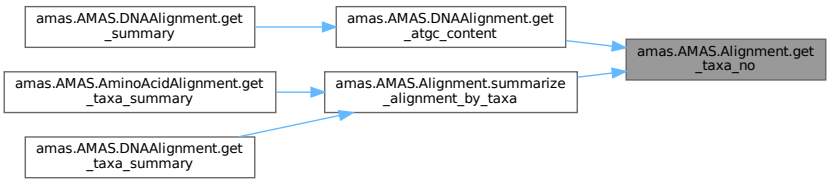
Definition at line 1007 of file AMAS.py.

```
01007     def get_taxa_no(self):
01008         # get number of taxa
01009         return len(self.parsed_aln.values())
01010
```

References amas.AMAS.Alignment.parsed_aln.

Referenced by amas.AMAS.DNAAlignment.get_atgc_content(), and amas.AMAS.Alignment.summarize_alignment_by_taxa().

Here is the caller graph for this function:



7.1.3.26 get_taxon_char_summary()

```
amas.AMAS.Alignment.get_taxon_char_summary (
              self )
```

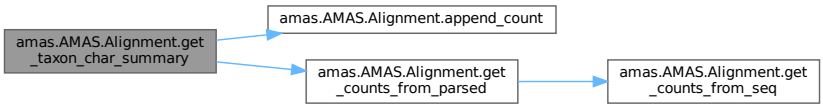
Definition at line 914 of file AMAS.py.

```
00914     def get_taxon_char_summary(self):
00915         # get summary of frequencies for all characters
00916         records = (self.append_count(char_dict) for taxon, char_dict in self.get_counts_from_parsed())
00917         return records
00918
```

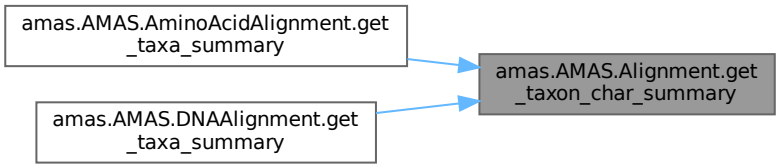
References amas.AMAS.Alignment.append_count(), and amas.AMAS.Alignment.get_counts_from_parsed().

Referenced by amas.AMAS.AminoAcidAlignment.get_taxa_summary(), and amas.AMAS.DNAAlignment.get_taxa_summary().

Here is the call graph for this function:



Here is the caller graph for this function:



7.1.3.27 get_trim_selection()

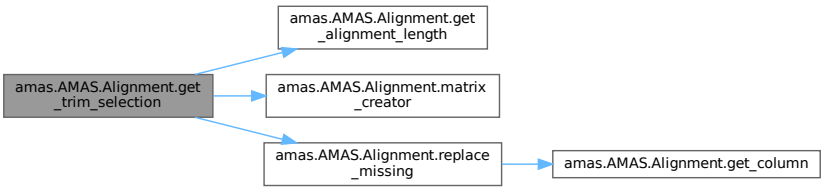
```
amas.AMAS.Alignment.get_trim_selection (
              self,
              trim_fraction,
              parsimony_check )
```

Definition at line 953 of file AMAS.py.

```
00953     def get_trim_selection(self, trim_fraction, parsimony_check):
00954         # this checks each column of alignment for minimum occupancy
00955         self.matrix = self.matrix_creator()
00956         trim_vector = []
00957         for column in range(self.get_alignment_length()):
00958             site = self.replace_missing(column)
00959             occ = (len(site) - site.count("-")) / len(site)
00960             if parsimony_check:
00961                 unique_chars = set(site)
00962                 try:
00963                     unique_chars.remove("-")
00964                 except KeyError:
00965                     pass # this occurs if we have no missing data
00966                 pattern = [base for base in unique_chars if site.count(base) >= 2]
00967                 trim_vector.append(len(pattern) >= 2 and occ >= trim_fraction)
00968             else:
00969                 trim_vector.append(occ >= trim_fraction)
00970         return trim_vector
00971
```

References amas.AMAS.Alignment.get_alignment_length(), amas.AMAS.Alignment.matrix, amas.AMAS.Alignment.matrix_creator(), and amas.AMAS.Alignment.replace_missing().

Here is the call graph for this function:



7.1.3.28 get_variable()

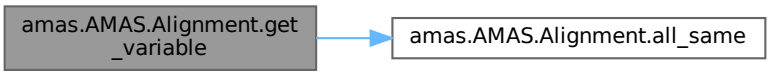
```
amas.AMAS.Alignment.get_variable (
              self )
```

Definition at line 972 of file AMAS.py.

```
00972     def get_variable(self):
00973         # if all elements of a site without missing or ambiguous characters
00974         # are not the same, consider it variable
00975         variable = len([site for site in self.no_missing_ambiguous if not self.all_same(site)])
00976         return variable
00977
```

References amas.AMAS.Alignment.all_same(), and amas.AMAS.Alignment.no_missing_ambiguous.

Here is the call graph for this function:



7.1.3.29 matrix_creator()

```
amas.AMAS.Alignment.matrix_creator (
              self )
```

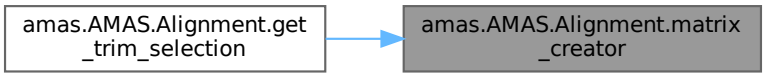
Definition at line 928 of file AMAS.py.

```
00928     def matrix_creator(self):
00929         # decompose character matrix into a two-dimensional list
00930         matrix = [list(sequence) for sequence in self.parsed_aln.values()]
00931         return matrix
00932
```

References amas.AMAS.Alignment.parsed_aln.

Referenced by amas.AMAS.Alignment.get_trim_selection().

Here is the caller graph for this function:



7.1.3.30 replace_missing()

```
amas.AMAS.Alignment.replace_missing (
                self,
                column )
```

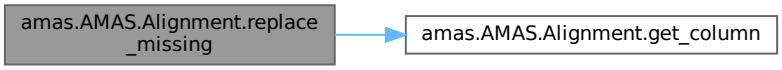
Definition at line 950 of file AMAS.py.

```
00950     def replace_missing(self, column):
00951         return ["-" if x in self.missing_chars else x for x in self.get_column(column)]
00952
```

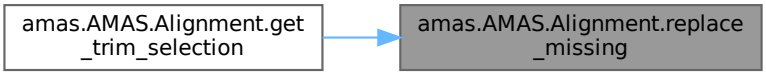
References amas.AMAS.Alignment.get_column(), amas.AMAS.AminoAcidAlignment.missing_chars, and amas.AMAS.DNAAlignment.missing_chars.

Referenced by amas.AMAS.Alignment.get_trim_selection().

Here is the call graph for this function:



Here is the caller graph for this function:



7.1.3.31 summarize_alignment()

```
amas.AMAS.Alignment.summarize_alignment (
                self )
```

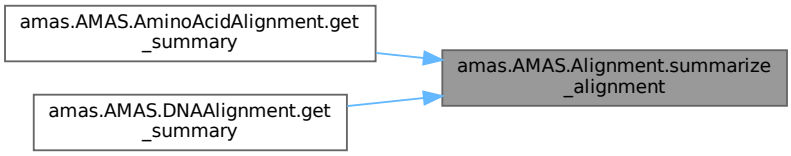
Definition at line 847 of file AMAS.py.

```
00847     def summarize_alignment(self):
00848         # call methods to create sequences list, matrix, sites without ambiguous or
00849         # missing characters; get and summarize alignment statistics
00850         summary = []
00851         self.length = str(self.get_alignment_length())
00852         self.matrix = self.matrix_creator()
00853         self.no_missing_ambiguous = self.get_sites_no_missing_ambiguous()
00854         self.variable_sites = self.get_variable()
00855         self.prop_variable = self.get_prop_variable()
00856         self.parsimony_informative = self.get_parsimony_informative()
00857         self.prop_parsimony = self.get_prop_parsimony()
00858         self.missing_records = self.get_missing_from_parsed()
00859         name = str(self.get_name())
00860         taxa_no = str(self.get_taxa_no())
00861         cells = str(self.get_matrix_cells())
00862         missing = str(self.get_missing())
00863         missing_percent = str(self.get_missing_percent())
```

```
00864         self.check_data_type()
00865         summary = [
00866             name,
00867             taxa_no,
00868             self.length,
00869             cells,
00870             missing,
00871             missing_percent,
00872             str(self.variable_sites),
00873             str(self.prop_variable),
00874             str(self.parsimony_informative),
00875             str(self.prop_parsimony)
00876         ]
00877         return summary
00878
```

Referenced by amas.AMAS.AminoAcidAlignment.get_summary(), and amas.AMAS.DNAAlignment.get_summary().

Here is the caller graph for this function:



7.1.3.32 summarize_alignment_by_taxa()

```
amas.AMAS.Alignment.summarize_alignment_by_taxa (
               self )
```

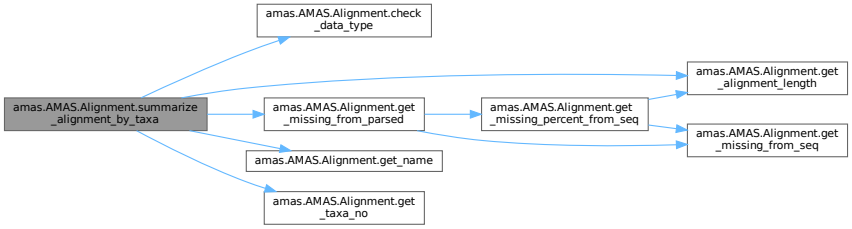
Definition at line 879 of file AMAS.py.

```
00879     def summarize_alignment_by_taxa(self):
00880         # get summary for all taxa/sequences in alignment
00881         per_taxon_summary = []
00882         taxa_no = self.get_taxa_no()
00883         self.missing_records = self.get_missing_from_parsed()
00884         self.length = self.get_alignment_length()
00885         lengths = (self.length for i in range(taxa_no))
00886         name = self.get_name()
00887         names = (name for i in range(taxa_no))
00888         taxa_names = (
00889             taxon.replace(" ", "_").replace(".", "_").replace("'", "")
00890             for taxon, missing_count, missing_percent in self.missing_records
00891         )
00892         missing = (missing_count for taxon, missing_count, missing_percent in self.missing_records)
00893         missing_percent = (missing_percent for taxon, missing_count, missing_percent in
      self.missing_records)
00894         self.check_data_type()
00895         per_taxon_summary = (names, taxa_names, lengths, missing, missing_percent)
00896         zipped = list(zip(*per_taxon_summary))
00897         return zipped
00898
```

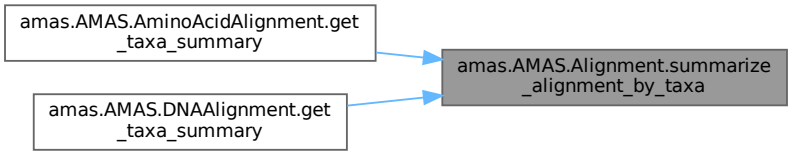
References amas.AMAS.Alignment.check_data_type(), amas.AMAS.Alignment.get_alignment_length(), amas.AMAS.Alignment.get_n
amas.AMAS.Alignment.get_name(), amas.AMAS.Alignment.get_taxa_no(), amas.AMAS.Alignment.length, and
amas.AMAS.Alignment.missing_records.

Referenced by amas.AMAS.AminoAcidAlignment.get_taxa_summary(), and amas.AMAS.DNAAlignment.get_taxa_summary().

Here is the call graph for this function:



Here is the caller graph for this function:



7.1.4 Member Data Documentation

7.1.4.1 all_matrix_cells

`amas.AMAS.Alignment.all_matrix_cells`

Definition at line 1018 of file AMAS.py.

Referenced by amas.AMAS.Alignment.get_missing_percent().

7.1.4.2 check

`amas.AMAS.Alignment.check`

Definition at line 1077 of file AMAS.py.

7.1.4.3 data_type

`amas.AMAS.Alignment.data_type`

Definition at line 818 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_alignment_object(), amas.AMAS.MetaAlignment.get_summaries(), amas.AMAS.MetaAlignment.get_taxon_summaries(), amas.AMAS.MetaAlignment.print_nexus(), amas.AMAS.MetaAlignment.print_n and amas.AMAS.MetaAlignment.write_out().

7.1.4.4 in_file

amas.AMAS.Alignment.in_file

Definition at line 816 of file AMAS.py.

Referenced by amas.AMAS.FileHandler.__exit__(), amas.AMAS.Alignment.get_aln_input(), and amas.AMAS.Alignment.get_name().

7.1.4.5 in_format

amas.AMAS.Alignment.in_format

Definition at line 817 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_alignment_object(), amas.AMAS.Alignment.get_parsed_aln(), and amas.AMAS.MetaAlignment.write_out().

7.1.4.6 length

amas.AMAS.Alignment.length

Definition at line 851 of file AMAS.py.

Referenced by amas.AMAS.Alignment.get_prop_parsimony(), amas.AMAS.Alignment.get_prop_variable(), and amas.AMAS.Alignment.summarize_alignment_by_taxa().

7.1.4.7 matrix

amas.AMAS.Alignment.matrix

Definition at line 852 of file AMAS.py.

Referenced by amas.AMAS.Alignment.get_column(), and amas.AMAS.Alignment.get_trim_selection().

7.1.4.8 missing

amas.AMAS.Alignment.missing

Definition at line 1023 of file AMAS.py.

Referenced by amas.AMAS.Alignment.get_missing_percent().

7.1.4.9 missing_records

amas.AMAS.Alignment.missing_records

Definition at line 858 of file AMAS.py.

Referenced by amas.AMAS.Alignment.get_missing_from_parsed(), and amas.AMAS.Alignment.summarize_alignment_by_taxa().

7.1.4.10 no_missing_ambiguous

`amas.AMAS.Alignment.no_missing_ambiguous`

Definition at line 853 of file AMAS.py.

Referenced by amas.AMAS.Alignment.get_parsimony_informative(), and amas.AMAS.Alignment.get_variable().

7.1.4.11 parsed_aln

`amas.AMAS.Alignment.parsed_aln`

Definition at line 820 of file AMAS.py.

Referenced by amas.AMAS.Alignment.check_data_type(), amas.AMAS.Alignment.get_alignment_length(), amas.AMAS.DNAAlignment.get_atgc_from_parsed(), amas.AMAS.Alignment.get_counts_from_parsed(), amas.AMAS.Alignment.get amas.AMAS.Alignment.get_taxa_no(), and amas.AMAS.Alignment.matrix_creator().

7.1.4.12 parsimony_informative

`amas.AMAS.Alignment.parsimony_informative`

Definition at line 856 of file AMAS.py.

Referenced by amas.AMAS.Alignment.get_prop_parsimony().

7.1.4.13 prop_parsimony

`amas.AMAS.Alignment.prop_parsimony`

Definition at line 857 of file AMAS.py.

7.1.4.14 prop_variable

`amas.AMAS.Alignment.prop_variable`

Definition at line 855 of file AMAS.py.

7.1.4.15 variable_sites

`amas.AMAS.Alignment.variable_sites`

Definition at line 854 of file AMAS.py.

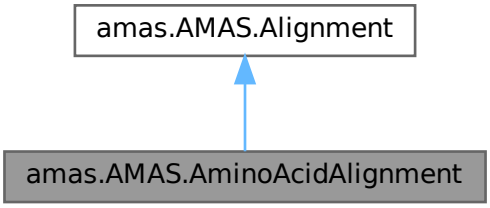
Referenced by amas.AMAS.Alignment.get_prop_variable().

The documentation for this class was generated from the following file:

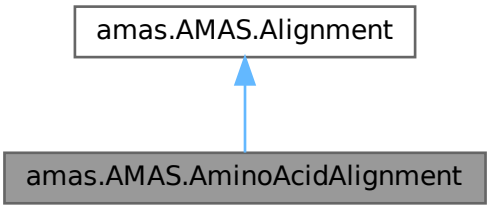
- amas/AMAS.py

7.2 amas.AMAS.AminoAcidAlignment Class Reference

Inheritance diagram for amas.AMAS.AminoAcidAlignment:



Collaboration diagram for amas.AMAS.AminoAcidAlignment:



Public Member Functions

- get_summary (self)
- get_taxa_summary (self)

Static Public Attributes

- list alphabet = ["A", "C", "D", "E", "F", "G", "H", "I", "K", "L", "M", "N", "P", "Q", "R", "S", "T", "V", "W", "Y", "B", "J", "Z", "X", ".", "*", "-", "?"]
- list missing_ambiguous_chars = ["B", "J", "Z", "X", ".", "*", "-", "?"]
- list missing_chars = ["X", ".", "*", "-", "?"]
- list non_alphabet = ["O"]

7.2.1 Detailed Description

```
Alphabets specific to amino acid alignments
```

Definition at line 1085 of file AMAS.py.

7.2.2 Member Function Documentation

7.2.2.1 get_summary()

```
amas.AMAS.AminoAcidAlignment.get_summary (
              self )
```

Definition at line 1093 of file AMAS.py.

```
01093     def get_summary(self):
01094         # get alignment summary specific to amino acids
01095         data = self.summarize_alignment()
01096         new_data = data + list(self.get_char_summary()[1])
01097         return new_data
01098
```

References amas.AMAS.Alignment.get_char_summary(), and amas.AMAS.Alignment.summarize_alignment().

Here is the call graph for this function:

7.2.2.2 get_taxa_summary()

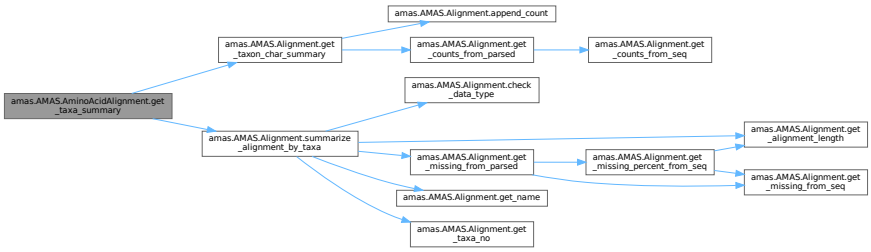
```
amas.AMAS.AminoAcidAlignment.get_taxa_summary (
              self )
```

Definition at line 1099 of file AMAS.py.

```
01099     def get_taxa_summary(self):
01100         # get per-taxon/sequence alignment summary specific to amino acids
01101         data = self.summarize_alignment_by_taxa()
01102         aa_summary = (data, self.get_taxon_char_summary())
01103         zipped_list = list(zip(*aa_summary))
01104         new_data = [list(data_tupl) + chars for data_tupl, chars in zipped_list]
01105         return new_data
01106
```

References amas.AMAS.Alignment.get_taxon_char_summary(), and amas.AMAS.Alignment.summarize_alignment_by_taxa().

Here is the call graph for this function:



7.2.3 Member Data Documentation

7.2.3.1 alphabet

```
list amas.AMAS.AminoAcidAlignment.alphabet = ["A", "C", "D", "E", "F", "G", "H", "I", "K",
"L", "M", "N", "P", "Q", "R", "S", "T", "V", "W", "Y", "B", "J", "Z", "X", ".", "*", "-", "?"]
[static]
```

Definition at line 1088 of file AMAS.py.

Referenced by amas.AMAS.Alignment.append_count(), amas.AMAS.Alignment.get_char_summary(), and amas.AMAS.Alignment.get_counts_from_seq().

7.2.3.2 missing_ambiguous_chars

```
list amas.AMAS.AminoAcidAlignment.missing_ambiguous_chars = ["B", "J", "Z", "X", ".", "*",
"-", "?"]  [static]
```

Definition at line 1089 of file AMAS.py.

Referenced by amas.AMAS.Alignment.get_site_no_missing_ambiguous().

7.2.3.3 missing_chars

```
list amas.AMAS.AminoAcidAlignment.missing_chars = ["X", ".", "*", "-", "?"]  [static]
```

Definition at line 1090 of file AMAS.py.

Referenced by amas.AMAS.Alignment.get_missing_from_seq(), and amas.AMAS.Alignment.replace_missing().

7.2.3.4 non_alphabet

```
list amas.AMAS.AminoAcidAlignment.non_alphabet = ["O"]  [static]
```

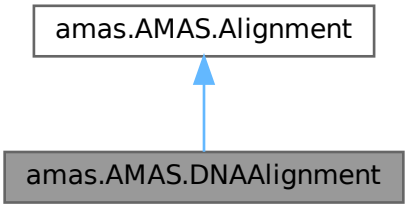
Definition at line 1091 of file AMAS.py.

The documentation for this class was generated from the following file:

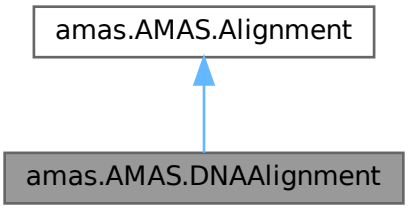
- amas/AMAS.py

7.3 amas.AMAS.DNAAlignment Class Reference

Inheritance diagram for amas.AMAS.DNAAlignment:



Collaboration diagram for amas.AMAS.DNAAlignment:



Public Member Functions

- get_summary (self)
- get_taxa_summary (self)
- get_atgc_content (self)
- get_list_from_atgc (self)
- get_atgc_from_parsed (self)
- get_atgc_from_seq (self, seq)

Static Public Attributes

- list alphabet = ["A", "C", "G", "T", "K", "M", "R", "Y", "S", "W", "B", "V", "H", "D", "X", "N", "O", "-", "?"]
- list missing_ambiguous_chars = ["K", "M", "R", "Y", "S", "W", "B", "V", "H", "D", "X", "N", "O", "-", "?"]
- list missing_chars = ["X", "N", "O", "-", "?"]
- list non_alphabet = ["E", "F", "I", "L", "P", "Q", "J", "Z", ".", "*"]

7.3.1 Detailed Description

```
Alphabets specific to DNA alignments
```

Definition at line 1107 of file AMAS.py.

7.3.2 Member Function Documentation

7.3.2.1 get_atgc_content()

```
amas.AMAS.DNAAlignment.get_atgc_content (
              self )
```

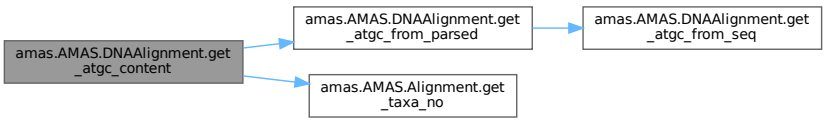
Definition at line 1129 of file AMAS.py.

```
01129     def get_atgc_content(self):
01130         # get AC and GC contents for all sequences
01131         # AT content is the first element of AT, GC content tuple
01132         # returned by get_atgc_from_seq()
01133         atgc_records = self.get_atgc_from_parsed()
01134         at_content = round(sum(atgc[0] for taxon, atgc in atgc_records) / self.get_taxa_no(), 3)
01135         gc_content = round(1 - float(at_content), 3)
01136 
01137         atgc_content = [str(at_content), str(gc_content)]
01138         return atgc_content
01139 
```

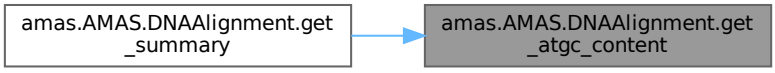
References amas.AMAS.DNAAlignment.get_atgc_from_parsed(), and amas.AMAS.Alignment.get_taxa_no().

Referenced by amas.AMAS.DNAAlignment.get_summary().

Here is the call graph for this function:



Here is the caller graph for this function:



7.3.2.2 get_atgc_from_parsed()

```
amas.AMAS.DNAAlignment.get_atgc_from_parsed (
                self )
```

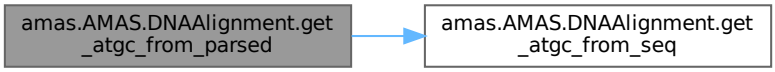
Definition at line 1144 of file AMAS.py.

```
01144     def get_atgc_from_parsed(self):
01145         # get AT and GC contents from parsed alignment dictionary
01146         # return a list of tuples with taxon name, AT content, and GC content
01147         return sorted([(taxon, self.get_atgc_from_seq(seq)) for taxon, seq in
      self.parsed_aln.items()])
01148
```

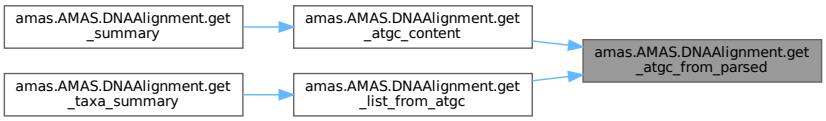
References amas.AMAS.DNAAlignment.get_atgc_from_seq(), and amas.AMAS.Alignment.parsed_aln.

Referenced by amas.AMAS.DNAAlignment.get_atgc_content(), and amas.AMAS.DNAAlignment.get_list_from_atgc().

Here is the call graph for this function:



Here is the caller graph for this function:



7.3.2.3 get_atgc_from_seq()

```
amas.AMAS.DNAAlignment.get_atgc_from_seq (
                self,
                seq )
```

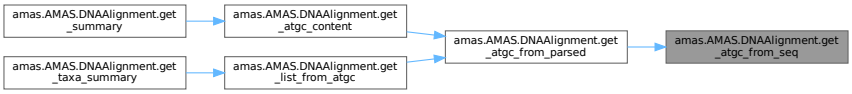
Definition at line 1149 of file AMAS.py.

```
01149     def get_atgc_from_seq(self, seq):
01150         # get AT and GC contents from individual sequences
01151
01152         at_count = seq.count("A") + seq.count("T") + seq.count("W")
01153         gc_count = seq.count("G") + seq.count("C") + seq.count("S")
01154
01155         try:
01156             at_content = round(at_count / (at_count + gc_count), 3)
01157             gc_content = round(1 - float(at_content), 3)
01158
01159         except ZeroDivisionError:
01160             at_content = 0
01161             gc_content = 0
01162
01163         return at_content, gc_content
```

```
01164
```

Referenced by amas.AMAS.DNAAlignment.get_atgc_from_parsed().

Here is the caller graph for this function:



7.3.2.4 get_list_from_atgc()

```
amas.AMAS.DNAAlignment.get_list_from_atgc (
              self )
```

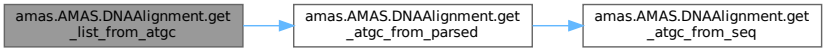
Definition at line 1140 of file AMAS.py.

```
01140     def get_list_from_atgc(self):
01141         records = (atgc for taxon, atgc in self.get_atgc_from_parsed())
01142         return records
01143
```

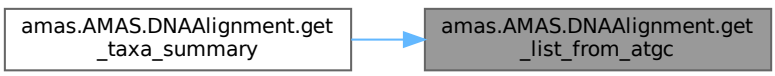
References amas.AMAS.DNAAlignment.get_atgc_from_parsed().

Referenced by amas.AMAS.DNAAlignment.get_taxa_summary().

Here is the call graph for this function:



Here is the caller graph for this function:



7.3.2.5 get_summary()

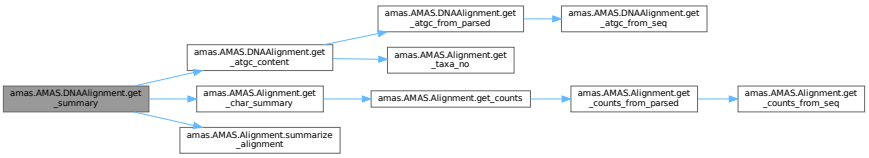
```
amas.AMAS.DNAAlignment.get_summary (
              self )
```

Definition at line 1115 of file AMAS.py.

```
01115     def get_summary(self):
01116         # get alignment summarry specific to nucleotide
01117         data = self.summarize_alignment()
01118         new_data = data + self.get_atgc_content() + list(self.get_char_summary()[1])
01119         return new_data
01120
```

References amas.AMAS.DNAAlignment.get_atgc_content(), amas.AMAS.Alignment.get_char_summary(), and amas.AMAS.Alignment.summarize_alignment().

Here is the call graph for this function:



7.3.2.6 get_taxa_summary()

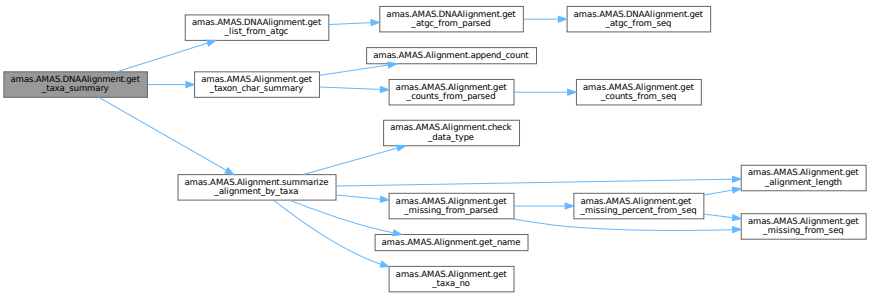
```
amas.AMAS.DNAAlignment.get_taxa_summary (
              self )
```

Definition at line 1121 of file AMAS.py.

```
01121     def get_taxa_summary(self):
01122         # get per-taxon/sequence alignment summary specific to nucleotides
01123         data = self.summarize_alignment_by_taxa()
01124         dna_summary = (data, self.get_list_from_atgc(), self.get_taxon_char_summary())
01125         zipped_list = list(zip(*dna_summary))
01126         new_data = [list(data_tupl) + list(atgc) + chars for data_tupl, atgc, chars in zipped_list]
01127         return new_data
01128
```

References amas.AMAS.DNAAlignment.get_list_from_atgc(), amas.AMAS.Alignment.get_taxon_char_summary(), and amas.AMAS.Alignment.summarize_alignment_by_taxa().

Here is the call graph for this function:



7.3.3 Member Data Documentation

7.3.3.1 alphabet

```
list amas.AMAS.DNAAlignment.alphabet = ["A", "C", "G", "T", "K", "M", "R", "Y", "S", "W", "B",
"V", "H", "D", "X", "N", "O", "-", "?"]  [static]
```

Definition at line 1110 of file AMAS.py.

Referenced by amas.AMAS.Alignment.append_count(), amas.AMAS.Alignment.get_char_summary(), and amas.AMAS.Alignment.get_counts_from_seq().

7.3.3.2 missing_ambiguous_chars

```
list amas.AMAS.DNAAlignment.missing_ambiguous_chars = ["K", "M", "R", "Y", "S", "W", "B", "V",
"H", "D", "X", "N", "O", "-", "?"]  [static]
```

Definition at line 1111 of file AMAS.py.

Referenced by amas.AMAS.Alignment.get_site_no_missing_ambiguous().

7.3.3.3 missing_chars

```
list amas.AMAS.DNAAlignment.missing_chars = ["X", "N", "O", "-", "?"]  [static]
```

Definition at line 1112 of file AMAS.py.

Referenced by amas.AMAS.Alignment.get_missing_from_seq(), and amas.AMAS.Alignment.replace_missing().

7.3.3.4 non_alphabet

```
list amas.AMAS.DNAAlignment.non_alphabet = ["E", "F", "I", "L", "P", "Q", "J", "Z", ".", "*"]
[static]
```

Definition at line 1113 of file AMAS.py.

The documentation for this class was generated from the following file:

- amas/AMAS.py

7.4 amas.AMAS.FileHandler Class Reference

Public Member Functions

- __init__ (self, file_name)
- __enter__ (self)
- __exit__ (self, *args)
- get_file_name (self)

Public Attributes

- file_name
- in_file

7.4.1 Detailed Description

```
Define file handle that closes when out of scope
```

Definition at line 518 of file AMAS.py.

7.4.2 Constructor & Destructor Documentation

7.4.2.1 __init__()

```
amas.AMAS.FileHandler.__init__ (
              self,
              file_name )
```

Definition at line 521 of file AMAS.py.

```
00521     def __init__(self, file_name):
00522         self.file_name = file_name
00523
```

7.4.3 Member Function Documentation

7.4.3.1 __enter__()

```
amas.AMAS.FileHandler.__enter__ (
              self )
```

Definition at line 524 of file AMAS.py.

```
00524     def __enter__(self):
00525         try:
00526             self.in_file = open(self.file_name, "r", encoding="utf-8")
00527         except FileNotFoundError:
00528             print("ERROR: File '" + self.file_name + "' not found.")
00529             sys.exit()
00530         return self.in_file
00531
```

7.4.3.2 __exit__()

```
amas.AMAS.FileHandler.__exit__ (
              self,
             * args )
```

Definition at line 532 of file AMAS.py.

```
00532     def __exit__(self, *args):
00533         self.in_file.close()
00534
```

References amas.AMAS.FileHandler.in_file, amas.AMAS.FileParser.in_file, and amas.AMAS.Alignment.in_file.

7.4.3.3 get_file_name()

```
amas.AMAS.FileHandler.get_file_name (
               self )
```

Definition at line 535 of file AMAS.py.

```
00535     def get_file_name(self):
00536         return self.file_name
00537
```

References amas.AMAS.FileHandler.file_name.

7.4.4 Member Data Documentation

7.4.4.1 file_name

```
amas.AMAS.FileHandler.file_name
```

Definition at line 522 of file AMAS.py.

Referenced by amas.AMAS.FileHandler.get_file_name().

7.4.4.2 in_file

```
amas.AMAS.FileHandler.in_file
```

Definition at line 526 of file AMAS.py.

Referenced by amas.AMAS.FileHandler.__exit__(), amas.AMAS.Alignment.get_aln_input(), and amas.AMAS.Alignment.get_name().

The documentation for this class was generated from the following file:

- amas/AMAS.py

7.5 amas.AMAS.FileParser Class Reference

Public Member Functions

- __init__ (self, in_file)
- fasta_parse (self)
- phylip_parse (self)
- phylip_interleaved_parse (self)
- nexus_parse (self)
- nexus_interleaved_parse (self)
- translate_ambiguous (self, seq)
- partitions_parse (self)

Public Attributes

- in_file
- in_file_lines

7.5.1 Detailed Description

```
Parse file contents and return sequences and sequence names
```

Definition at line 538 of file AMAS.py.

7.5.2 Constructor & Destructor Documentation

7.5.2.1 __init__()

```
amas.AMAS.FileParser.__init__ (
               self,
               in_file )
```

Definition at line 541 of file AMAS.py.

```
00541     def __init__(self, in_file):
00542         self.in_file = in_file
00543         with FileHandler(in_file) as handle:
00544             self.in_file_lines = handle.read().rstrip("\r\n")
00545
```

7.5.3 Member Function Documentation

7.5.3.1 fasta_parse()

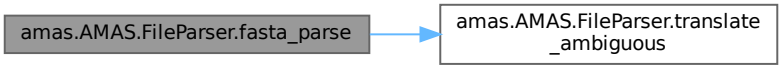
```
amas.AMAS.FileParser.fasta_parse (
               self )
```

Definition at line 546 of file AMAS.py.

```
00546     def fasta_parse(self):
00547         # use regex to parse names and sequences in sequential fasta files
00548         matches = re.finditer(
00549             r"^>(.*[^$])([^>]*)",
00550             self.in_file_lines, re.MULTILINE
00551         )
00552         records = {}
00553
00554         for match in matches:
00555             name_match = match.group(1).replace("\n", "")
00556             seq_match = match.group(2).replace("\n", "").upper()
00557             seq_match = self.translate_ambiguous(seq_match)
00558             records[name_match] = seq_match
00559
00560         return records
00561
```

References amas.AMAS.FileParser.in_file_lines, and amas.AMAS.FileParser.translate_ambiguous().

Here is the call graph for this function:



7.5.3.2 nexus_interleaved_parse()

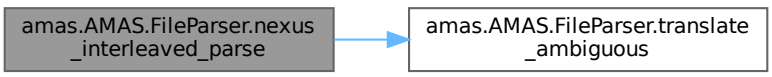
```
amas.AMAS.FileParser.nexus_interleaved_parse (
                self )
```

Definition at line 671 of file AMAS.py.

```
00671     def nexus_interleaved_parse(self):
00672         # use regex to parse names and sequences in sequential nexus files
00673         # find the matrix block
00674         matches = re.finditer(
00675             r"(\s+)?(MATRIX\n|matrix\n|MATRIX\r\n|matrix\r\n)(.*?;)",
00676             self.in_file_lines, re.DOTALL
00677         )
00678         # initiate lists for taxa names and sequence strings on separate lines
00679         taxa = []
00680         sequences = []
00681         # initiate a dictionary for the name:sequence records
00682         records = {}
00683 
00684         for match in matches:
00685             matrix_match = match.group(3)
00686             # get names and sequences from the matrix block
00687             seq_matches = re.finditer(
00688                 r"^(\s+)?[']?(\S+\s\S+|\S+)[']?\s+([A-Za-z*?.{}-]+)($|\s+\[[0-9]+\]$)",
00689                 matrix_match, re.MULTILINE
00690             )
00691 
00692             for match in seq_matches:
00693                 name_match = match.group(2)
00694                 if name_match not in taxa:
00695                     taxa.append(name_match)
00696                 seq_match = match.group(3)
00697 
00698                 sequences.append(seq_match)
00699 
00700         # initiate a counter to keep track of sequences strung together
00701         # from separate lines
00702         counter = 0
00703 
00704         for taxon_no in range(len(taxa)):
00705 
00706             full_length_sequence = "".join([sequences[index] for index in
      range(counter,len(sequences),len(taxa))])
00707             records[taxa[taxon_no]] = self.translate_ambiguous(full_length_sequence).replace("\n",
      "").upper()
00708             counter += 1
00709 
00710         return records
00711 
```

References amas.AMAS.FileParser.in_file_lines, and amas.AMAS.FileParser.translate_ambiguous().

Here is the call graph for this function:



7.5.3.3 nexus_parse()

```
amas.AMAS.FileParser.nexus_parse (
                self )
```

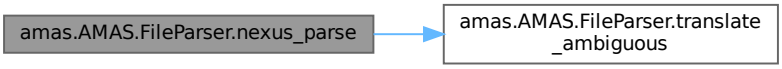
Definition at line 645 of file AMAS.py.

```
00645     def nexus_parse(self):
```

```
00646         # use regex to parse names and sequences in sequential nexus files
00647         # find the matrix block
00648         matches = re.finditer(
00649             r"(\s+)?(MATRIX\n|matrix\n|MATRIX\r\n|matrix\r\n)(.*?;)",
00650             self.in_file_lines, re.DOTALL
00651         )
00652 
00653         records = {}
00654         # get names and sequences from the matrix block
00655 
00656         for match in matches:
00657             matrix_match = match.group(3)
00658             seq_matches = re.finditer(
00659                 r"^(\s+)?['']?(\S+\s\S+|\S+)['']?\s+([A-Za-z*?.{}-]+)($|\s+\[[0-9]+\]$)",
00660                 matrix_match, re.MULTILINE
00661             )
00662 
00663             for match in seq_matches:
00664                 name_match = match.group(2).replace("\n", "")
00665                 seq_match = match.group(3).replace("\n", "").upper()
00666                 seq_match = self.translate_ambiguous(seq_match)
00667                 records[name_match] = seq_match
00668 
00669         return records
00670 
```

References amas.AMAS.FileParser.in_file_lines, and amas.AMAS.FileParser.translate_ambiguous().

Here is the call graph for this function:



7.5.3.4 partitions_parse()

```
amas.AMAS.FileParser.partitions_parse (
            self )
```

Definition at line 731 of file AMAS.py.

```
00731     def partitions_parse(self):
00732         # parse partitions file using regex
00733         # original: `matches = re.finditer(r"^(\s+)?([^ =]+)[ =]+([\\0-9, -]+)", self.in_file_lines,
      re.MULTILINE)`
00734         # new version: more permissive -> handles PartionFinder/RAxML/(IQ-TREE 2)best_scheme.nex
      format partition files
00735         matches = re.finditer(
00736             r"""^[ \t]*                                 # start of line w/ zero-or-more (just)
      whitespaces/tabs
00737                 (
00738                  (?P<nexus>charset[ ]+)                 # case 1: (IQ-TREE 2)best_scheme.nex partition
      directive; partition name
00739                  |
00740                  (?P<raxml>[A-Za-z0-9_+\.]+,[ \t]+)     # case 2: RAxML/RAxML-NG model(+other pars);
      partition name
00741                 )?
00742                 (?P<partition_name>[A-Za-z0-9_\-]+)     # case 3: just partition name (including one
      that contain residual '-out'/'-meta' suffixes)
00743                 [ ]*=[ ]*                               # whitespace-padded (or unpadded) '=':
      (IQ-TREE 2)best_scheme.nex compatabiliy
00744                 (?P<numbers>[\\0-9, -]+)                # position ranges w/ stride (multiple
      intervals; from original regex)
00745                 (?P<nexus_term>[ ]*[;])?                # whitespace-prepended (or unprepended) ';'
      (nexus terminator)
00746             """,
00747             self.in_file_lines,
00748             re.MULTILINE | re.VERBOSE
00749         )
00750 
```

```
00751         # initiate list to store dictionaries with lists
00752         # of slice positions as values
00753         partitions = []
00754         add_to_partitions = partitions.append
00755 
00756         for match in matches:
00757             # initiate dictionary of partition name as key
00758             dict_of_dicts = {}
00759             # and list of dictionaries with slice positions
00760             list_of_dicts = []
00761             add_to_list_of_dicts = list_of_dicts.append
00762             # get parition name and numbers from parsed partition strings
00763             partition_name = match.group('partition_name')
00764             numbers = match.group('numbers')
00765             # remove any whitespace padding '-' (to be consistent with partition-writing format)
00766             numbers = re.sub(r"[ ]*-[ ]*", "-", numbers)
00767             # find all numbers that will be used to parse positions
00768             positions = re.findall(r"([^ ,]+)", numbers)
00769 
00770             for position in positions:
00771                 # create dictionary for slicing input sequence
00772                 # conditioning on whether positions are represented
00773                 # by range, range with stride, or single number
00774                 pos_dict = {}
00775 
00776                 if "-" in position:
00777                     m = re.search(r"([0-9]+)-([0-9]+)", position)
00778                     pos_dict["start"] = int(m.group(1)) - 1
00779                     pos_dict["stop"] = int(m.group(2))
00780                 else:
00781                     pos_dict["start"] = int(position) - 1
00782                     pos_dict["stop"] = int(position)
00783 
00784                 if "\\" in position:
00785                     # Note: the value of `N' in `...\N' isn't read: the script simply assumes `N' is
    consistent with the number of
00786                     # increments per interval when the alignment is parsed with a stride of 3
    (designating each cpos).
00787                     # E.g. For the partition file:
00788                     #       ...'1-N\2'
00789                     #       ...'2-N\2'
00790                     #       ...'(N+1)-M\2'
00791                     #       ...'(N+2)-M\2'
00792                     # 3'cpos are ignored due to the absence of intervals `3-N...', `(N+3)-M...', not
    because the associated stride values are'\2'
00793                     pos_dict["stride"] = 3
00794                 elif "\\" not in position:
00795                     pos_dict["stride"] = 1
00796 
00797                 add_to_list_of_dicts(pos_dict)
00798 
00799             dict_of_dicts[partition_name] = list_of_dicts
00800             add_to_partitions(dict_of_dicts)
00801 
00802         return partitions
00803 
00804 
```

References amas.AMAS.FileParser.in_file_lines.

7.5.3.5 phylip_interleaved_parse()

```
amas.AMAS.FileParser.phylip_interleaved_parse (
              self )
```

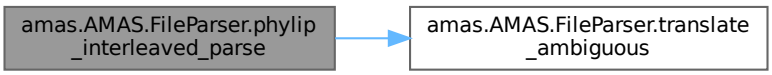
Definition at line 579 of file AMAS.py.

```
00579     def phylip_interleaved_parse(self):
00580         # use regex to parse names and sequences in interleaved phylip files
00581         tax_chars_matches = re.finditer(
00582             r"^(\s+)?([0-9]+)[ \t]+([0-9]+)",
00583             self.in_file_lines, re.MULTILINE
00584         )
00585         name_matches = re.finditer(
00586             r"^(\s+)?(\S+)[ \t]+[A-Za-z*?.{}-]+",
00587             self.in_file_lines, re.MULTILINE
00588         )
00589         seq_matches = re.finditer(
00590             r"(^(\s+)?\S+[ \t]+|^)([A-Za-z*?.{}-]+)$",
00591             self.in_file_lines, re.MULTILINE
00592         )
```

```
00593         # get number of taxa and chars
00594         for match in tax_chars_matches:
00595             tax_match = match.group(2)
00596             chars_match = match.group(3)
00597 
00598         # initiate lists for taxa names and sequence strings on separate lines
00599         taxa = []
00600         sequences = []
00601         # initiate a dictionary for the name:sequence records
00602         records = {}
00603         # initiate a counter to keep track of sequences strung together
00604         # from separate lines
00605         counter = 0
00606 
00607         for match in name_matches:
00608             name_match = match.group(2).replace("\n", "")
00609             taxa.append(name_match)
00610 
00611         for match in seq_matches:
00612             seq_match = match.group(3).replace("\n", "").upper()
00613             seq_match = self.translate_ambiguous(seq_match)
00614             sequences.append(seq_match)
00615         # try parsing PHYLUCE-style interleaved phylip
00616         if len(taxa) != int(tax_match):
00617             taxa = []
00618             sequences = []
00619             matches = re.finditer(
00620                 r"(^(\s+)?(\S+)( ){2,}|^\s+)([ A-Za-z*?.{}-]+)",
00621                 self.in_file_lines, re.MULTILINE
00622             )
00623 
00624             for match in matches:
00625                 try:
00626                     name_match = match.group(3).replace("\n", "")
00627                     taxa.append(name_match)
00628                 except AttributeError:
00629                     pass
00630                 seq_match = match.group(5).replace("\n", "").upper()
00631                 seq_match = "".join(seq_match.split())
00632                 seq_match = self.translate_ambiguous(seq_match)
00633                 sequences.append(seq_match)
00634 
00635         for taxon_no in range(len(taxa)):
00636             sequence = ""
00637             for index in range(counter,len(sequences),len(taxa)):
00638                 sequence += sequences[index]
00639 
00640             records[taxa[taxon_no]] = sequence
00641             counter += 1
00642 
00643         return records
00644 
```

References amas.AMAS.FileParser.in_file_lines, and amas.AMAS.FileParser.translate_ambiguous().

Here is the call graph for this function:



7.5.3.6 phylip_parse()

```
amas.AMAS.FileParser.phylip_parse (
               self )
```

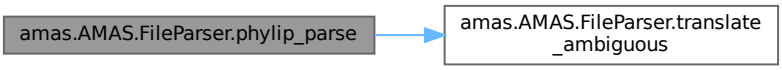
Definition at line 562 of file AMAS.py.

```
00562     def phylip_parse(self):
```

```
00563         # use regex to parse names and sequences in sequential phylip files
00564         matches = re.finditer(
00565             r"^(\s+)?(\S+)\s+([A-Za-z*?.{}-]+)",
00566             self.in_file_lines, re.MULTILINE
00567         )
00568 
00569         records = {}
00570 
00571         for match in matches:
00572             name_match = match.group(2).replace("\n", "")
00573             seq_match = match.group(3).replace("\n", "").upper()
00574             seq_match = self.translate_ambiguous(seq_match)
00575             records[name_match] = seq_match
00576 
00577         return records
00578 
```

References amas.AMAS.FileParser.in_file_lines, and amas.AMAS.FileParser.translate_ambiguous().

Here is the call graph for this function:



7.5.3.7 translate_ambiguous()

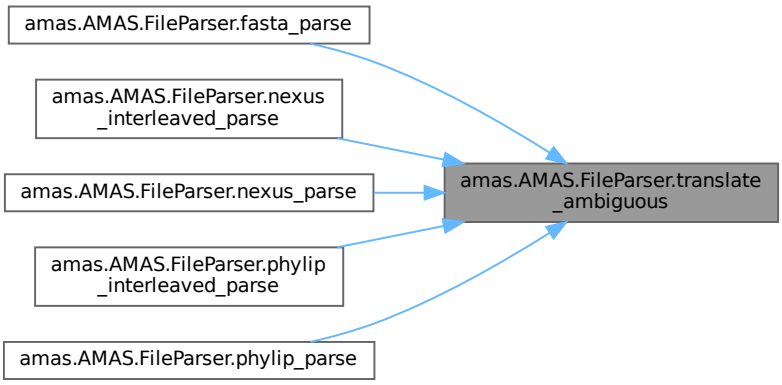
```
amas.AMAS.FileParser.translate_ambiguous (
               self,
               seq )
```

Definition at line 712 of file AMAS.py.

```
00712     def translate_ambiguous(self, seq):
00713         # translate ambiguous characters from curly bracket format
00714         # to single letter format
00715         # also remove spaces from sequences
00716         seq = seq.replace("{GT}", "K")
00717         seq = seq.replace("{AC}", "M")
00718         seq = seq.replace("{AG}", "R")
00719         seq = seq.replace("{CT}", "Y")
00720         seq = seq.replace("{CG}", "S")
00721         seq = seq.replace("{AT}", "W")
00722         seq = seq.replace("{CGT}", "B")
00723         seq = seq.replace("{ACG}", "V")
00724         seq = seq.replace("{ACT}", "H")
00725         seq = seq.replace("{AGT}", "D")
00726         seq = seq.replace("{GATC}", "N")
00727         seq = seq.replace(" ", "")
00728 
00729         return seq
00730 
```

Referenced by amas.AMAS.FileParser.fasta_parse(), amas.AMAS.FileParser.nexus_interleaved_parse(), amas.AMAS.FileParser.nexus_parse(), amas.AMAS.FileParser.phylip_interleaved_parse(), and amas.AMAS.FileParser.phylip_parse(

Here is the caller graph for this function:



7.5.4 Member Data Documentation

7.5.4.1 in_file

`amas.AMAS.FileParser.in_file`

Definition at line 542 of file AMAS.py.

Referenced by amas.AMAS.FileHandler.__exit__(), amas.AMAS.Alignment.get_aln_input(), and amas.AMAS.Alignment.get_name().

7.5.4.2 in_file_lines

`amas.AMAS.FileParser.in_file_lines`

Definition at line 544 of file AMAS.py.

Referenced by amas.AMAS.FileParser.fasta_parse(), amas.AMAS.FileParser.nexus_interleaved_parse(), amas.AMAS.FileParser.nexus_parse(), amas.AMAS.FileParser.partitions_parse(), amas.AMAS.FileParser.phylip_interleaved_parse() and amas.AMAS.FileParser.phylip_parse().

The documentation for this class was generated from the following file:

- amas/AMAS.py

7.6 amas.AMAS.MetaAlignment Class Reference

Public Member Functions

- __init__ (self, ∗∗kwargs)
- translate_dna_to_aa (self, seq, translation_table, frame)
- translate_dict (self, source_dict)
- get_translated (self, translation_table, reading_frame)
- trim_dict (self, alignment)
- get_trimmed (self, trim_fraction, parsimony_check)
- remove_unknown_chars (self, seq)
- remove_empty_sequences (self, split_alignment)
- get_partitions (self, partitions_file)
- get_alignment_object (self, alignment)
- get_alignment_objects (self)
- get_parsed_alignments (self)
- get_partitioned (self, partitions_file)
- get_summaries (self)
- summarize_alignments (self, alignment)
- get_taxon_summaries (self)
- summarize_alignments_taxa (self, alignment)
- write_summaries (self, file_name)
- write_taxa_summaries (self)
- get_replicate (self, no_replicates, no_loci)
- get_concatenated (self, alignments)
- remove_from_alignment (self, alignment, species_to_remove_set, index)
- remove_taxa (self, species_to_remove_set)
- print_fasta (self, source_dict)
- print_phylip (self, source_dict)
- print_phylip_int (self, source_dict)
- print_nexus (self, source_dict)
- print_nexus_int (self, source_dict)
- natural_sort (self, a_list)
- print_unspecified_partitions (self, data_type, codons)
- print_nexus_partitions (self, data_type, codons)
- print_iqtree_nexus_partitions (self, data_type, codons)
- print_raxml_partitions (self, data_type, codons)
- replace_string_in_file (self, file_name, old_string, new_string)
- write_partitions (self, file_name, part_format, data_type, codons)
- get_extension (self, file_format)
- get_metapartition_extension (self, file_format)
- file_overwrite_error (self, file_name)
- write_formatted_file (self, file_format, file_name, alignment)
- get_alignment_name (self, i, extension)
- get_alignment_name_no_ext (self, i)
- write_concat (self, file_format)
- write_convert (self, index, alignment, file_format, extension)
- write_replicate (self, index, alignment, file_format, extension)
- write_split (self, item, file_format, extension)
- write_reduced (self, file_format, extension)
- write_translated (self, index, alignment, file_format, extension)
- write_trimmed (self, index, alignment, file_format, extension)
- write_metapartitions (self, file_format)
- write_out (self, action, file_format)

Public Attributes

- in_files
- in_format
- data_type
- command
- concat_out
- using_metapartitions
- check_align
- cores
- by_taxon_summary
- no_sup_aln_name
- no_mpan
- codons
- no_replicates
- no_loci
- split
- remove_empty
- prepend_label
- species_to_remove
- species_to_remove_set
- reduced_file_prefix
- check_taxa
- reading_frame
- genetic_code
- trim_fraction
- trim_out
- parsimony_check
- alignment_objects
- parsed_alignments
- codes_list
- gencode_NCBI_1
- gencode_NCBI_2
- gencode_NCBI_3
- gencode_NCBI_4
- gencode_NCBI_5
- gencode_NCBI_6
- gencode_NCBI_9
- gencode_NCBI_10
- gencode_NCBI_11
- gencode_NCBI_12
- gencode_NCBI_13
- gencode_NCBI_14
- gencode_NCBI_16
- gencode_NCBI_21
- gencode_NCBI_22
- gencode_NCBI_23
- gencode_NCBI_24
- gencode_NCBI_25
- gencode_NCBI_26
- codes

7.6.1 Detailed Description

```
Class of multiple sequence alignments
```

Definition at line 1165 of file AMAS.py.

7.6.2 Constructor & Destructor Documentation

7.6.2.1 __init__()

```
amas.AMAS.MetaAlignment.__init__ (
              self,
             ** kwargs )
```

Definition at line 1168 of file AMAS.py.

```
01168     def __init__(self, **kwargs):
01169         # set defaults and get values from kwargs
01170         self.in_files = kwargs.get("in_files")
01171         self.in_format = kwargs.get("in_format")
01172         self.data_type = kwargs.get("data_type")
01173         self.command = kwargs.get("command")
01174         self.concat_out = kwargs.get("concat_out", "concatenated.out")
01175         self.using_metapartitions = False
01176         self.check_align = kwargs.get("check_align", False)
01177         self.cores = kwargs.get("cores")
01178         self.by_taxon_summary = kwargs.get("by_taxon_summary")
01179         self.no_sup_aln_name = False
01180         self.no_mpan = False
01181
01182         if self.command == "concat":
01183             self.codons = kwargs.get("codons", "none")
01184             if self.data_type == "aa" and self.codons != "none":
01185                 print("ERROR: when option -d|--data-type is set to 'aa', option -n|--codons must be
      set to 'none'.")
01186                 sys.exit(1)
01187
01188         if self.command == "replicate":
01189             self.no_replicates = kwargs.get("replicate_args")[0]
01190             self.no_loci = kwargs.get("replicate_args")[1]
01191
01192         if self.command == "split":
01193             self.split = kwargs.get("split_by")
01194             self.remove_empty = kwargs.get("remove_empty", False)
01195             self.no_sup_aln_name = kwargs.get("no_sup_aln_name", False)
01196
01197         if self.command == "metapartitions":
01198             self.using_metapartitions = True
01199             self.split = kwargs.get("split_by")
01200             self.remove_empty = kwargs.get("remove_empty", False)
01201             self.no_sup_aln_name = kwargs.get("no_sup_aln_name", False)
01202             self.no_mpan = kwargs.get("no_mpan", False)
01203             self.prepend_label = kwargs.get("prepend_label")
01204             if self.prepend_label is not None and isinstance(self.prepend_label, str):
01205                 self.prepend_label = self.prepend_label + "_"
01206             else:
01207                 self.prepend_label = ""
01208
01209         if self.command == "remove":
01210             self.species_to_remove = kwargs.get("taxa_to_remove")
01211             self.species_to_remove_set = set(self.species_to_remove)
01212             self.reduced_file_prefix = kwargs.get("out_prefix")
01213             self.check_taxa = kwargs.get("check_taxa", False)
01214
01215         if self.command == "translate":
01216             self.reading_frame = kwargs.get("reading_frame")
01217             self.genetic_code = kwargs.get("genetic_code")
01218
01219         if self.command == "trim":
01220             self.trim_fraction = kwargs.get("trim_fraction")
01221             self.trim_out = kwargs.get("trim_out")
01222             self.parsimony_check = kwargs.get("parsimony_check", False)
01223
01224         self.alignment_objects = self.get_alignment_objects()
01225         self.parsed_alignments = self.get_parsed_alignments()
01226
```

```
01227         # The code list:
01228         self.codes_list = """
01229          1. The Standard Code
01230          2. The Vertebrate Mitochondrial Code
01231          3. The Yeast Mitochondrial Code
01232          4. The Mold, Protozoan, and Coelenterate Mitochondrial Code and the Mycoplasma/Spiroplasma
     Code
01233          5. The Invertebrate Mitochondrial Code
01234          6. The Ciliate, Dasycladacean and Hexamita Nuclear Code
01235          9. The Echinoderm and Flatworm Mitochondrial Code
01236         10. The Euplotid Nuclear Code
01237         11. The Bacterial, Archaeal and Plant Plastid Code
01238         12. The Alternative Yeast Nuclear Code
01239         13. The Ascidian Mitochondrial Code
01240         14. The Alternative Flatworm Mitochondrial Code
01241         16. Chlorophycean Mitochondrial Code
01242         21. Trematode Mitochondrial Code
01243         22. Scenedesmus obliquus Mitochondrial Code
01244         23. Thraustochytrium Mitochondrial Code
01245         24. Pterobranchia Mitochondrial Code
01246         25. Candidate Division SR1 and Gracilibacteria Code
01247         26. Pachysolen tannophilus Nuclear Code
01248         """
01249
01250         # 1: The Standard Code
01251         self.gencode_NCBI_1 = {
01252         "TTT" : "F", # Phe
01253         "TCT" : "S", # Ser
01254         "TAT" : "Y", # Tyr
01255         "TGT" : "C", # Cys
01256         "TTC" : "F", # Phe
01257         "TCC" : "S", # Ser
01258         "TAC" : "Y", # Tyr
01259         "TGC" : "C", # Cys
01260         "TTA" : "L", # Leu
01261         "TCA" : "S", # Ser
01262         "TAA" : "*", # Ter
01263         "TGA" : "*", # Ter
01264         "TTG" : "L", # Leu i
01265         "TCG" : "S", # Ser
01266         "TAG" : "*", # Ter
01267         "TGG" : "W", # Trp
01268         "CTT" : "L", # Leu
01269         "CCT" : "P", # Pro
01270         "CAT" : "H", # His
01271         "CGT" : "R", # Arg
01272         "CTC" : "L", # Leu
01273         "CCC" : "P", # Pro
01274         "CAC" : "H", # His
01275         "CGC" : "R", # Arg
01276         "CTA" : "L", # Leu
01277         "CCA" : "P", # Pro
01278         "CAA" : "Q", # Gln
01279         "CGA" : "R", # Arg
01280         "CTG" : "L", # Leu i
01281         "CCG" : "P", # Pro
01282         "CAG" : "Q", # Gln
01283         "CGG" : "R", # Arg
01284         "ATT" : "I", # Ile
01285         "ACT" : "T", # Thr
01286         "AAT" : "N", # Asn
01287         "AGT" : "S", # Ser
01288         "ATC" : "I", # Ile
01289         "ACC" : "T", # Thr
01290         "AAC" : "N", # Asn
01291         "AGC" : "S", # Ser
01292         "ATA" : "I", # Ile
01293         "ACA" : "T", # Thr
01294         "AAA" : "K", # Lys
01295         "AGA" : "R", # Arg
01296         "ATG" : "M", # Met i
01297         "ACG" : "T", # Thr
01298         "AAG" : "K", # Lys
01299         "AGG" : "R", # Arg
01300         "GTT" : "V", # Val
01301         "GCT" : "A", # Ala
01302         "GAT" : "D", # Asp
01303         "GGT" : "G", # Gly
01304         "GTC" : "V", # Val
01305         "GCC" : "A", # Ala
01306         "GAC" : "D", # Asp
01307         "GGC" : "G", # Gly
01308         "GTA" : "V", # Val
01309         "GCA" : "A", # Ala
01310         "GAA" : "E", # Glu
01311         "GGA" : "G", # Gly
01312         "GTG" : "V", # Val
```

```
01313         "GCG" : "A", # Ala
01314         "GAG" : "E", # Glu
01315         "GGG" : "G", # Gly
01316         "---" : "-", # Gap
01317         "???" : "?", # Unk
01318         "NNN" : "X", # Unk
01319         }
01320
01321         # 2: The Vertebrate Mitochondrial Code
01322         self.gencode_NCBI_2 = self.gencode_NCBI_1.copy()
01323         self.gencode_NCBI_2["AGA"] = "*" # Ter
01324         self.gencode_NCBI_2["AGG"] = "*" # Ter
01325         self.gencode_NCBI_2["ATA"] = "M" # Met
01326         self.gencode_NCBI_2["TGA"] = "W" # Trp
01327
01328         # 3: The Yeast Mitochondrial Code
01329         self.gencode_NCBI_3 = self.gencode_NCBI_1.copy()
01330         self.gencode_NCBI_3["ATA"] = "M" # Met
01331         self.gencode_NCBI_3["CTT"] = "T" # Thr
01332         self.gencode_NCBI_3["CTC"] = "T" # Thr
01333         self.gencode_NCBI_3["CTA"] = "T" # Thr
01334         self.gencode_NCBI_3["CTG"] = "T" # Thr
01335         self.gencode_NCBI_3["TGA"] = "W" # Trp
01336
01337         del self.gencode_NCBI_3["CGA"]
01338         del self.gencode_NCBI_3["CGC"]
01339
01340         # 4: The Mold, Protozoan, and Coelenterate Mitochondrial Code and the Mycoplasma/Spiroplasma
      Code
01341         self.gencode_NCBI_4 = self.gencode_NCBI_1.copy()
01342         self.gencode_NCBI_4["TGA"] = "W" # Trp
01343
01344         # 5: The Invertebrate Mitochondrial Code
01345         self.gencode_NCBI_5 = self.gencode_NCBI_1.copy()
01346         self.gencode_NCBI_5["AGA"] = "S" # Ser
01347         self.gencode_NCBI_5["AGG"] = "S" # Ser
01348         self.gencode_NCBI_5["ATA"] = "M" # Met
01349         self.gencode_NCBI_5["TGA"] = "W" # Trp
01350
01351         # 6: The Ciliate, Dasycladacean and Hexamita Nuclear Code
01352         self.gencode_NCBI_6 = self.gencode_NCBI_1.copy()
01353         self.gencode_NCBI_6["TAA"] = "Q" # Gln
01354         self.gencode_NCBI_6["TAG"] = "Q" # Gln
01355
01356         # 9: The Echinoderm and Flatworm Mitochondrial Code
01357         self.gencode_NCBI_9 = self.gencode_NCBI_1.copy()
01358         self.gencode_NCBI_9["AAA"] = "N" # Asn
01359         self.gencode_NCBI_9["AGA"] = "S" # Ser
01360         self.gencode_NCBI_9["AGG"] = "S" # Ser
01361         self.gencode_NCBI_9["TGA"] = "W" # Trp
01362
01363         # 10: The Euplotid Nuclear Code
01364         self.gencode_NCBI_10 = self.gencode_NCBI_1.copy()
01365         self.gencode_NCBI_10["TGA"] = "C" # Cys
01366
01367         # 11: The Bacterial, Archaeal and Plant Plastid Code
01368         self.gencode_NCBI_11 = self.gencode_NCBI_1.copy()
01369
01370         # 12: The Alternative Yeast Nuclear Code
01371         self.gencode_NCBI_12 = self.gencode_NCBI_1.copy()
01372         self.gencode_NCBI_12["CTG"] = "S" # Ser
01373
01374         # 13: The Ascidian Mitochondrial Code
01375         self.gencode_NCBI_13 = self.gencode_NCBI_1.copy()
01376         self.gencode_NCBI_13["AGA"] = "G" # Gly
01377         self.gencode_NCBI_13["AGG"] = "G" # Gly
01378         self.gencode_NCBI_13["ATA"] = "M" # Met
01379         self.gencode_NCBI_13["TGA"] = "W" # Trp
01380
01381         # 14: The Alternative Flatworm Mitochondrial Code
01382         self.gencode_NCBI_14 = self.gencode_NCBI_1.copy()
01383         self.gencode_NCBI_14["AAA"] = "N" # Asn
01384         self.gencode_NCBI_14["AGA"] = "S" # Ser
01385         self.gencode_NCBI_14["AGG"] = "S" # Ser
01386         self.gencode_NCBI_14["TAA"] = "Y" # Tyr
01387         self.gencode_NCBI_14["TGA"] = "W" # Trp
01388
01389         # 16: Chlorophycean Mitochondrial Code
01390         self.gencode_NCBI_16 = self.gencode_NCBI_1.copy()
01391         self.gencode_NCBI_16["TAG"] = "L" # Leu
01392
01393         # 21: Trematode Mitochondrial Code
01394         self.gencode_NCBI_21 = self.gencode_NCBI_1.copy()
01395         self.gencode_NCBI_21["TGA"] = "W" # Trp
01396         self.gencode_NCBI_21["ATA"] = "M" # Met
01397         self.gencode_NCBI_21["AGA"] = "S" # Ser
01398         self.gencode_NCBI_21["AGG"] = "S" # Ser
```

```
01399         self.gencode_NCBI_21["AAA"] = "N" # Asn
01400
01401         # 22: Scenedesmus obliquus Mitochondrial Code
01402         self.gencode_NCBI_22 = self.gencode_NCBI_1.copy()
01403         self.gencode_NCBI_22["TCA"] = "*" # Ter
01404         self.gencode_NCBI_22["TAG"] = "L" # Leu
01405
01406         # 23: Thraustochytrium Mitochondrial Code
01407         self.gencode_NCBI_23 = self.gencode_NCBI_1.copy()
01408         self.gencode_NCBI_23["TTA"] = "*" # Ter
01409
01410         # 24: Pterobranchia Mitochondrial Code
01411         self.gencode_NCBI_24 = self.gencode_NCBI_1.copy()
01412         self.gencode_NCBI_24["AGA"] = "S" # Ser
01413         self.gencode_NCBI_24["AGG"] = "K" # Lys
01414         self.gencode_NCBI_24["TGA"] = "W" # Trp
01415
01416         # 25: Candidate Division SR1 and Gracilibacteria Code
01417         self.gencode_NCBI_25 = self.gencode_NCBI_1.copy()
01418         self.gencode_NCBI_25["TGA"] = "G" # Gly
01419
01420         # 26: Pachysolen tannophilus Nuclear Code
01421         self.gencode_NCBI_26 = self.gencode_NCBI_1.copy()
01422         self.gencode_NCBI_26["CTG"] = "A" # Ala
01423
01424         self.codes = {
01425         1 : self.gencode_NCBI_1,
01426         2 : self.gencode_NCBI_2,
01427         3 : self.gencode_NCBI_3,
01428         4 : self.gencode_NCBI_4,
01429         5 : self.gencode_NCBI_5,
01430         6 : self.gencode_NCBI_6,
01431         9 : self.gencode_NCBI_9,
01432         10 : self.gencode_NCBI_10,
01433         11 : self.gencode_NCBI_11,
01434         12 : self.gencode_NCBI_12,
01435         13 : self.gencode_NCBI_13,
01436         14 : self.gencode_NCBI_14,
01437         16 : self.gencode_NCBI_16,
01438         21 : self.gencode_NCBI_21,
01439         22 : self.gencode_NCBI_22,
01440         23 : self.gencode_NCBI_23,
01441         24 : self.gencode_NCBI_24,
01442         25 : self.gencode_NCBI_25,
01443         26 : self.gencode_NCBI_26
01444         }
01445
```

7.6.3 Member Function Documentation

7.6.3.1 file_overwrite_error()

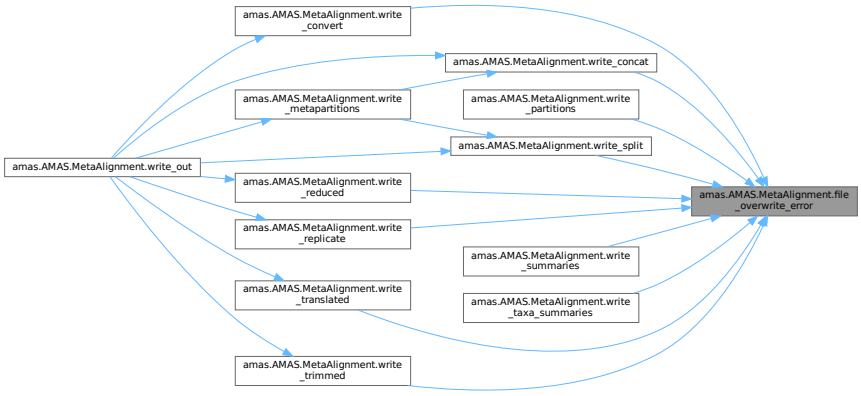
```
amas.AMAS.MetaAlignment.file_overwrite_error (
                self,
                file_name )
```

Definition at line 2157 of file AMAS.py.

```
02157     def file_overwrite_error(self, file_name):
02158         # print warning when overwriting a file
02159         if path.exists(file_name):
02160             print("WARNING: You are overwriting '" + file_name + "'")
02161
```

Referenced by amas.AMAS.MetaAlignment.write_concat(), amas.AMAS.MetaAlignment.write_convert(), amas.AMAS.MetaAlignment
amas.AMAS.MetaAlignment.write_reduced(), amas.AMAS.MetaAlignment.write_replicate(), amas.AMAS.MetaAlignment.write_split()
amas.AMAS.MetaAlignment.write_summaries(), amas.AMAS.MetaAlignment.write_taxa_summaries(), amas.AMAS.MetaAlignment.w
and amas.AMAS.MetaAlignment.write_trimmed().

Here is the caller graph for this function:



7.6.3.2 get_alignment_name()

```
amas.AMAS.MetaAlignment.get_alignment_name (
              self,
              i,
              extension )
```

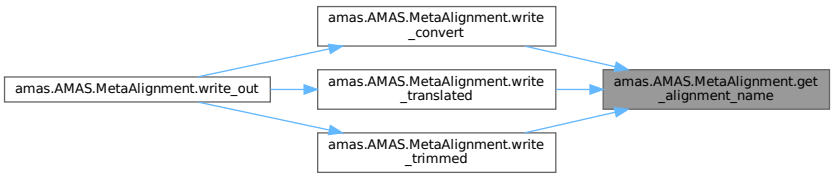
Definition at line 2176 of file AMAS.py.

```
02176     def get_alignment_name(self, i, extension):
02177         # get file name
02178         file_name = self.alignment_objects[i].get_name() + extension
02179 
02180         return file_name
02181 
```

References amas.AMAS.MetaAlignment.alignment_objects.

Referenced by amas.AMAS.MetaAlignment.write_convert(), amas.AMAS.MetaAlignment.write_translated(), and amas.AMAS.MetaAlignment.write_trimmed().

Here is the caller graph for this function:



7.6.3.3 get_alignment_name_no_ext()

```
amas.AMAS.MetaAlignment.get_alignment_name_no_ext (
              self,
              i )
```

Definition at line 2182 of file AMAS.py.

```
02182     def get_alignment_name_no_ext(self, i):
02183         # get file name without extension
02184         file_name = self.alignment_objects[i].get_name()
02185 
02186         return file_name
02187 
```

References amas.AMAS.MetaAlignment.alignment_objects.

Referenced by amas.AMAS.MetaAlignment.remove_from_alignment().

Here is the caller graph for this function:

7.6.3.4 get_alignment_object()

```
amas.AMAS.MetaAlignment.get_alignment_object (
              self,
              alignment )
```

Definition at line 1523 of file AMAS.py.

```
01523     def get_alignment_object(self, alignment):
01524         # parse according to the given alphabet;
01525         # Note:('alignment') <=> `in_file` outside MetaAlignment, e.g.
01526         #
      AminoAcidAlignment(Alignment<-.get_parsed_aln<-.get_aln_input)<-FileParser.__init__(in_file)<-FileHandler(...open(self.
01527         if self.data_type == "aa":
01528             aln = AminoAcidAlignment(alignment, self.in_format, self.data_type)
01529         elif self.data_type == "dna":
01530             aln = DNAAlignment(alignment, self.in_format, self.data_type)
01531         return aln
01532 
```

References amas.AMAS.Alignment.data_type, amas.AMAS.MetaAlignment.data_type, amas.AMAS.Alignment.in_format, and amas.AMAS.MetaAlignment.in_format.

Referenced by amas.AMAS.MetaAlignment.get_alignment_objects().

Here is the caller graph for this function:

7.6.3.5 get_alignment_objects()

```
amas.AMAS.MetaAlignment.get_alignment_objects (
              self )
```

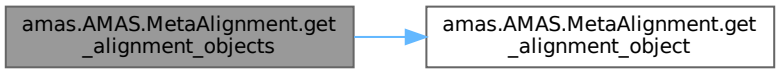
Definition at line 1533 of file AMAS.py.

```
01533     def get_alignment_objects(self):
01534         # get alignment objects on which statistics can be computed
01535         # use multiprocessing if more than one core specified
01536         if int(self.cores) == 1:
01537             alignments = [self.get_alignment_object(alignment) for alignment in self.in_files]
01538         elif int(self.cores) > 1:
01539             pool = mp.Pool(int(self.cores))
01540             alignments = pool.map(self.get_alignment_object, self.in_files)
01541         return alignments
01542
```

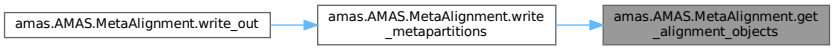
References amas.AMAS.MetaAlignment.cores, amas.AMAS.MetaAlignment.get_alignment_object(), and amas.AMAS.MetaAlignment.in_files.

Referenced by amas.AMAS.MetaAlignment.write_metapartitions().

Here is the call graph for this function:



Here is the caller graph for this function:



7.6.3.6 get_concatenated()

```
amas.AMAS.MetaAlignment.get_concatenated (
              self,
              alignments )
```

Definition at line 1746 of file AMAS.py.

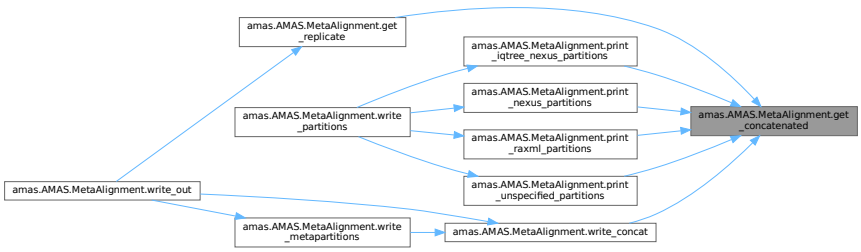
```
01746     def get_concatenated(self, alignments):
01747         # concatenate muntiple input alignments
01748         # create empty dictionary of lists
01749         concatenated = defaultdict(list)
01750
01751         # first create list of taxa in all alignments
01752         # you need this to insert empty seqs in
01753         # the concatenated alignment
01754         all_taxa = []
01755         for alignment in alignments:
01756             for taxon in alignment.keys():
01757                 if taxon not in all_taxa:
01758                     all_taxa.append(taxon)
01759
```

```
01760         # start counters to keep track of partitions
01761         partition_counter = 1
01762         position_counter = 1
01763         # get dict for alignment name and partition
01764         partitions = {}
01765         digits_to_pad = len(str(len(alignments)))
01766 
01767         for alignment in alignments:
01768             # get alignment length from a random taxon
01769             partition_length = len(alignment[list(alignment.keys())[0]])
01770             # get base name of each alignment for use when writing partitions file
01771             # NOTE: the base name here is whatever comes before fist period in the file name
01772             alignment_name = self.alignment_objects[partition_counter - 1].get_name().split('.')[0]
01773 
01774             if self.using_metapartitions:
01775                 # implementation of option --no-mpan; option --prepend(-label) will assign a string or
      "" (see class definition)
01776                 if self.no_mpan:
01777                     # omit original alignment names from the printed partition file
01778                     partition_name = self.prepend_label + "p" +
      str(partition_counter).zfill(digits_to_pad)
01779                 else:
01780                     # keep original alignment names in the printed partition file
01781                     partition_name = self.prepend_label + "p" +
      str(partition_counter).zfill(digits_to_pad) + "_" + alignment_name
01782             else:
01783                 partition_name = "p" + str(partition_counter) + "_" + alignment_name
01784 
01785             start = position_counter
01786             position_counter += partition_length
01787             end = position_counter - 1
01788             partitions[partition_name] = str(start) + "-" + str(end)
01789             partition_counter += 1
01790 
01791             # get empty sequence if there is missing taxon
01792             # getting length from first element of list of keys
01793             # created from the original dict for this alignment
01794             empty_seq = '?' * partition_length
01795 
01796             for taxon in all_taxa:
01797 
01798                 if taxon not in alignment.keys():
01799                     concatenated[taxon].append(empty_seq)
01800                 else:
01801                     concatenated[taxon].append(alignment[taxon])
01802 
01803         concatenated = {taxon:''.join(seqs) for taxon, seqs in concatenated.items()}
01804 
01805         return concatenated, partitions
01806 
```

References amas.AMAS.MetaAlignment.alignment_objects, amas.AMAS.MetaAlignment.no_mpan, amas.AMAS.MetaAlignment.prep
amas.AMAS.MetaAlignment.split, and amas.AMAS.MetaAlignment.using_metapartitions.

Referenced by amas.AMAS.MetaAlignment.get_replicate(), amas.AMAS.MetaAlignment.print_iqtree_nexus_partitions(),
amas.AMAS.MetaAlignment.print_nexus_partitions(), amas.AMAS.MetaAlignment.print_raxml_partitions(),
amas.AMAS.MetaAlignment.print_unspecified_partitions(), and amas.AMAS.MetaAlignment.write_concat().

Here is the caller graph for this function:



7.6.3.7 get_extension()

```
amas.AMAS.MetaAlignment.get_extension (
                self,
                file_format )
```

Definition at line 2127 of file AMAS.py.

```
02127     def get_extension(self, file_format):
02128         # get proper extension string
02129         if file_format == "phylip":
02130             extension = "-out.phy"
02131         elif file_format == "phylip-int":
02132             extension = "-out.int-phy"
02133         elif file_format == "fasta":
02134             extension = "-out.fas"
02135         elif file_format == "nexus":
02136             extension = "-out.nex"
02137         elif file_format == "nexus-int":
02138             extension = "-out.int-nex"
02139 
02140         return extension
02141 
```

Referenced by amas.AMAS.MetaAlignment.write_out().

Here is the caller graph for this function:

7.6.3.8 get_metapartition_extension()

```
amas.AMAS.MetaAlignment.get_metapartition_extension (
                self,
                file_format )
```

Definition at line 2142 of file AMAS.py.

```
02142     def get_metapartition_extension(self, file_format):
02143         # get proper metapartition_extension string
02144         if file_format == "phylip":
02145             metapartition_extension = "-meta.phy"
02146         elif file_format == "phylip-int":
02147             metapartition_extension = "-meta.int-phy"
02148         elif file_format == "fasta":
02149             metapartition_extension = "-meta.fas"
02150         elif file_format == "nexus":
02151             metapartition_extension = "-meta.nex"
02152         elif file_format == "nexus-int":
02153             metapartition_extension = "-meta.int-nex"
02154 
02155         return metapartition_extension
02156 
```

Referenced by amas.AMAS.MetaAlignment.write_metapartitions().

Here is the caller graph for this function:

7.6.3.9 get_parsed_alignments()

```
amas.AMAS.MetaAlignment.get_parsed_alignments (
               self )
```

Definition at line 1543 of file AMAS.py.

```
01543     def get_parsed_alignments(self):
01544         # get parsed dictionaries with taxa and sequences
01545         parsed_alignments = []
01546         add_to_parsed_alignments = parsed_alignments.append
01547         for alignment in self.alignment_objects:
01548             parsed = alignment.parsed_aln
01549             add_to_parsed_alignments(parsed)
01550             # checking if every seq has the same length or if parsed is not empty; exit if false
01551             if self.check_align:
01552                 equal = all(
01553                     x == [len(list(parsed.values())[i]) for i in
      range(0,len(list(parsed.values())))][0]
01554                     for x in [len(list(parsed.values())[i]) for i in
      range(0,len(list(parsed.values())))]
01555                 )
01556                 if equal is False:
01557                     print("ERROR: Sequences in input are of varying lengths. Be sure to align them
      first.")
01558                     sys.exit()
01559
01560             if not parsed.keys() or not any(parsed.values()):
01561                 print(
01562                     "ERROR: Parsed sequences of " + alignment.in_file + " are empty. "
01563                     "Are you sure you specified the right input format and/or that input is a valid
      alignment?"
01564                 )
01565                 sys.exit()
01566
01567         return parsed_alignments
01568
```

References amas.AMAS.MetaAlignment.alignment_objects, and amas.AMAS.MetaAlignment.check_align.

Referenced by amas.AMAS.MetaAlignment.write_metapartitions().

Here is the caller graph for this function:

7.6.3.10 get_partitioned()

```
amas.AMAS.MetaAlignment.get_partitioned (
               self,
               partitions_file )
```

Definition at line 1569 of file AMAS.py.

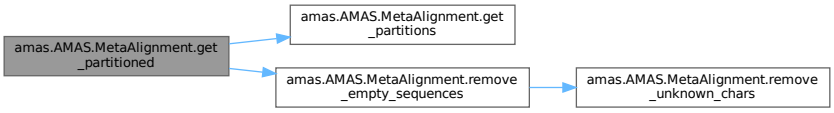
```
01569     def get_partitioned(self, partitions_file):
01570         # partition alignment according to a partitions file
01571         partitions = self.get_partitions(partitions_file)
01572         alignment = self.parsed_alignments[0]
01573
01574         # initiate list of newly partitioned alignments
01575         list_of_parts = []
01576         add_to_list_of_parts = list_of_parts.append
01577         for partition in partitions:
01578             # loop over all parsed partitions, adding taxa and sliced sequences
01579             for name, elements in partition.items():
01580                 new_dict = {}
01581
```

```
01582                 for taxon, seq in alignment.items():
01583                     new_seq = ""
01584 
01585                     for dictionary in elements:
01586                         new_seq = new_seq +
      seq[dictionary["start"]:dictionary["stop"]:dictionary["stride"]]
01587                         new_dict[taxon] = new_seq
01588 
01589                 if self.remove_empty:
01590                     # check if remove empty sequences
01591                     no_empty_dict = self.remove_empty_sequences(new_dict)
01592                     add_to_list_of_parts({name : no_empty_dict})
01593                 else:
01594                     # add partition name : dict of taxa and sequences to the list
01595                     add_to_list_of_parts({name : new_dict})
01596 
01597         return list_of_parts
01598 
```

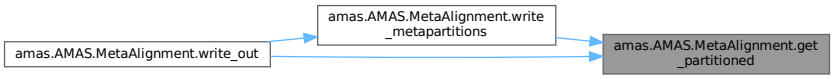
References amas.AMAS.MetaAlignment.get_partitions(), amas.AMAS.MetaAlignment.parsed_alignments, amas.AMAS.MetaAlignment.remove_empty, and amas.AMAS.MetaAlignment.remove_empty_sequences().

Referenced by amas.AMAS.MetaAlignment.write_metapartitions(), and amas.AMAS.MetaAlignment.write_out().

Here is the call graph for this function:



Here is the caller graph for this function:



7.6.3.11 get_partitions()

```
amas.AMAS.MetaAlignment.get_partitions (
              self,
              partitions_file )
```

Definition at line 1516 of file AMAS.py.

```
01516     def get_partitions(self, partitions_file):
01517         # parse and get partitions from partitions file
01518         partitions = FileParser(partitions_file)
01519         parsed_partitions = partitions.partitions_parse()
01520 
01521         return parsed_partitions
01522 
```

Referenced by amas.AMAS.MetaAlignment.get_partitioned().

Here is the caller graph for this function:

7.6.3.12 get_replicate()

```
amas.AMAS.MetaAlignment.get_replicate (
               self,
               no_replicates,
               no_loci )
```

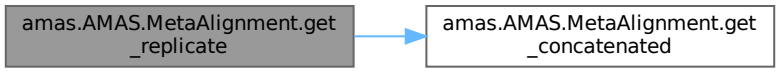
Definition at line 1726 of file AMAS.py.

```
01726     def get_replicate(self, no_replicates, no_loci):
01727         # construct replicate data sets for phylogenetic jackknife
01728         replicates = []
01729         add_to_replicates = replicates.append
01730         counter = 1
01731         for replicate in range(no_replicates):
01732 
01733             try:
01734                 random_alignments = sample(self.parsed_alignments, no_loci)
01735             except ValueError:
01736                 print("ERROR: You specified more loci per replicate than there are in your input.")
01737                 sys.exit()
01738 
01739             random_alignments = sample(self.parsed_alignments, no_loci)
01740             concat_replicate = self.get_concatenated(random_alignments)[0]
01741             add_to_replicates(concat_replicate)
01742             counter += 1
01743 
01744         return replicates
01745 
```

References amas.AMAS.MetaAlignment.get_concatenated(), and amas.AMAS.MetaAlignment.parsed_alignments.

Referenced by amas.AMAS.MetaAlignment.write_out().

Here is the call graph for this function:



Here is the caller graph for this function:

7.6.3.13 get_summaries()

```
amas.AMAS.MetaAlignment.get_summaries (
               self )
```

Definition at line 1599 of file AMAS.py.

```
01599     def get_summaries(self):
```

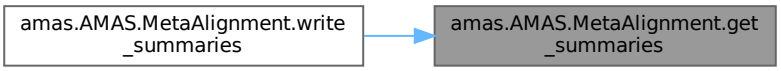
```
01600         # get summaries for all alignment objects
01601
01602         # define different headers for aa and dna alignments
01603         aa_header = [
01604             "Alignment_name",
01605             "No_of_taxa",
01606             "Alignment_length",
01607             "Total_matrix_cells",
01608             "Undetermined_characters",
01609             "Missing_percent",
01610             "No_variable_sites",
01611             "Proportion_variable_sites",
01612             "Parsimony_informative_sites",
01613             "Proportion_parsimony_informative"
01614         ]
01615
01616         dna_header = [
01617             "Alignment_name",
01618             "No_of_taxa",
01619             "Alignment_length",
01620             "Total_matrix_cells",
01621             "Undetermined_characters",
01622             "Missing_percent",
01623             "No_variable_sites",
01624             "Proportion_variable_sites",
01625             "Parsimony_informative_sites",
01626             "Proportion_parsimony_informative",
01627             "AT_content",
01628             "GC_content"
01629         ]
01630
01631         alignments = self.alignment_objects
01632         parsed_alignments = self.parsed_alignments
01633         freq_header = [char for char in alignments[0].alphabet]
01634
01635         if self.data_type == "aa":
01636             header = aa_header + freq_header
01637         elif self.data_type == "dna":
01638             header = dna_header + freq_header
01639
01640         # use multiprocessing if more than one core specified
01641         if int(self.cores) == 1:
01642             summaries = [alignment.get_summary() for alignment in alignments]
01643         elif int(self.cores) > 1:
01644             pool = mp.Pool(int(self.cores))
01645             summaries = pool.map(self.summarize_alignments, alignments)
01646         return header, summaries
01647
```

References amas.AMAS.MetaAlignment.alignment_objects, amas.AMAS.MetaAlignment.cores, amas.AMAS.Alignment.data_type, amas.AMAS.MetaAlignment.data_type, amas.AMAS.MetaAlignment.parsed_alignments, and amas.AMAS.MetaAlignment.summarize

Referenced by amas.AMAS.MetaAlignment.write_summaries().

Here is the call graph for this function:

Here is the caller graph for this function:



7.6.3.14 get_taxon_summaries()

```
amas.AMAS.MetaAlignment.get_taxon_summaries (
              self )
```

Definition at line 1653 of file AMAS.py.

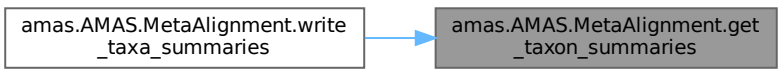
```
01653     def get_taxon_summaries(self):
01654         # get per-sequence summaries for all alignment objects
01655
01656         # define different headers for aa and dna alignments
01657         aa_header = [
01658             "Alignment_name",
01659             "Taxon_name",
01660             "Sequence_length",
01661             "Undetermined_characters",
01662             "Missing_percent"
01663         ]
01664
01665         dna_header = [
01666             "Alignment_name",
01667             "Taxon_name",
01668             "Sequence_length",
01669             "Undetermined_characters",
01670             "Missing_percent",
01671             "AT_content",
01672             "GC_content"
01673         ]
01674
01675         alignments = self.alignment_objects
01676         parsed_alignments = self.parsed_alignments
01677         freq_header = alignments[0].alphabet
01678
01679         if self.data_type == "aa":
01680             header = aa_header + freq_header
01681         elif self.data_type == "dna":
01682             header = dna_header + freq_header
01683
01684         # use multiprocessing if more than one core specified
01685         if int(self.cores) == 1:
01686             summaries = [alignment.get_taxa_summary() for alignment in alignments]
01687         elif int(self.cores) > 1:
01688             pool = mp.Pool(int(self.cores))
01689             summaries = pool.map(self.summarize_alignments_taxa, alignments)
01690
01691         return header, summaries
01692
```

References amas.AMAS.MetaAlignment.alignment_objects, amas.AMAS.MetaAlignment.cores, amas.AMAS.Alignment.data_type, amas.AMAS.MetaAlignment.data_type, amas.AMAS.MetaAlignment.parsed_alignments, and amas.AMAS.MetaAlignment.summarize

Referenced by amas.AMAS.MetaAlignment.write_taxa_summaries().

Here is the call graph for this function:

Here is the caller graph for this function:



7.6.3.15 get_translated()

```
amas.AMAS.MetaAlignment.get_translated (
               self,
               translation_table,
               reading_frame )
```

Definition at line 1478 of file AMAS.py.

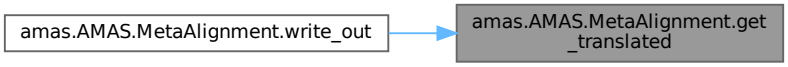
```
01478     def get_translated(self, translation_table, reading_frame):
01479         if int(self.cores) == 1:
01480             translated_alignments = [self.translate_dict(alignment) for alignment in
      self.parsed_alignments]
01481         elif int(self.cores) > 1:
01482             pool = mp.Pool(int(self.cores))
01483             translated_alignments = pool.map(self.translate_dict, self.parsed_alignments)
01484
01485         return translated_alignments
01486
```

References amas.AMAS.MetaAlignment.cores, amas.AMAS.MetaAlignment.parsed_alignments, and amas.AMAS.MetaAlignment.tra

Referenced by amas.AMAS.MetaAlignment.write_out().

Here is the call graph for this function:

Here is the caller graph for this function:



7.6.3.16 get_trimmed()

```
amas.AMAS.MetaAlignment.get_trimmed (
               self,
               trim_fraction,
               parsimony_check )
```

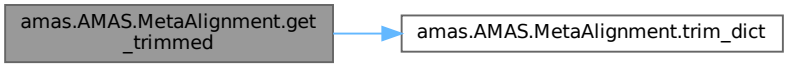
Definition at line 1495 of file AMAS.py.

```
01495     def get_trimmed(self, trim_fraction, parsimony_check):
01496         if int(self.cores) == 1:
01497             trimmed_alignments = [self.trim_dict(alignment) for alignment in self.alignment_objects]
01498         elif int(self.cores) > 1:
01499             pool = mp.Pool(int(self.cores))
01500             trimmed_alignments = pool.map(self.trim_dict, self.alignment_objects)
01501
01502         return trimmed_alignments
01503
```

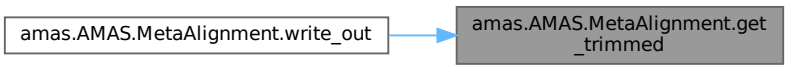
References amas.AMAS.MetaAlignment.alignment_objects, amas.AMAS.MetaAlignment.cores, and amas.AMAS.MetaAlignment.trim

Referenced by amas.AMAS.MetaAlignment.write_out().

Here is the call graph for this function:



Here is the caller graph for this function:



7.6.3.17 natural_sort()

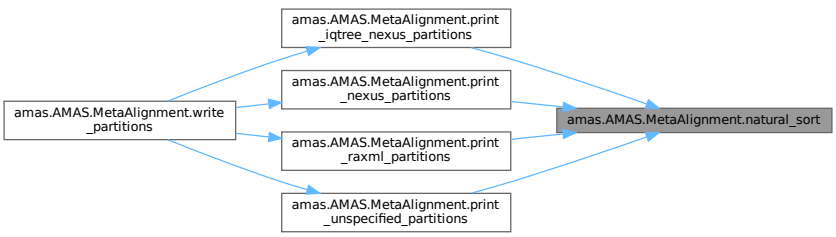
```
amas.AMAS.MetaAlignment.natural_sort (
              self,
              a_list )
```

Definition at line 1960 of file AMAS.py.

```
01960     def natural_sort(self, a_list):
01961         # create a function that does 'human sort' on a list
01962         convert = lambda text: int(text) if text.isdigit() else text.lower()
01963         alphanum_key = lambda key: [convert(c) for c in re.split('([0-9]+)', key)]
01964         return sorted(a_list, key = alphanum_key)
01965
```

Referenced by amas.AMAS.MetaAlignment.print_iqtree_nexus_partitions(), amas.AMAS.MetaAlignment.print_nexus_partitions(), amas.AMAS.MetaAlignment.print_raxml_partitions(), and amas.AMAS.MetaAlignment.print_unspecified_partitions().

Here is the caller graph for this function:



7.6.3.18 print_fasta()

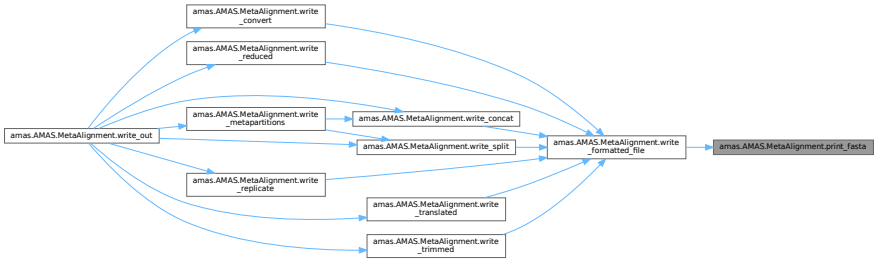
```
amas.AMAS.MetaAlignment.print_fasta (
                self,
                source_dict )
```

Definition at line 1833 of file AMAS.py.

```
01833     def print_fasta(self, source_dict):
01834         # print fasta-formatted string from a dictionary
01835         fasta_string = ""
01836         # each sequence line will have 80 characters
01837         n = 80
01838 
01839         for taxon, seq in sorted(source_dict.items()):
01840             # split dictionary values to a list of string, each n chars long
01841             seq = [seq[i:i+n] for i in range(0, len(seq), n)]
01842             # in case there are unwanted spaces in taxon names
01843             taxon = taxon.replace(" ", "_").strip("'")
01844             fasta_string += ">" + taxon + "\n"
01845             for element in seq:
01846                 fasta_string += element + "\n"
01847 
01848         return fasta_string
01849 
```

Referenced by amas.AMAS.MetaAlignment.write_formatted_file().

Here is the caller graph for this function:



7.6.3.19 print_iqtree_nexus_partitions()

```
amas.AMAS.MetaAlignment.print_iqtree_nexus_partitions (
                self,
                data_type,
                codons )
```

Definition at line 2027 of file AMAS.py.

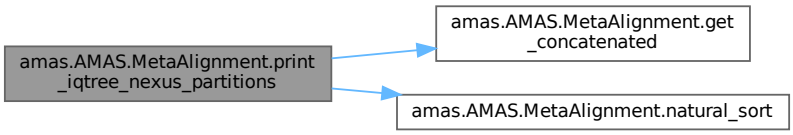
```
02027     def print_iqtree_nexus_partitions(self, data_type, codons):
02028         # print partitions for concatenated alignment
02029         part_string = ""
02030         part_dict = self.get_concatenated(self.parsed_alignments)[1]
02031         part_list = self.natural_sort(part_dict.keys())
02032         # write beginning of nexus sets
02033         part_string += "#nexus\n"
02034         part_string += "begin sets;\n"
02035 
02036         if data_type == "dna":
02037             if codons == "none":
02038                 for key in part_list:
02039                     part_string += "  charset " + key + " = " + str(part_dict[key]) + ";\n"
02040             elif codons == "12":
02041                 for key in part_list:
02042                     start, end = str(part_dict[key]).split("-")
02043                     part_string += "  charset " + key + "_pos1" + " = " + start + " - " + end + "\\2"
      + ";\n"
```

```
02044                     part_string += "  charset " + key + "_pos2" + " = " + str(int(start) + 1) + " - "
      + end + "\\2" + ";\n"
02045                 elif codons == "123":
02046                     for key in part_list:
02047                         start, end = str(part_dict[key]).split("-")
02048                         part_string += "  charset " + key + "_pos1" + " = " + start + " - " + end + "\\3"
      + ";\n"
02049                         part_string += "  charset " + key + "_pos2" + " = " + str(int(start) + 1) + " - "
      + end + "\\3" + ";\n"
02050                         part_string += "  charset " + key + "_pos3" + " = " + str(int(start) + 2) + " - "
      + end + "\\3" + ";\n"
02051                 part_string += "end;\n"
02052
02053             elif data_type == "aa":
02054                 for key in part_list:
02055                     part_string += "  charset " + key + " = " + str(part_dict[key]) + ";\n"
02056                 part_string += "end;\n"
02057
02058             return part_string
02059
```

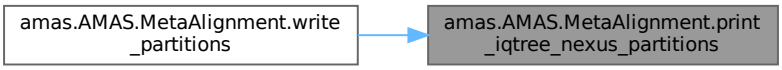
References amas.AMAS.MetaAlignment.get_concatenated(), amas.AMAS.MetaAlignment.natural_sort(), amas.AMAS.MetaAlignmen and amas.AMAS.MetaAlignment.split.

Referenced by amas.AMAS.MetaAlignment.write_partitions().

Here is the call graph for this function:



Here is the caller graph for this function:



7.6.3.20 print_nexus()

```
amas.AMAS.MetaAlignment.print_nexus (
              self,
              source_dict )
```

Definition at line 1897 of file AMAS.py.

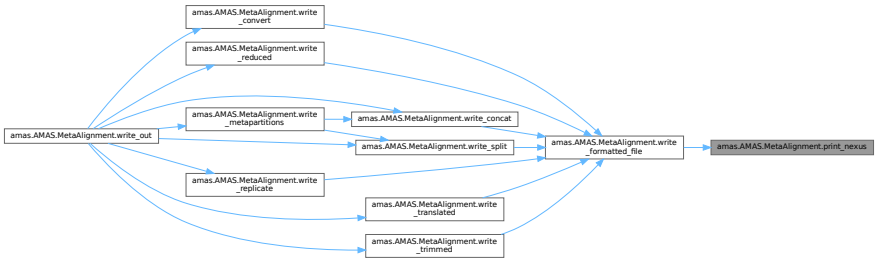
```
01897     def print_nexus(self, source_dict):
01898         # print nexus-formatted string from a dictionary
01899         if self.data_type == "aa" or self.command == "translate":
01900             data_type = "PROTEIN"
01901         elif self.data_type == "dna":
01902             data_type = "DNA"
01903
```

```
01904         taxa_list = list(source_dict.keys())
01905         no_taxa = len(taxa_list)
01906         pad_longest_name = len(max(taxa_list, key=len)) + 3
01907         seq_length = len(next(iter(source_dict.values())))
01908         header = str(len(source_dict)) + " " + str(seq_length)
01909         nexus_string = (
01910             "#NEXUS\n\nBEGIN DATA;\n\tDIMENSIONS  NTAX=" + str(no_taxa) + " NCHAR=" + str(seq_length)
01911             + ";\n\tFORMAT DATATYPE=" + data_type + "  GAP = - MISSING = ?;\n\tMATRIX\n"
01912         )
01913 
01914         for taxon, seq in sorted(source_dict.items()):
01915             taxon = taxon.replace(" ", "_").strip("'")
01916             nexus_string += "\t" + taxon.ljust(pad_longest_name, ' ') + seq + "\n"
01917         nexus_string += "\n;\n\nEND;"
01918 
01919         return nexus_string
01920 
```

References amas.AMAS.MetaAlignment.command, amas.AMAS.Alignment.data_type, and amas.AMAS.MetaAlignment.data_type.

Referenced by amas.AMAS.MetaAlignment.write_formatted_file().

Here is the caller graph for this function:



7.6.3.21 print_nexus_int()

```
amas.AMAS.MetaAlignment.print_nexus_int (
              self,
              source_dict )
```

Definition at line 1921 of file AMAS.py.

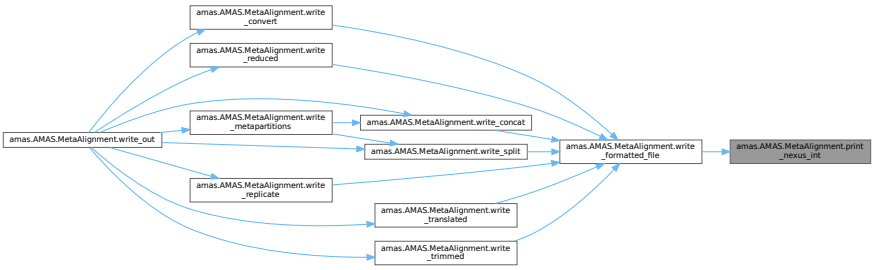
```
01921     def print_nexus_int(self, source_dict):
01922         # print nexus interleaved-formatted string from a dictionary
01923         if self.data_type == "aa":
01924             data_type = "PROTEIN"
01925         elif self.data_type == "dna":
01926             data_type = "DNA"
01927 
01928         taxa_list = list(source_dict.keys())
01929         no_taxa = len(taxa_list)
01930         pad_longest_name = len(max(taxa_list, key=len)) + 3
01931         seq_length = len(next(iter(source_dict.values())))
01932         header = str(len(source_dict)) + " " + str(seq_length)
01933         # this will be a list of tuples to hold taxa names and sequences
01934         seq_matrix = []
01935         nexus_int_string = (
01936             "#NEXUS\n\nBEGIN DATA;\n\tDIMENSIONS  NTAX=" + str(no_taxa) + " NCHAR=" + str(seq_length)
01937             + ";\n\tFORMAT   INTERLEAVE" + "   DATATYPE=" + data_type + "  GAP = - MISSING =
      ?;\n\tMATRIX\n"
01938         )
01939         # each sequence line will have 500 characters
01940         n = 500
01941 
01942         # recreate sequence matrix
01943         add_to_matrix = seq_matrix.append
01944         for taxon, seq in sorted(source_dict.items()):
01945             add_to_matrix((taxon, [seq[i:i+n] for i in range(0, len(seq), n)]))
01946 
```

```
01947         first_seq = seq_matrix[0][1]
01948         for index, item in enumerate(first_seq):
01949             for taxon, sequence in seq_matrix:
01950                 if index == 0:
01951                     nexus_int_string += taxon.ljust(pad_longest_name, ' ') + sequence[index] + "\n"
01952                 else:
01953                     nexus_int_string += sequence[index] + "\n"
01954             nexus_int_string += "\n"
01955 
01956         nexus_int_string += "\n;\n\nEND;"
01957 
01958         return nexus_int_string
01959 
```

References amas.AMAS.Alignment.data_type, and amas.AMAS.MetaAlignment.data_type.

Referenced by amas.AMAS.MetaAlignment.write_formatted_file().

Here is the caller graph for this function:



7.6.3.22 print_nexus_partitions()

```
amas.AMAS.MetaAlignment.print_nexus_partitions (
              self,
              data_type,
              codons )
```

Definition at line 1994 of file AMAS.py.

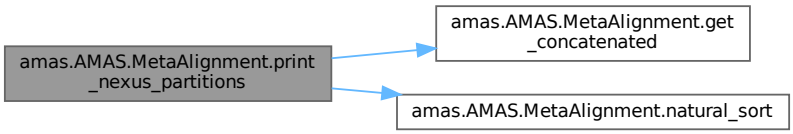
```
01994     def print_nexus_partitions(self, data_type, codons):
01995         # print partitions for concatenated alignment
01996         part_string = ""
01997         part_dict = self.get_concatenated(self.parsed_alignments)[1]
01998         part_list = self.natural_sort(part_dict.keys())
01999         # write beginning of nexus sets
02000         part_string += "#NEXUS\n\n"
02001         part_string += "BEGIN SETS;\n"
02002 
02003         if data_type == "dna":
02004             if codons == "none":
02005                 for key in part_list:
02006                     part_string += "\tcharset " + key + " = " + str(part_dict[key]) + ";\n"
02007             elif codons == "12":
02008                 for key in part_list:
02009                     start, end = str(part_dict[key]).split("-")
02010                     part_string += "\tcharset " + key + "_pos1" + " = " + start + "-" + end + "\\2" +
      ";\n"
02011                     part_string += "\tcharset " + key + "_pos2" + " = " + str(int(start) + 1) + "-" +
      end + "\\2" + ";\n"
02012             elif codons == "123":
02013                 for key in part_list:
02014                     start, end = str(part_dict[key]).split("-")
02015                     part_string += "\tcharset " + key + "_pos1" + " = " + start + "-" + end + "\\3" +
      ";\n"
02016                     part_string += "\tcharset " + key + "_pos2" + " = " + str(int(start) + 1) + "-" +
      end + "\\3" + ";\n"
02017                     part_string += "\tcharset " + key + "_pos3" + " = " + str(int(start) + 2) + "-" +
      end + "\\3" + ";\n"
02018             part_string += "END;"
```

```
02019
02020         elif data_type == "aa":
02021             for key in part_list:
02022                 part_string += "\tcharset " + key + " = " + str(part_dict[key]) + ";\n"
02023             part_string += "END;"
02024
02025         return part_string
02026
```

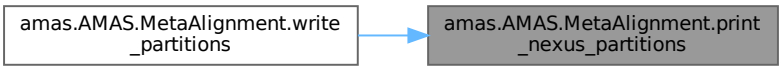
References amas.AMAS.MetaAlignment.get_concatenated(), amas.AMAS.MetaAlignment.natural_sort(), amas.AMAS.MetaAlignmen and amas.AMAS.MetaAlignment.split.

Referenced by amas.AMAS.MetaAlignment.write_partitions().

Here is the call graph for this function:



Here is the caller graph for this function:



7.6.3.23 print_phylip()

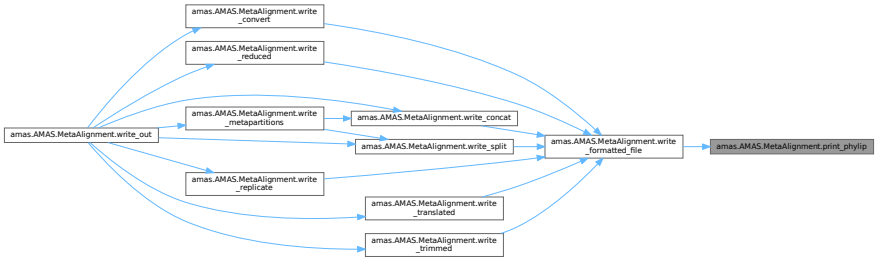
```
amas.AMAS.MetaAlignment.print_phylip (
                self,
                source_dict )
```

Definition at line 1850 of file AMAS.py.

```
01850     def print_phylip(self, source_dict):
01851         # print phylip-formatted string from a dictionary
01852         taxa_list = list(source_dict.keys())
01853         no_taxa = len(taxa_list)
01854         # figure out the max length of a taxon for nice padding of sequences
01855         pad_longest_name = len(max(taxa_list, key=len)) + 3
01856         # get sequence length from a random value
01857         seq_length = len(next(iter(source_dict.values())))
01858         header = str(len(source_dict)) + " " + str(seq_length)
01859         phylip_string = header + "\n"
01860         for taxon, seq in sorted(source_dict.items()):
01861             taxon = taxon.replace(" ", "_").strip("'")
01862             # left-justify taxon names relative to sequences
01863             phylip_string += taxon.ljust(pad_longest_name, ' ') + seq + "\n"
01864
01865         return phylip_string
01866
```

Referenced by amas.AMAS.MetaAlignment.write_formatted_file().

Here is the caller graph for this function:



7.6.3.24 print_phylip_int()

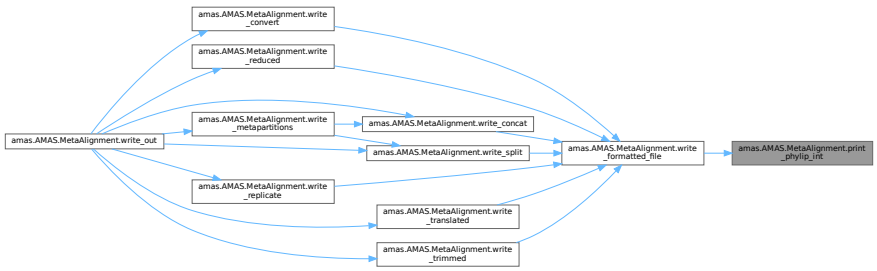
```
amas.AMAS.MetaAlignment.print_phylip_int (
            self,
            source_dict )
```

Definition at line 1867 of file AMAS.py.

```
01867     def print_phylip_int(self, source_dict):
01868         # print phylip interleaved-formatted string from a dictionary
01869         taxa_list = list(source_dict.keys())
01870         no_taxa = len(taxa_list)
01871         pad_longest_name = len(max(taxa_list, key=len)) + 3
01872         seq_length = len(next(iter(source_dict.values())))
01873         header = str(len(source_dict)) + " " + str(seq_length)
01874         phylip_int_string = header + "\n\n"
01875         # this will be a list of tuples to hold taxa names and sequences
01876         seq_matrix = []
01877 
01878         # each sequence line will have 500 characters
01879         n = 500
01880 
01881         # recreate sequence matrix
01882         add_to_matrix = seq_matrix.append
01883         for taxon, seq in sorted(source_dict.items()):
01884             add_to_matrix((taxon, [seq[i:i+n] for i in range(0, len(seq), n)]))
01885 
01886         first_seq = seq_matrix[0][1]
01887         for index, item in enumerate(first_seq):
01888             for taxon, sequence in seq_matrix:
01889                 if index == 0:
01890                     phylip_int_string += taxon.ljust(pad_longest_name, ' ') + sequence[index] + "\n"
01891                 else:
01892                     phylip_int_string += sequence[index] + "\n"
01893             phylip_int_string += "\n"
01894 
01895         return phylip_int_string
01896 
```

Referenced by amas.AMAS.MetaAlignment.write_formatted_file().

Here is the caller graph for this function:



7.6.3.25 print_raxml_partitions()

```
amas.AMAS.MetaAlignment.print_raxml_partitions (
                self,
                data_type,
                codons )
```

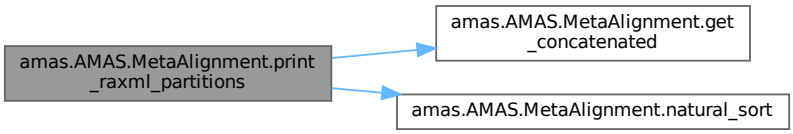
Definition at line 2060 of file AMAS.py.

```
02060     def print_raxml_partitions(self, data_type, codons):
02061         # print partitions for concatenated alignment
02062         part_string = ""
02063         part_dict = self.get_concatenated(self.parsed_alignments)[1]
02064         part_list = self.natural_sort(part_dict.keys())
02065 
02066         if data_type == "dna":
02067             if codons == "none":
02068                 for key in part_list:
02069                     part_string += "DNA, " + key + " = " + str(part_dict[key]) + "\n"
02070             elif codons == "12":
02071                 for key in part_list:
02072                     start, end = str(part_dict[key]).split("-")
02073                     part_string += "DNA, " + key + "_pos1" + " = " + start + "-" + end + "\\2" + "\n"
02074                     part_string += "DNA, " + key + "_pos2" + " = " + str(int(start) + 1) + "-" + end +
      "\\2" + "\n"
02075             elif codons == "123":
02076                 for key in part_list:
02077                     start, end = str(part_dict[key]).split("-")
02078                     part_string += "DNA, " + key + "_pos1" + " = " + start + "-" + end + "\\3" + "\n"
02079                     part_string += "DNA, " + key + "_pos2" + " = " + str(int(start) + 1) + "-" + end +
      "\\3" + "\n"
02080                     part_string += "DNA, " + key + "_pos3" + " = " + str(int(start) + 2) + "-" + end +
      "\\3" + "\n"
02081 
02082         elif data_type == "aa":
02083             for key in part_list:
02084                 part_string += "WAG, " + key + " = " + str(part_dict[key]) + "\n"
02085 
02086             #  aa-partition files with strides are probably not useful? (original below)
02087 #             elif codons == "12":
02088 #                 for key in part_list:
02089 #                     start, end = str(part_dict[key]).split("-")
02090 #                     part_string += "WAG, " + key + "_pos1" + " = " + start + "-" + end + "\\2" + "\n"
02091 #                     part_string += "WAG, " + key + "_pos2" + " = " + str(int(start) + 1) + "-" + end
      + "\\2" + "\n"
02092 #             elif codons == "123":
02093 #                 for key in part_list:
02094 #                     start, end = str(part_dict[key]).split("-")
02095 #                     part_string += "WAG, " + key + "_pos1" + " = " + start + "-" + end + "\\3" + "\n"
02096 #                     part_string += "WAG, " + key + "_pos2" + " = " + str(int(start) + 1) + "-" + end
      + "\\3" + "\n"
02097 #                     part_string += "WAG, " + key + "_pos3" + " = " + str(int(start) + 2) + "-" + end
      + "\\3" + "\n"
02098         return part_string
02099 
```

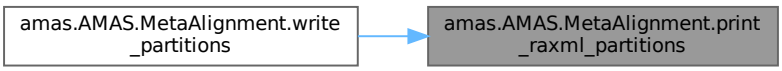
References amas.AMAS.MetaAlignment.get_concatenated(), amas.AMAS.MetaAlignment.natural_sort(), amas.AMAS.MetaAlignmen
and amas.AMAS.MetaAlignment.split.

Referenced by amas.AMAS.MetaAlignment.write_partitions().

Here is the call graph for this function:



Here is the caller graph for this function:



7.6.3.26 print_unspecified_partitions()

```
amas.AMAS.MetaAlignment.print_unspecified_partitions (
              self,
              data_type,
              codons )
```

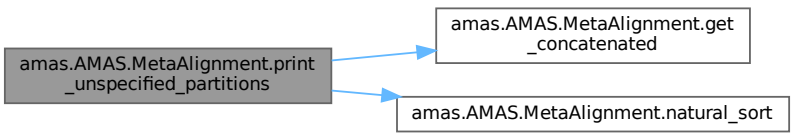
Definition at line 1966 of file AMAS.py.

```
01966     def print_unspecified_partitions(self, data_type, codons):
01967         # print partitions for concatenated alignment
01968         part_string = ""
01969         part_dict = self.get_concatenated(self.parsed_alignments)[1]
01970         part_list = self.natural_sort(part_dict.keys())
01971 
01972         if data_type == "dna":
01973             if codons == "none":
01974                 for key in part_list:
01975                     part_string += key + " = " + str(part_dict[key]) + "\n"
01976             elif codons == "12":
01977                 for key in part_list:
01978                     start, end = str(part_dict[key]).split("-")
01979                     part_string += key + "_pos1" + " = " + start + "-" + end + "\\2" + "\n"
01980                     part_string += key + "_pos2" + " = " + str(int(start) + 1) + "-" + end + "\\2" +
      "\n"
01981             elif codons == "123":
01982                 for key in part_list:
01983                     start, end = str(part_dict[key]).split("-")
01984                     part_string += key + "_pos1" + " = " + start + "-" + end + "\\3" + "\n"
01985                     part_string += key + "_pos2" + " = " + str(int(start) + 1) + "-" + end + "\\3" +
      "\n"
01986                     part_string += key + "_pos3" + " = " + str(int(start) + 2) + "-" + end + "\\3" +
      "\n"
01987 
01988         elif data_type == "aa":
01989             for key in part_list:
01990                 part_string += key + " = " + str(part_dict[key]) + "\n"
01991 
01992         return part_string
01993 
```

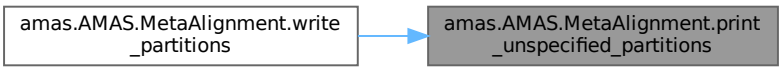
References amas.AMAS.MetaAlignment.get_concatenated(), amas.AMAS.MetaAlignment.natural_sort(), amas.AMAS.MetaAlignmen
and amas.AMAS.MetaAlignment.split.

Referenced by amas.AMAS.MetaAlignment.write_partitions().

Here is the call graph for this function:



Here is the caller graph for this function:



7.6.3.27 remove_empty_sequences()

```
amas.AMAS.MetaAlignment.remove_empty_sequences (
              self,
              split_alignment )
```

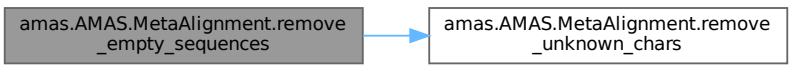
Definition at line 1510 of file AMAS.py.

```
01510     def remove_empty_sequences(self, split_alignment):
01511         # remove taxa from alignment if they are composed of only empty sequences
01512         new_alignment = {taxon : seq for taxon, seq in split_alignment.items() if
      self.remove_unknown_chars(seq)}
01513
01514         return new_alignment
01515
```

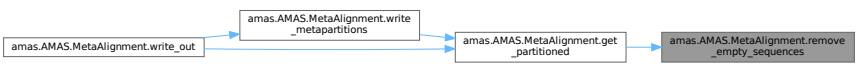
References amas.AMAS.MetaAlignment.remove_unknown_chars().

Referenced by amas.AMAS.MetaAlignment.get_partitioned().

Here is the call graph for this function:



Here is the caller graph for this function:



7.6.3.28 remove_from_alignment()

```
amas.AMAS.MetaAlignment.remove_from_alignment (
               self,
               alignment,
               species_to_remove_set,
               index )
```

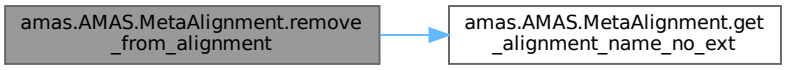
Definition at line 1807 of file AMAS.py.

```
01807     def remove_from_alignment(self, alignment, species_to_remove_set, index):
01808         # remove taxa from alignment
01809         aln_name = self.get_alignment_name_no_ext(index)
01810         for taxon in species_to_remove_set:
01811             if taxon not in alignment.keys():
01812                 print(
01813                     "WARNING: Taxon '" + taxon + "' not found in '" + aln_name + "'.\nIf you expected
      it to be there, "
01814                     "make sure to replace all taxon name spaces with underscores and that you are not
      using quotes."
01815                 )
01816         # originally within for-loop scope (redundancy)
01817         new_alignment = {species: seq for species, seq in alignment.items() if species not in
      species_to_remove_set}
01818
01819         return aln_name, new_alignment
01820
```

References amas.AMAS.MetaAlignment.get_alignment_name_no_ext().

Referenced by amas.AMAS.MetaAlignment.remove_taxa().

Here is the call graph for this function:



Here is the caller graph for this function:

7.6.3.29 remove_taxa()

```
amas.AMAS.MetaAlignment.remove_taxa (
               self,
               species_to_remove_set )
```

Definition at line 1821 of file AMAS.py.

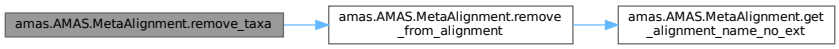
```
01821     def remove_taxa(self, species_to_remove_set):
01822         new_alns = {}
01823         for index, alignment in enumerate(self.parsed_alignments):
01824             aln_name, aln_dict = self.remove_from_alignment(alignment, species_to_remove_set, index)
01825             # check if alignment is not empty:
01826             if aln_dict:
```

```
01827                 new_alns[aln_name] = aln_dict
01828             else:
01829                 print("ERROR: You asked to remove all taxa from the alignment " + aln_name + ". No
      output file will be written.")
01830 
01831         return new_alns
01832 
```

References amas.AMAS.MetaAlignment.parsed_alignments, and amas.AMAS.MetaAlignment.remove_from_alignment().

Referenced by amas.AMAS.MetaAlignment.write_reduced().

Here is the call graph for this function:



Here is the caller graph for this function:

7.6.3.30 remove_unknown_chars()

```
amas.AMAS.MetaAlignment.remove_unknown_chars (
              self,
              seq )
```

Definition at line 1504 of file AMAS.py.

```
01504     def remove_unknown_chars(self, seq):
01505         # remove unknown characters from sequence
01506         new_seq = seq.replace("?", "").replace("-", "")
01507 
01508         return new_seq
01509 
```

Referenced by amas.AMAS.MetaAlignment.remove_empty_sequences().

Here is the caller graph for this function:

7.6.3.31 replace_string_in_file()

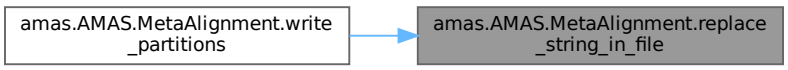
```
amas.AMAS.MetaAlignment.replace_string_in_file (
              self,
              file_name,
              old_string,
              new_string )
```

Definition at line 2100 of file AMAS.py.

```
02100     def replace_string_in_file(self, file_name, old_string, new_string):
02101         # global string replacement in file
02102         with open(file_name, "r", encoding="utf-8") as file:
02103             file_content = file.read()
02104         # write globally replaced content back to file
02105         glb_replaced_content = file_content.replace(old_string, new_string)
02106         with open(file_name, "w", encoding="utf-8") as file:
02107             file.write(glb_replaced_content)
02108
```

Referenced by amas.AMAS.MetaAlignment.write_partitions().

Here is the caller graph for this function:



7.6.3.32 summarize_alignments()

```
amas.AMAS.MetaAlignment.summarize_alignments (
              self,
              alignment )
```

Definition at line 1648 of file AMAS.py.

```
01648     def summarize_alignments(self, alignment):
01649         # helper function to summarize alignments
01650         summary = alignment.get_summary()
01651         return summary
01652
```

Referenced by amas.AMAS.MetaAlignment.get_summaries().

Here is the caller graph for this function:

7.6.3.33 summarize_alignments_taxa()

```
amas.AMAS.MetaAlignment.summarize_alignments_taxa (
                self,
                alignment )
```

Definition at line 1693 of file AMAS.py.

```
01693     def summarize_alignments_taxa(self, alignment):
01694         # helper function to summarize alignments by taxon
01695         summary = alignment.get_taxa_summary()
01696         return summary
01697
```

Referenced by amas.AMAS.MetaAlignment.get_taxon_summaries().

Here is the caller graph for this function:

7.6.3.34 translate_dict()

```
amas.AMAS.MetaAlignment.translate_dict (
                self,
                source_dict )
```

Definition at line 1467 of file AMAS.py.

```
01467     def translate_dict(self, source_dict):
01468         translation_table = self.codes.get(self.genetic_code)
01469         translated_dict = {}
01470         for taxon, seq in sorted(source_dict.items()):
01471             translated_seq = self.translate_dna_to_aa(seq, translation_table, self.reading_frame)
01472             if "*" in translated_seq:
01473                 print("WARNING: stop codon(s), indicated as *, found in {} sequence".format(taxon))
01474             translated_dict[taxon] = translated_seq
01475
01476         return translated_dict
01477
```

References amas.AMAS.MetaAlignment.codes, amas.AMAS.MetaAlignment.genetic_code, amas.AMAS.MetaAlignment.reading_fra and amas.AMAS.MetaAlignment.translate_dna_to_aa().

Referenced by amas.AMAS.MetaAlignment.get_translated().

Here is the call graph for this function:

Here is the caller graph for this function:

7.6.3.35 translate_dna_to_aa()

```
amas.AMAS.MetaAlignment.translate_dna_to_aa (
              self,
              seq,
              translation_table,
              frame )
```

Definition at line 1446 of file AMAS.py.

```
01446     def translate_dna_to_aa(self, seq, translation_table, frame):
01447         # translate DNA string into amino acids
01448         # where the last codon starts
01449         last_codon_start = len(seq) - 2
01450         # where the first codon starts
01451         if frame == 1:
01452             first = 0
01453         elif frame == 2:
01454             first = 1
01455         elif frame == 3:
01456             first = 2
01457         # create protein sequence by growing list
01458         protein = []
01459         add_to_protein = protein.append
01460         for start in range(first, last_codon_start, 3):
01461             codon = seq[start : start + 3]
01462             aa = translation_table.get(codon.upper(), 'X')
01463             add_to_protein(aa)
01464 
01465         return "".join(protein)
01466 
```

Referenced by amas.AMAS.MetaAlignment.translate_dict().

Here is the caller graph for this function:

7.6.3.36 trim_dict()

```
amas.AMAS.MetaAlignment.trim_dict (
              self,
              alignment )
```

Definition at line 1487 of file AMAS.py.

```
01487     def trim_dict(self, alignment):
01488         trim_vector = alignment.get_trim_selection(self.trim_fraction, self.parsimony_check)
01489         aln_dict = alignment.parsed_aln
01490         for key in aln_dict:
01491             aln_dict[key] = ''.join(list(compress(aln_dict[key], trim_vector)))
01492 
01493         return aln_dict
01494 
```

References amas.AMAS.MetaAlignment.parsimony_check, and amas.AMAS.MetaAlignment.trim_fraction.

Referenced by amas.AMAS.MetaAlignment.get_trimmed().

Here is the caller graph for this function:

7.6.3.37 write_concat()

```
amas.AMAS.MetaAlignment.write_concat (
            self,
            file_format )
```

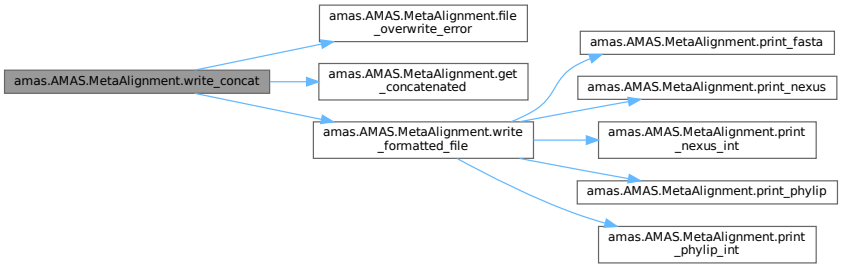
Definition at line 2188 of file AMAS.py.

```
02188     def write_concat(self, file_format):
02189         # write concatenated alignment into a file
02190         concatenated_alignment = self.get_concatenated(self.parsed_alignments)[0]
02191         file_name = self.concat_out
02192         self.file_overwrite_error(file_name)
02193         self.write_formatted_file(file_format, file_name, concatenated_alignment)
02194 
02195         print("Wrote concatenated sequences to " + file_format + " file '" + file_name + "'")
02196 
```

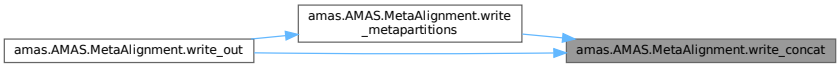
References amas.AMAS.MetaAlignment.concat_out, amas.AMAS.MetaAlignment.file_overwrite_error(), amas.AMAS.MetaAlignment.
amas.AMAS.MetaAlignment.parsed_alignments, and amas.AMAS.MetaAlignment.write_formatted_file().

Referenced by amas.AMAS.MetaAlignment.write_metapartitions(), and amas.AMAS.MetaAlignment.write_out().

Here is the call graph for this function:



Here is the caller graph for this function:



7.6.3.38 write_convert()

```
amas.AMAS.MetaAlignment.write_convert (
            self,
            index,
            alignment,
            file_format,
            extension )
```

Definition at line 2197 of file AMAS.py.

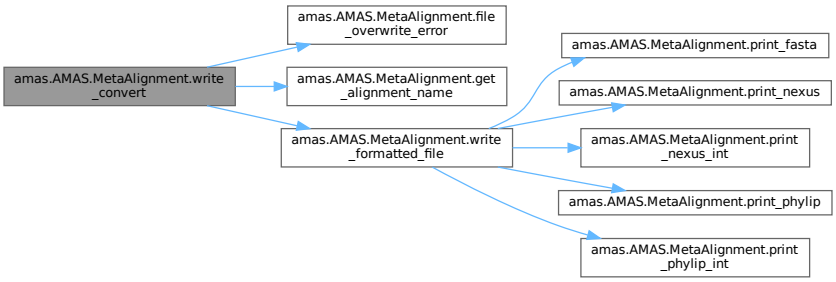
```
02197     def write_convert(self, index, alignment, file_format, extension):
02198         # write converted alignment into a file
```

```
02199         file_name = self.get_alignment_name(index, extension)
02200         self.file_overwrite_error(file_name)
02201         self.write_formatted_file(file_format, file_name, alignment)
02202
```

References amas.AMAS.MetaAlignment.file_overwrite_error(), amas.AMAS.MetaAlignment.get_alignment_name(), and amas.AMAS.MetaAlignment.write_formatted_file().

Referenced by amas.AMAS.MetaAlignment.write_out().

Here is the call graph for this function:



Here is the caller graph for this function:

7.6.3.39 write_formatted_file()

```
amas.AMAS.MetaAlignment.write_formatted_file (
              self,
              file_format,
              file_name,
              alignment )
```

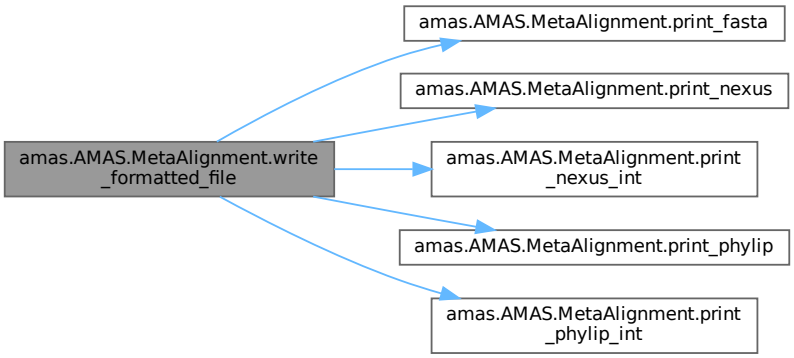
Definition at line 2162 of file AMAS.py.

```
02162     def write_formatted_file(self, file_format, file_name, alignment):
02163         # write the correct format string into a file
02164         with open(file_name, "w", encoding="utf-8") as out_file:
02165             if file_format == "phylip":
02166                 out_file.write(self.print_phylip(alignment))
02167             elif file_format == "fasta":
02168                 out_file.write(self.print_fasta(alignment))
02169             elif file_format == "phylip-int":
02170                 out_file.write(self.print_phylip_int(alignment))
02171             elif file_format == "nexus":
02172                 out_file.write(self.print_nexus(alignment))
02173             elif file_format == "nexus-int":
02174                 out_file.write(self.print_nexus_int(alignment))
02175
```

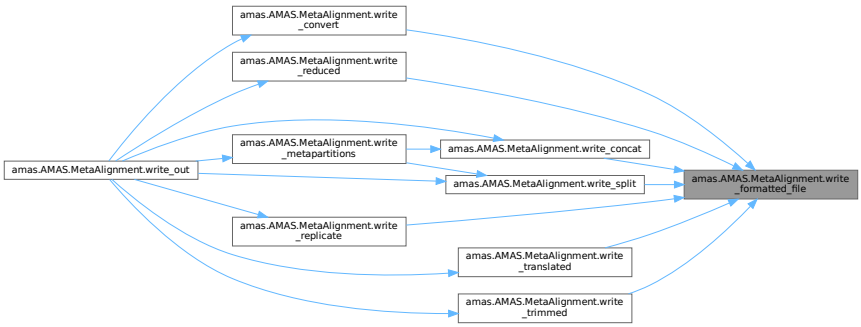
References amas.AMAS.MetaAlignment.print_fasta(), amas.AMAS.MetaAlignment.print_nexus(), amas.AMAS.MetaAlignment.print_r
amas.AMAS.MetaAlignment.print_phylip(), and amas.AMAS.MetaAlignment.print_phylip_int().

Referenced by amas.AMAS.MetaAlignment.write_concat(), amas.AMAS.MetaAlignment.write_convert(), amas.AMAS.MetaAlignment
amas.AMAS.MetaAlignment.write_replicate(), amas.AMAS.MetaAlignment.write_split(), amas.AMAS.MetaAlignment.write_translated
and amas.AMAS.MetaAlignment.write_trimmed().

Here is the call graph for this function:



Here is the caller graph for this function:



7.6.3.40 write_metapartitions()

```
amas.AMAS.MetaAlignment.write_metapartitions (
              self,
              file_format )
```

Definition at line 2263 of file AMAS.py.

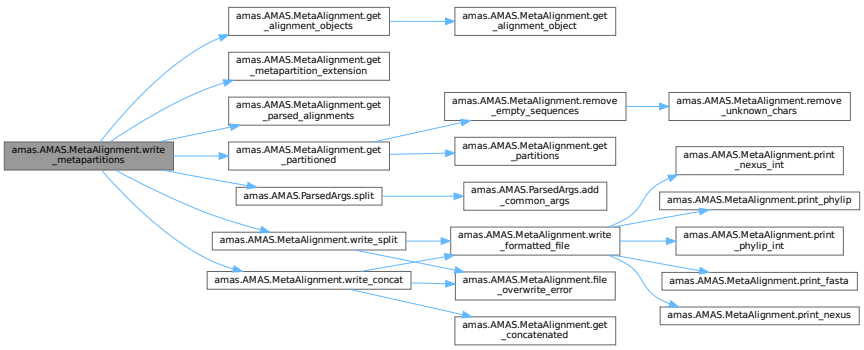
```
02263     def write_metapartitions(self, file_format):
02264         # write metapartitions - combines split and concat
02265         print("write_out elif action == metapartitions")
02266         metapartition_extension = self.get_metapartition_extension(file_format)
02267         list_of_alignments = self.get_partitioned(self.split)
02268         written_split_files = []
02269         err_indx = 0
```

```
02270
02271         for item in list_of_alignments:
02272             try:
02273                 for split_file in self.write_split(item, file_format, metapartition_extension):
02274                     written_split_files.append(split_file)
02275             except ValueError as e:
02276                     print("WARNING: ", e)
02277                     err_indx += 1
02278         if len(written_split_files) > 0:
02279             print("Wrote %d %s metapartition files from partitions provided" %
      (len(written_split_files), file_format))
02280         if err_indx > 0:
02281             print("WARNING: %d file(s) raised an error while writing (see above)." % err_indx)
02282
02283         # now set inputs to be the collated metapartition alignment files
02284         self.in_files = written_split_files
02285         self.alignment_objects = self.get_alignment_objects()
02286         self.parsed_alignments = self.get_parsed_alignments()
02287
02288         # concat metapartition alignment files
02289         self.write_concat(file_format)
02290
```

References amas.AMAS.MetaAlignment.alignment_objects, amas.AMAS.MetaAlignment.get_alignment_objects(), amas.AMAS.MetaAlignment.get_metapartition_extension(), amas.AMAS.MetaAlignment.get_parsed_alignments(), amas.AMAS.MetaAlignment.get_partitioned(), amas.AMAS.MetaAlignment.in_files, amas.AMAS.MetaAlignment.parsed_alignments, amas.AMAS.ParsedArgs.split(), amas.AMAS.MetaAlignment.split, amas.AMAS.MetaAlignment.write_concat(), and amas.AMAS.MetaAlignment.write_split().

Referenced by amas.AMAS.MetaAlignment.write_out().

Here is the call graph for this function:



Here is the caller graph for this function:

7.6.3.41 write_out()

```
amas.AMAS.MetaAlignment.write_out (
              self,
```

action,
file_format)

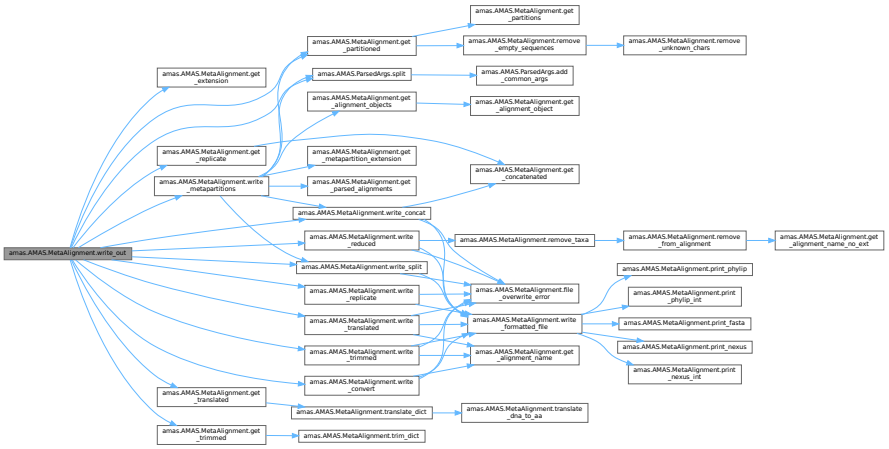
Definition at line 2291 of file AMAS.py.

```
02291     def write_out(self, action, file_format):
02292         # write other output files depending on command (action)
02293         extension = self.get_extension(file_format)
02294 
02295         if action == "concat":
02296             self.write_concat(file_format)
02297 
02298         elif action == "convert":
02299             length = len(self.alignment_objects)
02300             [
02301                 self.write_convert(i, alignment, file_format, extension)
02302                 for i, alignment in enumerate(self.parsed_alignments)
02303             ]
02304             print("Converted " + str(length) + " files from " + self.in_format + " to " + file_format)
02305 
02306         elif action == "replicate":
02307             [
02308                 self.write_replicate(i, alignment, file_format, extension)
02309                 for i, alignment in enumerate(self.get_replicate(self.no_replicates, self.no_loci))
02310             ]
02311 
02312             print("Constructed " + str(self.no_replicates) + " replicate data sets, each from " +
      str(self.no_loci) + " alignments")
02313 
02314         elif action == "split":
02315             list_of_alignments = self.get_partitioned(self.split)
02316             written_split_files = []
02317             err_indx = 0
02318 
02319             for item in list_of_alignments:
02320                 try:
02321                     for split_file in self.write_split(item, file_format, extension):
02322                         written_split_files.append(split_file)
02323                 except ValueError as e:
02324                         print("WARNING: ", e)
02325                         err_indx += 1
02326             if len(written_split_files) > 0:
02327                 print("Wrote %d %s files from partitions provided" % (len(written_split_files),
      file_format))
02328             if err_indx > 0:
02329                 print("WARNING: %d file(s) raised an error while writing (see above)." % err_indx)
02330 
02331         elif action == "metapartitions":
02332             self.write_metapartitions(file_format)
02333 
02334         elif action == "remove":
02335             aln_no = self.write_reduced(file_format, extension)
02336             if aln_no:
02337                 print("Wrote " + str(aln_no) + " " + str(file_format) + " files with reduced taxon
      set")
02338 
02339         elif action == "translate":
02340             if self.data_type == "aa":
02341                 print("ERROR: cannot translate; you said your alignment already contains amino acids")
02342                 sys.exit()
02343             translated_alignment_dicts = self.get_translated(self.genetic_code, self.reading_frame)
02344             length = len(self.alignment_objects)
02345             [
02346                 self.write_translated(i, alignment, file_format, extension)
02347                 for i, alignment in enumerate(translated_alignment_dicts)
02348             ]
02349             print("Translated " + str(length) + " files to amino acid sequences")
02350 
02351         elif action == "trim": # self.trim_fraction, self.parsimony_check
02352             trimmed_alignment_dicts = self.get_trimmed(self.trim_fraction, self.parsimony_check)
02353             length = len(self.alignment_objects)
02354             [
02355                 self.write_trimmed(i, alignment, file_format, extension)
02356                 for i, alignment in enumerate(trimmed_alignment_dicts)
02357             ]
02358             print("Trimmed", str(length), "file(s) to have", self.trim_fraction, "minimum occupancy
      per alignment column")
02359 
02360 
```

References amas.AMAS.MetaAlignment.alignment_objects, amas.AMAS.Alignment.data_type, amas.AMAS.MetaAlignment.data_typ
amas.AMAS.MetaAlignment.genetic_code, amas.AMAS.MetaAlignment.get_extension(), amas.AMAS.MetaAlignment.get_partitionec
amas.AMAS.MetaAlignment.get_replicate(), amas.AMAS.MetaAlignment.get_translated(), amas.AMAS.MetaAlignment.get_trimmed(
amas.AMAS.Alignment.in_format, amas.AMAS.MetaAlignment.in_format, amas.AMAS.MetaAlignment.no_loci,

amas.AMAS.MetaAlignment.no_replicates, amas.AMAS.MetaAlignment.parsed_alignments, amas.AMAS.MetaAlignment.parsimony_
amas.AMAS.MetaAlignment.reading_frame, amas.AMAS.ParsedArgs.split(), amas.AMAS.MetaAlignment.split,
amas.AMAS.MetaAlignment.trim_fraction, amas.AMAS.MetaAlignment.write_concat(), amas.AMAS.MetaAlignment.write_convert(),
amas.AMAS.MetaAlignment.write_metapartitions(), amas.AMAS.MetaAlignment.write_reduced(), amas.AMAS.MetaAlignment.write_
amas.AMAS.MetaAlignment.write_split(), amas.AMAS.MetaAlignment.write_translated(), and amas.AMAS.MetaAlignment.write_trim

Here is the call graph for this function:



7.6.3.42 write_partitions()

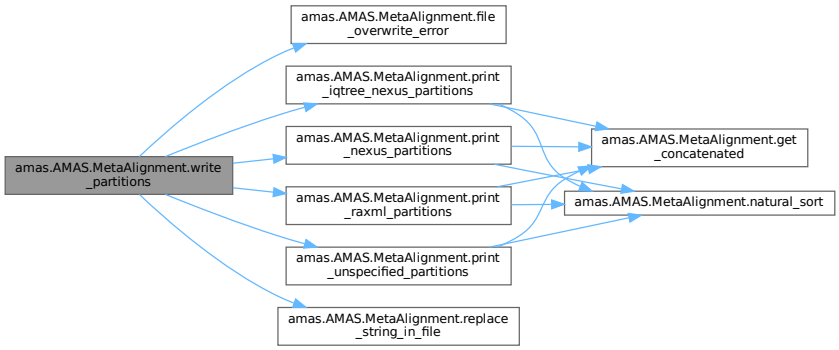
```
amas.AMAS.MetaAlignment.write_partitions (
              self,
              file_name,
              part_format,
              data_type,
              codons )
```

Definition at line 2109 of file AMAS.py.

```
02109     def write_partitions(self, file_name, part_format, data_type, codons):
02110         # write partitions file for concatenated alignment
02111         self.file_overwrite_error(file_name)
02112         with open(file_name, "w", encoding="utf-8") as part_file:
02113             if part_format == "nexus":
02114                 part_file.write(self.print_nexus_partitions(data_type, codons))
02115             if part_format == "iqtree-nexus":
02116                 part_file.write(self.print_iqtree_nexus_partitions(data_type, codons))
02117             if part_format == "raxml":
02118                 part_file.write(self.print_raxml_partitions(data_type, codons))
02119             if part_format == "unspecified":
02120                 part_file.write(self.print_unspecified_partitions(data_type, codons))
02121
02122             if self.using_metapartitions:
02123                 self.replace_string_in_file(file_name, '-meta =', ' =')
02124
02125         print("Wrote partitions for the concatenated file to '" + file_name + "'")
02126
```

References amas.AMAS.MetaAlignment.file_overwrite_error(), amas.AMAS.MetaAlignment.print_iqtree_nexus_partitions(), amas.AMAS.MetaAlignment.print_nexus_partitions(), amas.AMAS.MetaAlignment.print_raxml_partitions(), amas.AMAS.MetaAlignment.print_unspecified_partitions(), amas.AMAS.MetaAlignment.replace_string_in_file(), and amas.AMAS.MetaAlignment.using_metapartitions.

Here is the call graph for this function:



7.6.3.43 write_reduced()

```
amas.AMAS.MetaAlignment.write_reduced (
                self,
                file_format,
                extension )
```

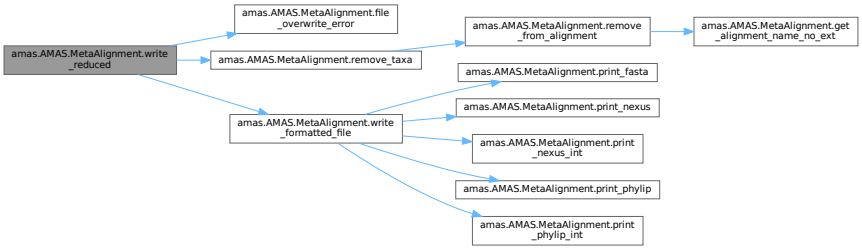
Definition at line 2234 of file AMAS.py.

```
02234     def write_reduced(self, file_format, extension):
02235         # write alignment with taxa removed into a file
02236         prefix = self.reduced_file_prefix
02237         alns = self.remove_taxa(self.species_to_remove)
02238         for file_name, aln_dict in alns.items():
02239             out_file_name = prefix + file_name + extension
02240             self.file_overwrite_error(out_file_name)
02241             self.write_formatted_file(file_format, out_file_name, aln_dict)
02242         return len(alns)
02243
```

References amas.AMAS.MetaAlignment.file_overwrite_error(), amas.AMAS.MetaAlignment.reduced_file_prefix, amas.AMAS.MetaAlignment.remove_taxa(), amas.AMAS.MetaAlignment.species_to_remove, and amas.AMAS.MetaAlignment.write_

Referenced by amas.AMAS.MetaAlignment.write_out().

Here is the call graph for this function:



Here is the caller graph for this function:

7.6.3.44 write_replicate()

```
amas.AMAS.MetaAlignment.write_replicate (
              self,
              index,
              alignment,
              file_format,
              extension )
```

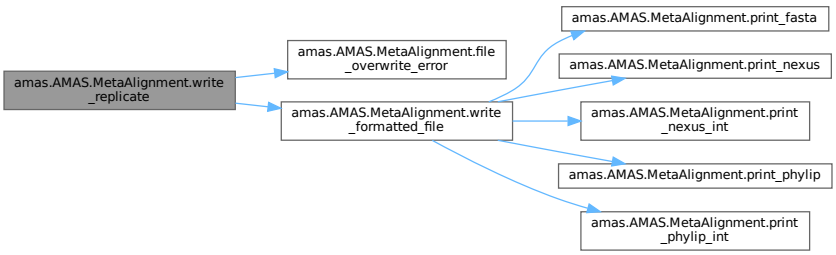
Definition at line 2203 of file AMAS.py.

```
02203     def write_replicate(self, index, alignment, file_format, extension):
02204         # write replicate alignment into a file
02205         file_name = "replicate" + str(index + 1) + "_" + str(self.no_loci) + "-loci" + extension
02206         self.file_overwrite_error(file_name)
02207         self.write_formatted_file(file_format, file_name, alignment)
02208
```

References amas.AMAS.MetaAlignment.file_overwrite_error(), amas.AMAS.MetaAlignment.no_loci, and amas.AMAS.MetaAlignment

Referenced by amas.AMAS.MetaAlignment.write_out().

Here is the call graph for this function:



Here is the caller graph for this function:

7.6.3.45 write_split()

```
amas.AMAS.MetaAlignment.write_split (
            self,
            item,
            file_format,
            extension )
```

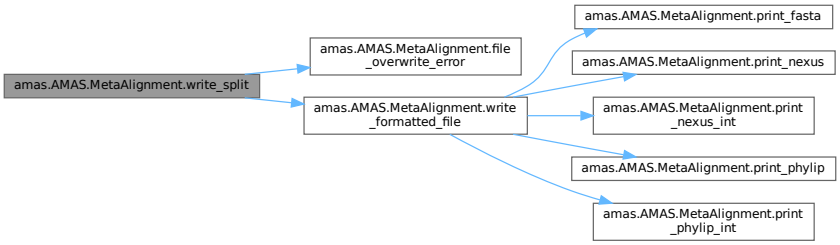
Definition at line 2209 of file AMAS.py.

```
02209     def write_split(self, item, file_format, extension):
02210         # write split alignments from partitions file
02211         # bad practice with the dicts; figure out better solution
02212         partition_name = list(item.keys())[0]
02213         alignment = item[partition_name]
02214 
02215         if not alignment:
02216             # If the alignment dict is empty, i.e. no alignment associated with partition name, raise
      error
02217             raise ValueError("Partition '%s' is empty. No sequences to write." % partition_name)
02218 
02219         # implementation of option --no-san (don't prepend input superalignment filename to the
      `split` outputs)
02220         if self.no_sup_aln_name:
02221             file_name = partition_name + extension
02222         else:
02223             file_name = str(self.in_files[0].split('.')[0]) + "_" + partition_name + extension
02224 
02225         try:
02226             self.file_overwrite_error(file_name)
02227             self.write_formatted_file(file_format, file_name, alignment)
02228             yield file_name
02229         except ValueError as e:
02230             print("There was an issue writing file '%s': %s" % (file_name, str(e)))
02231             remove(file_name)
02232             raise
02233 
```

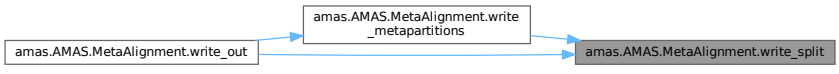
References amas.AMAS.MetaAlignment.file_overwrite_error(), amas.AMAS.MetaAlignment.in_files, amas.AMAS.MetaAlignment.no_
amas.AMAS.MetaAlignment.split, and amas.AMAS.MetaAlignment.write_formatted_file().

Referenced by amas.AMAS.MetaAlignment.write_metapartitions(), and amas.AMAS.MetaAlignment.write_out().

Here is the call graph for this function:



Here is the caller graph for this function:



7.6.3.46 write_summaries()

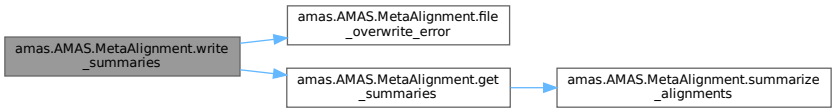
```
amas.AMAS.MetaAlignment.write_summaries (
              self,
              file_name )
```

Definition at line 1698 of file AMAS.py.

```
01698     def write_summaries(self, file_name):
01699         # write summaries to file
01700
01701         self.file_overwrite_error(file_name)
01702
01703         with open(file_name, "w", encoding="utf-8") as summary_file:
01704             summary_out = self.get_summaries()
01705             header = '\t'.join(summary_out[0])
01706             new_summ = ['\t'.join(summary) for summary in summary_out[1]]
01707             summary_file.write(header + '\n')
01708             summary_file.write('\n'.join(new_summ))
01709             summary_file.write('\n')
01710             print("Wrote summaries to file '" + file_name + "'")
01711
```

References amas.AMAS.MetaAlignment.file_overwrite_error(), and amas.AMAS.MetaAlignment.get_summaries().

Here is the call graph for this function:



7.6.3.47 write_taxa_summaries()

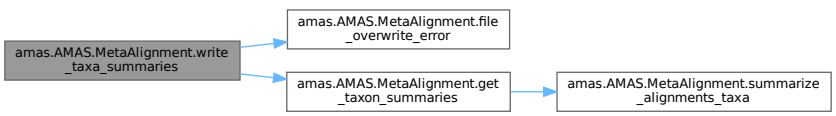
```
amas.AMAS.MetaAlignment.write_taxa_summaries (
              self )
```

Definition at line 1712 of file AMAS.py.

```
01712     def write_taxa_summaries(self):
01713         # write by-taxon summaries to file
01714         for index, in_file_name in enumerate(self.in_files):
01715             out_file_name = in_file_name + "-seq-summary.txt"
01716             self.file_overwrite_error(out_file_name)
01717             with open(out_file_name, "w", encoding="utf-8") as summary_file:
01718                 summary_out = self.get_taxon_summaries()
01719                 header = '\t'.join(summary_out[0])
01720                 summ = [[str(col) for col in element] for element in summary_out[1][index]]
01721                 new_summ = ['\t'.join(row) for row in summ]
01722                 summary_file.write(header + '\n')
01723                 summary_file.write('\n'.join(new_summ))
01724                 summary_file.write('\n')
01725
```

References amas.AMAS.MetaAlignment.file_overwrite_error(), amas.AMAS.MetaAlignment.get_taxon_summaries(), and amas.AMAS.MetaAlignment.in_files.

Here is the call graph for this function:



7.6.3.48 write_translated()

```
amas.AMAS.MetaAlignment.write_translated (
                self,
                index,
                alignment,
                file_format,
                extension )
```

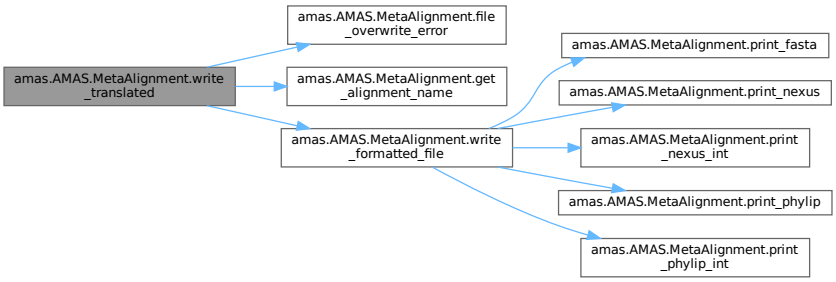
Definition at line 2244 of file AMAS.py.

```
02244     def write_translated(self, index, alignment, file_format, extension):
02245         # write alignments translated into amino acids
02246         prefix = "translated_"
02247         file_name = self.get_alignment_name(index, extension)
02248         out_file_name = prefix + file_name + extension
02249         self.file_overwrite_error(out_file_name)
02250         self.write_formatted_file(file_format, out_file_name, alignment)
02251
```

References amas.AMAS.MetaAlignment.file_overwrite_error(), amas.AMAS.MetaAlignment.get_alignment_name(), and amas.AMAS.MetaAlignment.write_formatted_file().

Referenced by amas.AMAS.MetaAlignment.write_out().

Here is the call graph for this function:



Here is the caller graph for this function:

7.6.3.49 write_trimmed()

```
amas.AMAS.MetaAlignment.write_trimmed (
                self,
                index,
                alignment,
```

```
            file_format,
            extension )
```

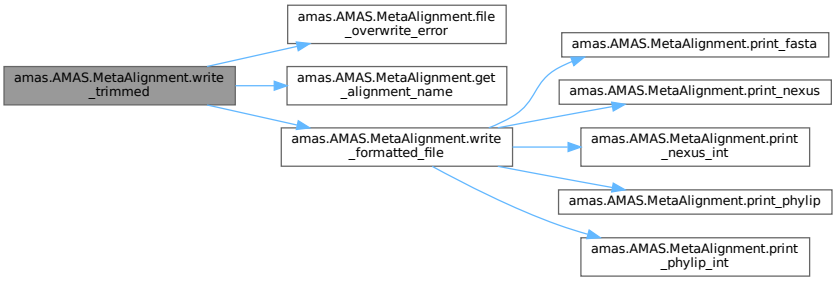
Definition at line 2252 of file AMAS.py.

```
02252     def write_trimmed(self, index, alignment, file_format, extension):
02253         # write trimmed alignments
02254         if self.trim_out:
02255             out_file_name = self.trim_out
02256         else:
02257             prefix = "trimmed_"
02258             file_name = self.get_alignment_name(index, extension)
02259             out_file_name = prefix + file_name
02260         self.file_overwrite_error(out_file_name)
02261         self.write_formatted_file(file_format, out_file_name, alignment)
02262
```

References amas.AMAS.MetaAlignment.file_overwrite_error(), amas.AMAS.MetaAlignment.get_alignment_name(), amas.AMAS.MetaAlignment.trim_out, and amas.AMAS.MetaAlignment.write_formatted_file().

Referenced by amas.AMAS.MetaAlignment.write_out().

Here is the call graph for this function:



Here is the caller graph for this function:

7.6.4 Member Data Documentation

7.6.4.1 alignment_objects

`amas.AMAS.MetaAlignment.alignment_objects`

Definition at line 1224 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_alignment_name(), amas.AMAS.MetaAlignment.get_alignment_name_no_ext(), amas.AMAS.MetaAlignment.get_concatenated(), amas.AMAS.MetaAlignment.get_parsed_alignments(), amas.AMAS.MetaAlignment amas.AMAS.MetaAlignment.get_taxon_summaries(), amas.AMAS.MetaAlignment.get_trimmed(), amas.AMAS.MetaAlignment.write_ and amas.AMAS.MetaAlignment.write_out().

7.6.4.2 by_taxon_summary

amas.AMAS.MetaAlignment.by_taxon_summary

Definition at line 1178 of file AMAS.py.

7.6.4.3 check_align

amas.AMAS.MetaAlignment.check_align

Definition at line 1176 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_parsed_alignments().

7.6.4.4 check_taxa

amas.AMAS.MetaAlignment.check_taxa

Definition at line 1213 of file AMAS.py.

7.6.4.5 codes

amas.AMAS.MetaAlignment.codes

Definition at line 1424 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.translate_dict().

7.6.4.6 codes_list

amas.AMAS.MetaAlignment.codes_list

Definition at line 1228 of file AMAS.py.

7.6.4.7 codons

amas.AMAS.MetaAlignment.codons

Definition at line 1183 of file AMAS.py.

7.6.4.8 command

amas.AMAS.MetaAlignment.command

Definition at line 1173 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.print_nexus().

7.6.4.9 concat_out

amas.AMAS.MetaAlignment.concat_out

Definition at line 1174 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.write_concat().

7.6.4.10 cores

amas.AMAS.MetaAlignment.cores

Definition at line 1177 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_alignment_objects(), amas.AMAS.MetaAlignment.get_summaries(), amas.AMAS.MetaAlignment.get_taxon_summaries(), amas.AMAS.MetaAlignment.get_translated(), and amas.AMAS.MetaAlignment.

7.6.4.11 data_type

amas.AMAS.MetaAlignment.data_type

Definition at line 1172 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_alignment_object(), amas.AMAS.MetaAlignment.get_summaries(), amas.AMAS.MetaAlignment.get_taxon_summaries(), amas.AMAS.MetaAlignment.print_nexus(), amas.AMAS.MetaAlignment.print_n and amas.AMAS.MetaAlignment.write_out().

7.6.4.12 gencode_NCBI_1

amas.AMAS.MetaAlignment.gencode_NCBI_1

Definition at line 1251 of file AMAS.py.

7.6.4.13 gencode_NCBI_10

amas.AMAS.MetaAlignment.gencode_NCBI_10

Definition at line 1364 of file AMAS.py.

7.6.4.14 gencode_NCBI_11

amas.AMAS.MetaAlignment.gencode_NCBI_11

Definition at line 1368 of file AMAS.py.

7.6.4.15 gencode_NCBI_12

amas.AMAS.MetaAlignment.gencode_NCBI_12

Definition at line 1371 of file AMAS.py.

7.6.4.16 gencode_NCBI_13

amas.AMAS.MetaAlignment.gencode_NCBI_13

Definition at line 1375 of file AMAS.py.

7.6.4.17 gencode_NCBI_14

amas.AMAS.MetaAlignment.gencode_NCBI_14

Definition at line 1382 of file AMAS.py.

7.6.4.18 gencode_NCBI_16

amas.AMAS.MetaAlignment.gencode_NCBI_16

Definition at line 1390 of file AMAS.py.

7.6.4.19 gencode_NCBI_2

amas.AMAS.MetaAlignment.gencode_NCBI_2

Definition at line 1322 of file AMAS.py.

7.6.4.20 gencode_NCBI_21

amas.AMAS.MetaAlignment.gencode_NCBI_21

Definition at line 1394 of file AMAS.py.

7.6.4.21 gencode_NCBI_22

amas.AMAS.MetaAlignment.gencode_NCBI_22

Definition at line 1402 of file AMAS.py.

7.6.4.22 gencode_NCBI_23

amas.AMAS.MetaAlignment.gencode_NCBI_23

Definition at line 1407 of file AMAS.py.

7.6.4.23 gencode_NCBI_24

amas.AMAS.MetaAlignment.gencode_NCBI_24

Definition at line 1411 of file AMAS.py.

7.6.4.24 gencode_NCBI_25

amas.AMAS.MetaAlignment.gencode_NCBI_25

Definition at line 1417 of file AMAS.py.

7.6.4.25 gencode_NCBI_26

amas.AMAS.MetaAlignment.gencode_NCBI_26

Definition at line 1421 of file AMAS.py.

7.6.4.26 gencode_NCBI_3

amas.AMAS.MetaAlignment.gencode_NCBI_3

Definition at line 1329 of file AMAS.py.

7.6.4.27 gencode_NCBI_4

amas.AMAS.MetaAlignment.gencode_NCBI_4

Definition at line 1341 of file AMAS.py.

7.6.4.28 gencode_NCBI_5

amas.AMAS.MetaAlignment.gencode_NCBI_5

Definition at line 1345 of file AMAS.py.

7.6.4.29 gencode_NCBI_6

amas.AMAS.MetaAlignment.gencode_NCBI_6

Definition at line 1352 of file AMAS.py.

7.6.4.30 gencode_NCBI_9

amas.AMAS.MetaAlignment.gencode_NCBI_9

Definition at line 1357 of file AMAS.py.

7.6.4.31 genetic_code

`amas.AMAS.MetaAlignment.genetic_code`

Definition at line 1217 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.translate_dict(), and amas.AMAS.MetaAlignment.write_out().

7.6.4.32 in_files

`amas.AMAS.MetaAlignment.in_files`

Definition at line 1170 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_alignment_objects(), amas.AMAS.MetaAlignment.write_metapartitions(), amas.AMAS.MetaAlignment.write_split(), and amas.AMAS.MetaAlignment.write_taxa_summaries().

7.6.4.33 in_format

`amas.AMAS.MetaAlignment.in_format`

Definition at line 1171 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_alignment_object(), amas.AMAS.Alignment.get_parsed_aln(), and amas.AMAS.MetaAlignment.write_out().

7.6.4.34 no_loci

`amas.AMAS.MetaAlignment.no_loci`

Definition at line 1190 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.write_out(), and amas.AMAS.MetaAlignment.write_replicate().

7.6.4.35 no_mpan

`amas.AMAS.MetaAlignment.no_mpan`

Definition at line 1180 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_concatenated().

7.6.4.36 no_replicates

`amas.AMAS.MetaAlignment.no_replicates`

Definition at line 1189 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.write_out().

7.6.4.37 no_sup_aln_name

`amas.AMAS.MetaAlignment.no_sup_aln_name`

Definition at line 1179 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.write_split().

7.6.4.38 parsed_alignments

`amas.AMAS.MetaAlignment.parsed_alignments`

Definition at line 1225 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_partitioned(), amas.AMAS.MetaAlignment.get_replicate(), amas.AMAS.MetaAlignment.get_summaries(), amas.AMAS.MetaAlignment.get_taxon_summaries(), amas.AMAS.MetaAlignment.get amas.AMAS.MetaAlignment.print_iqtree_nexus_partitions(), amas.AMAS.MetaAlignment.print_nexus_partitions(), amas.AMAS.MetaAlignment.print_raxml_partitions(), amas.AMAS.MetaAlignment.print_unspecified_partitions(), amas.AMAS.MetaAlignment.remove_taxa(), amas.AMAS.MetaAlignment.write_concat(), amas.AMAS.MetaAlignment.write_metapart and amas.AMAS.MetaAlignment.write_out().

7.6.4.39 parsimony_check

`amas.AMAS.MetaAlignment.parsimony_check`

Definition at line 1222 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.trim_dict(), and amas.AMAS.MetaAlignment.write_out().

7.6.4.40 prepend_label

`amas.AMAS.MetaAlignment.prepend_label`

Definition at line 1203 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_concatenated().

7.6.4.41 reading_frame

`amas.AMAS.MetaAlignment.reading_frame`

Definition at line 1216 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.translate_dict(), and amas.AMAS.MetaAlignment.write_out().

7.6.4.42 reduced_file_prefix

`amas.AMAS.MetaAlignment.reduced_file_prefix`

Definition at line 1212 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.write_reduced().

7.6.4.43 remove_empty

amas.AMAS.MetaAlignment.remove_empty

Definition at line 1194 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_partitioned().

7.6.4.44 species_to_remove

amas.AMAS.MetaAlignment.species_to_remove

Definition at line 1210 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.write_reduced().

7.6.4.45 species_to_remove_set

amas.AMAS.MetaAlignment.species_to_remove_set

Definition at line 1211 of file AMAS.py.

7.6.4.46 split

amas.AMAS.MetaAlignment.split

Definition at line 1193 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_concatenated(), amas.AMAS.MetaAlignment.print_iqtree_nexus_partitions(), amas.AMAS.MetaAlignment.print_nexus_partitions(), amas.AMAS.MetaAlignment.print_raxml_partitions(), amas.AMAS.MetaAlignment.print_unspecified_partitions(), amas.AMAS.MetaAlignment.write_metapartitions(), amas.AMAS.MetaAlignment.write_out(), and amas.AMAS.MetaAlignment.write_split().

7.6.4.47 trim_fraction

amas.AMAS.MetaAlignment.trim_fraction

Definition at line 1220 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.trim_dict(), and amas.AMAS.MetaAlignment.write_out().

7.6.4.48 trim_out

amas.AMAS.MetaAlignment.trim_out

Definition at line 1221 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.write_trimmed().

7.6.4.49 using_metapartitions

`amas.AMAS.MetaAlignment.using_metapartitions`

Definition at line 1175 of file AMAS.py.

Referenced by amas.AMAS.MetaAlignment.get_concatenated(), and amas.AMAS.MetaAlignment.write_partitions().

The documentation for this class was generated from the following file:

- amas/AMAS.py

7.7 amas.AMAS.ParsedArgs Class Reference

Public Member Functions

- __init__ (self)
- add_common_args (self, parser)
- trim (self)
- summary (self)
- concat (self)
- convert (self)
- replicate (self)
- split (self)
- metapartitions (self)
- translate (self)
- remove (self)
- get_args_dict (self)

Public Attributes

- args

7.7.1 Detailed Description

Definition at line 50 of file AMAS.py.

7.7.2 Constructor & Destructor Documentation

7.7.2.1 __init__()

```
amas.AMAS.ParsedArgs.__init__ (
              self )
```

Definition at line 52 of file AMAS.py.

```
00052     def __init__(self):
00053         parser = argparse.ArgumentParser(
00054             usage="'AMAS <command> [<args>]
00055
00056 The AMAS commands are:
00057    concat            Concatenate input alignments.
00058    convert           Convert to other file format.
00059    replicate         Create replicate data sets for phylogenetic jackknife.
00060    split             Split alignment according to a partitions file.
00061    summary           Write alignment summary.
00062    remove            Remove taxa from alignment.
00063    translate         Translate DNA alignment into protein alignment.
00064    trim              Remove columns from alignment.
00065    metapartitions    Runs `split` and concatenates the output.
00066
00067
00068 Use AMAS <command> -h for help with arguments of the command of interest
00069 "'
00070         )
00071
00072         parser.add_argument(
00073             "command",
00074             help="Subcommand to run"
00075         )
00076
00077         # parse_args defaults to [1:] for args, but you need to
00078         # exclude the rest of the args too, or validation will fail
00079         self.args = parser.parse_args(sys.argv[1:2])
00080         if not hasattr(self, self.args.command):
00081             print('Unrecognized command')
00082             parser.print_help()
00083             exit(1)
00084         # use dispatch pattern to invoke method with same name
00085         getattr(self, self.args.command)()
00086
```

7.7.3 Member Function Documentation

7.7.3.1 add_common_args()

```
amas.AMAS.ParsedArgs.add_common_args (
              self,
              parser )
```

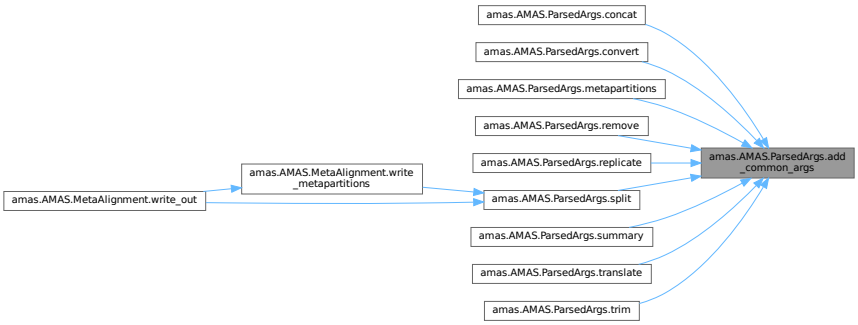
Definition at line 87 of file AMAS.py.

```
00087     def add_common_args(self, parser):
00088         # define required arguments for every command
00089         requiredNamed = parser.add_argument_group('required arguments')
00090         parser.add_argument(
00091             "-e",
00092             "--check-align",
00093             dest = "check_align",
00094             action = "store_true",
00095             default = False,
00096             help = "Check if input sequences are aligned. Default: no check"
00097         )
00098         parser.add_argument(
00099             # parallelization is used for file parsing and calculating summary stats
00100             "-c",
00101             "--cores",
00102             dest = "cores",
00103             default = 1,
00104             help = "Number of cores used. Default: 1"
00105         )
00106
```

```
00107         requiredNamed.add_argument(
00108             "-i",
00109             "--in-files",
00110             nargs = "+",
00111             type = str,
00112             dest = "in_files",
00113             required = True,
00114             help = """Alignment files to be taken as input.
00115             You can specify multiple files using wildcards (e.g. --in-files *fasta)"""
00116         )
00117         requiredNamed.add_argument(
00118             "-f",
00119             "--in-format",
00120             dest = "in_format",
00121             required = True,
00122             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00123             help = "The format of input alignment"
00124         )
00125         requiredNamed.add_argument(
00126             "-d",
00127             "--data-type",
00128             dest = "data_type",
00129             required = True,
00130             choices = ["aa", "dna"],
00131             help = "Type of data"
00132         )
00133
```

Referenced by amas.AMAS.ParsedArgs.concat(), amas.AMAS.ParsedArgs.convert(), amas.AMAS.ParsedArgs.metapartitions(), amas.AMAS.ParsedArgs.remove(), amas.AMAS.ParsedArgs.replicate(), amas.AMAS.ParsedArgs.split(), amas.AMAS.ParsedArgs.su amas.AMAS.ParsedArgs.translate(), and amas.AMAS.ParsedArgs.trim().

Here is the caller graph for this function:



7.7.3.2 concat()

```
amas.AMAS.ParsedArgs.concat (
              self )
```

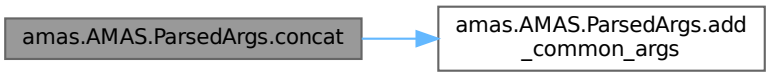
Definition at line 201 of file AMAS.py.

```
00201     def concat(self):
00202         # concat command
00203         parser = argparse.ArgumentParser(
00204             description="Concatenate input alignments"
00205         )
00206         parser.add_argument(
00207             "-p",
00208             "--concat-part",
00209             dest = "concat_part",
00210             default = "partitions.txt",
00211             help = "File name for the concatenated alignment partitions. Default: 'partitions.txt'"
00212         )
00213         parser.add_argument(
00214             "-t",
00215             "--concat-out",
```

```
00216             dest = "concat_out",
00217             default = "concatenated.out",
00218             help = "File name for the concatenated alignment. Default: 'concatenated.out'"
00219         )
00220         parser.add_argument(
00221             "-u",
00222             "--out-format",
00223             dest = "out_format",
00224             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00225             default = "fasta",
00226             help = "File format for the output alignment. Default: fasta"
00227         )
00228         parser.add_argument(
00229             "-y",
00230             "--part-format",
00231             dest = "part_format",
00232             choices = ["nexus", "iqtree-nexus", "raxml", "unspecified"],
00233             default = "unspecified",
00234             help = "Format of the partitions file. Default: 'unspecified'"
00235         )
00236         parser.add_argument(
00237             "-n",
00238             "--codons",
00239             dest = "codons",
00240             choices = ["none", "12", "123"],
00241             default = "none",
00242             help = "Use codon partitioning for 1st and 2nd or all three positions. Default: Don't use"
00243         )
00244         # add shared arguments
00245         self.add_common_args(parser)
00246         args = parser.parse_args(sys.argv[2:])
00247         return args
00248
```

References amas.AMAS.ParsedArgs.add_common_args().

Here is the call graph for this function:



7.7.3.3 convert()

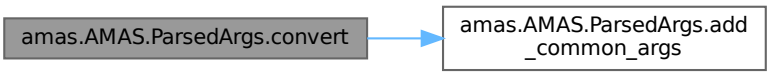
```
amas.AMAS.ParsedArgs.convert (
              self )
```

Definition at line 249 of file AMAS.py.

```
00249     def convert(self):
00250         # convert command
00251         parser = argparse.ArgumentParser(
00252             description="Convert to other file format",
00253         )
00254         parser.add_argument(
00255             "-u",
00256             "--out-format",
00257             dest = "out_format",
00258             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00259             default = "fasta",
00260             help = "File format for the output alignment. Default: fasta"
00261         )
00262         # add shared arguments
00263         self.add_common_args(parser)
00264         args = parser.parse_args(sys.argv[2:])
00265         return args
00266
```

References amas.AMAS.ParsedArgs.add_common_args().

Here is the call graph for this function:



7.7.3.4 get_args_dict()

```
amas.AMAS.ParsedArgs.get_args_dict (
              self )
```

Definition at line 508 of file AMAS.py.

```
00508     def get_args_dict(self):
00509         # store arguments in a dictionary
00510         command = self.args.__dict__
00511         arguments = getattr(self, self.args.command)().__dict__
00512         argument_dictionary = command.copy()
00513         argument_dictionary.update(arguments)
00514 
00515         return argument_dictionary
00516 
00517 
```

References amas.AMAS.ParsedArgs.args.

7.7.3.5 metapartitions()

```
amas.AMAS.ParsedArgs.metapartitions (
              self )
```

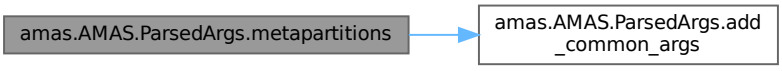
Definition at line 334 of file AMAS.py.

```
00334     def metapartitions(self):
00335         # metapartitions command
00336         parser = argparse.ArgumentParser(
00337             formatter_class=argparse.RawTextHelpFormatter,
00338             description="'Split alignment according to a partition file, then concatenate the
      output."'
00339             "'\n\nuse case:\n"'
00340             "'    Some utilities cannot parse partition definitions containing strides (\\) and/or
      discontinuous ranges.\n"'
00341             "'    In such case, running `split` + `concat` in separate passes can convert a
      corresponding (super)alignment it into an\n"'
00342             "'    equivalent compatible form with contiguous (meta)partitions; this may also require
      renaming metapartiton alignments\n"'
00343             "'    and partition file entries in order to remove tags applied by each respective
      operation.\n\n"'
00344             "'    `metapartitions` combines these steps into one command, with the options `--prepend`
      and `--no-mpan`\n"'
00345             "'    providing additional control over the collated (meta)partition names (see their
      respective help entries).\n\n"'
00346             "'    Note: in this mode, the format of the input (super)alignment file determines that of
      all outputs (-u|--out-format is disabled)!\n\n"'
00347         )
00348         parser.add_argument(
00349             "-p",
00350             "--concat-part",
00351             dest = "concat_part",
00352             default = "metapartitions.txt",
00353             help = "Partition file(name) for the final concatenated alignment of metapartitions.
      Default: 'metapartitions.txt'"
```

```
00354         )
00355         parser.add_argument(
00356             "-t",
00357             "--concat-out",
00358             dest = "concat_out",
00359             default = "concatenated-meta.out",
00360             help = "File name for the concatenated alignment of metapartitions. Default:
      'concatenated-meta.out'"
00361         )
00362         parser.add_argument(
00363             "-y",
00364             "--part-format",
00365             dest = "part_format",
00366             choices = ["nexus", "iqtree-nexus", "raxml", "unspecified"],
00367             default = "unspecified",
00368             help = "Partitions file format for the final concatenated alignment of metapartitions.
      Default: 'unspecified'"
00369         )
00370         parser.add_argument(
00371             "-l",
00372             "--split-by",
00373             dest = "split_by",
00374             help = "Partition file(name) to be used for splitting the initial concatenated
      alignment.",
00375             required = True
00376         )
00377         parser.add_argument(
00378             "-j",
00379             "--remove-empty",
00380             dest = "remove_empty",
00381             action = "store_true",
00382             default = False,
00383             help = "Remove taxa with sequences composed of only undetermined characters? Default:
      Don't remove"
00384         )
00385         parser.add_argument(
00386             "--no-san",
00387             dest = "no_sup_aln_name",
00388             action = "store_true",
00389             default = False,
00390             help = "'Don't prepend the input (super)alignment filename to the
      (meta)partition-alignment filenames output by `split`"'
00391         )
00392         parser.add_argument(
00393             "--prepend",
00394             dest = "prepend_label",
00395             default = None,
00396             help = "'Prepend <string> to the partition counter in partition file, e.g."'
00397             "'\n            --prepend <string>:  <string>_p001_metapartition_alignment_name = 1-1200
      ..."'
00398             "'\n               Default (None):           p001_metapartition_alignment_name = 1-1200
      ..."'
00399             "'\n--no-mpan + --prepend <string>:  <string>_p001 = 1-1200 ..."'
00400         )
00401         parser.add_argument(
00402             "--no-mpan",
00403             dest = "no_mpan",
00404             action = "store_true",
00405             default = False,
00406             help = "'Omits (meta)partition alignment names when printing partition file, e.g."'
00407             "'\n                      --no-mpan:          p001 = 1-1200 ..."'
00408             "'\n                Default (False):          p001_metapartition_alignment_name = 1-1200
      ..."'
00409             "'\n--prepend <string> + --no-mpan: <string>_p001 = 1-1200 ..."'
00410         )
00411         # add shared arguments
00412         self.add_common_args(parser)
00413         args = parser.parse_args(sys.argv[2:])
00414         return args
00415
```

References amas.AMAS.ParsedArgs.add_common_args().

Here is the call graph for this function:



7.7.3.6 remove()

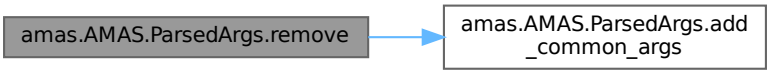
```
amas.AMAS.ParsedArgs.remove (
              self )
```

Definition at line 474 of file AMAS.py.

```
00474     def remove(self):
00475         # remove taxa command
00476         parser = argparse.ArgumentParser(
00477             description="Remove taxa from alignment",
00478         )
00479         parser.add_argument(
00480             "-x",
00481             "--taxa-to-remove",
00482             nargs = "+",
00483             type = str,
00484             dest = "taxa_to_remove",
00485             help = "Taxon/sequence names to be removed.",
00486             required = True
00487         )
00488         parser.add_argument(
00489             "-u",
00490             "--out-format",
00491             dest = "out_format",
00492             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00493             default = "fasta",
00494             help = "File format for the output alignment. Default: fasta"
00495         )
00496         parser.add_argument(
00497             "-g",
00498             "--out-prefix",
00499             dest = "out_prefix",
00500             default = "reduced_",
00501             help = "File name prefix for the concatenated alignment. Default: 'reduced_'"
00502         )
00503         # add shared arguments
00504         self.add_common_args(parser)
00505         args = parser.parse_args(sys.argv[2:])
00506         return args
00507
```

References amas.AMAS.ParsedArgs.add_common_args().

Here is the call graph for this function:



7.7.3.7 replicate()

```
amas.AMAS.ParsedArgs.replicate (
              self )
```

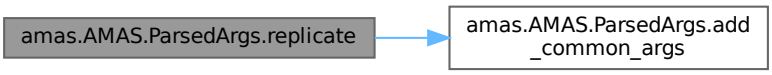
Definition at line 267 of file AMAS.py.

```
00267     def replicate(self):
00268         # replicate command
00269         parser = argparse.ArgumentParser(
00270             description="Create replicate datasets for phylogenetic jackknife",
00271         )
00272         parser.add_argument(
00273             "-r",
00274             "--rep-aln",
00275             nargs = 2,
00276             type = int,
```

```
00277             dest = "replicate_args",
00278             help = "Create replicate data sets for phylogenetic jackknife [replicates, no alignments
      for each replicate]",
00279             required = True
00280         )
00281         parser.add_argument(
00282             "-u",
00283             "--out-format",
00284             dest = "out_format",
00285             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00286             default = "fasta",
00287             help = "File format for the output alignment. Default: fasta"
00288         )
00289         # add shared arguments
00290         self.add_common_args(parser)
00291         args = parser.parse_args(sys.argv[2:])
00292         return args
00293
```

References amas.AMAS.ParsedArgs.add_common_args().

Here is the call graph for this function:



7.7.3.8 split()

```
amas.AMAS.ParsedArgs.split (
               self )
```

Definition at line 294 of file AMAS.py.

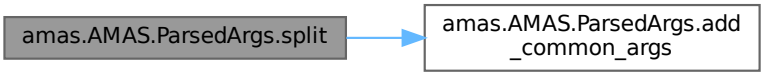
```
00294     def split(self):
00295         # split command
00296         parser = argparse.ArgumentParser(
00297             description="Split alignment according to a partitions file",
00298         )
00299         parser.add_argument(
00300             "-l",
00301             "--split-by",
00302             dest = "split_by",
00303             help = "File name for partitions to be used for alignment splitting.",
00304             required = True
00305         )
00306         parser.add_argument(
00307             "-j",
00308             "--remove-empty",
00309             dest = "remove_empty",
00310             action = "store_true",
00311             default = False,
00312             help = "Remove taxa with sequences composed of only undetermined characters? Default:
      Don't remove"
00313         )
00314         parser.add_argument(
00315             "-u",
00316             "--out-format",
00317             dest = "out_format",
00318             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00319             default = "fasta",
00320             help = "File format for the output alignment. Default: fasta"
00321         )
00322         parser.add_argument(
00323             "--no-san",
00324             dest = "no_sup_aln_name",
00325             action = "store_true",
00326             default = False,
```

```
00327             help = "'Don't prepend the input (super)alignment filename to the partition-alignment
      filenames output by `split`"'
00328         )
00329         # add shared arguments
00330         self.add_common_args(parser)
00331         args = parser.parse_args(sys.argv[2:])
00332         return args
00333
```

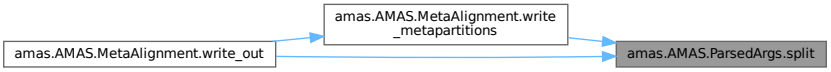
References amas.AMAS.ParsedArgs.add_common_args().

Referenced by amas.AMAS.MetaAlignment.write_metapartitions(), and amas.AMAS.MetaAlignment.write_out().

Here is the call graph for this function:



Here is the caller graph for this function:



7.7.3.9 summary()

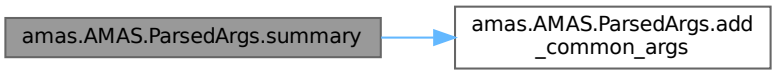
```
amas.AMAS.ParsedArgs.summary (
              self )
```

Definition at line 176 of file AMAS.py.

```
00176     def summary(self):
00177         # summary command
00178         parser = argparse.ArgumentParser(
00179             description="Write alignment summary",
00180         )
00181         parser.add_argument(
00182             "-o",
00183             "--summary-out",
00184             dest = "summary_out",
00185             default = "summary.txt",
00186             help = "File name for the alignment summary. Default: 'summary.txt'"
00187         )
00188         parser.add_argument(
00189             "-s",
00190             "--by-taxon",
00191             dest = "by_taxon_summary",
00192             action = "store_true",
00193             default = False,
00194             help = "In addition to alignment summary, write by sequence/taxon summaries. Default:
      Don't write"
00195         )
00196         # add shared arguments
00197         self.add_common_args(parser)
00198         args = parser.parse_args(sys.argv[2:])
00199         return args
00200
```

References amas.AMAS.ParsedArgs.add_common_args().

Here is the call graph for this function:



7.7.3.10 translate()

```
amas.AMAS.ParsedArgs.translate (
              self )
```

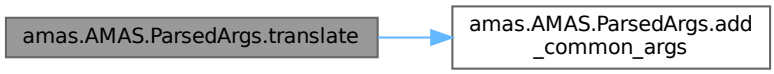
Definition at line 416 of file AMAS.py.

```
00416     def translate(self):
00417         # translate command
00418         parser = argparse.ArgumentParser(
00419             formatter_class=argparse.RawTextHelpFormatter,
00420             description="Translate a protein-coding DNA alignment into amino acids"
00421         )
00422         parser.add_argument(
00423             "-b",
00424             "--code",
00425             type = int,
00426             dest = "genetic_code",
00427             choices = [1, 2, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 16, 21, 22, 23, 24, 25, 26],
00428             default = 1,
00429             help = "'\nNCBI genetic code to use (Default: 1):"'
00430 "'
00431    1. The Standard Code
00432    2. The Vertebrate Mitochondrial Code
00433    3. The Yeast Mitochondrial Code
00434    4. The Mold, Protozoan, and Coelenterate Mitochondrial Code and the Mycoplasma/Spiroplasma Code
00435    5. The Invertebrate Mitochondrial Code
00436    6. The Ciliate, Dasycladacean and Hexamita Nuclear Code
00437    9. The Echinoderm and Flatworm Mitochondrial Code
00438   10. The Euplotid Nuclear Code
00439   11. The Bacterial, Archaeal and Plant Plastid Code
00440   12. The Alternative Yeast Nuclear Code
00441   13. The Ascidian Mitochondrial Code
00442   14. The Alternative Flatworm Mitochondrial Code
00443   16. Chlorophycean Mitochondrial Code
00444   21. Trematode Mitochondrial Code
00445   22. Scenedesmus obliquus Mitochondrial Code
00446   23. Thraustochytrium Mitochondrial Code
00447   24. Pterobranchia Mitochondrial Code
00448   25. Candidate Division SR1 and Gracilibacteria Code
00449   26. Pachysolen tannophilus Nuclear Code\n
00450 "'
00451         )
00452         parser.add_argument(
00453             "-k",
00454             "--reading-frame",
00455             type = int,
00456             dest = "reading_frame",
00457             choices = [1, 2, 3],
00458             default = 1,
00459             help = "Number specifying reading frame; i.e. '2' means codons start at the second
      character of the alignment. Default: 1",
00460         )
00461         parser.add_argument(
00462             "-u",
00463             "--out-format",
00464             dest = "out_format",
00465             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00466             default = "fasta",
00467             help = "File format for the output alignment. Default: fasta"
00468         )
00469         # add shared arguments
```

```
00470         self.add_common_args(parser)
00471         args = parser.parse_args(sys.argv[2:])
00472         return args
00473
```

References amas.AMAS.ParsedArgs.add_common_args().

Here is the call graph for this function:



7.7.3.11 trim()

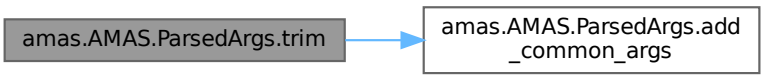
```
amas.AMAS.ParsedArgs.trim (
               self )
```

Definition at line 134 of file AMAS.py.

```
00134     def trim(self):
00135         # trim command
00136         parser = argparse.ArgumentParser(
00137             formatter_class=argparse.RawDescriptionHelpFormatter,
00138             description="'Trim alignment by occupancy. Optionally removes sites that are not parsimony
      informative."'
00139             "'\nCAUTION: when running on amino acids stop codons marked with * will be treated as
      missing data!"'
00140         )
00141         parser.add_argument(
00142             "-u",
00143             "--out-format",
00144             dest = "out_format",
00145             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00146             default = "fasta",
00147             help = "File format for the output alignment. Default: fasta"
00148         )
00149         parser.add_argument(
00150             "-o",
00151             "--trim-out",
00152             dest = "trim_out",
00153             help = "File name for the trimmed alignment when providing a single file as input."
00154         )
00155         parser.add_argument(
00156             "-t",
00157             "--trim-fraction",
00158             type = proportion,
00159             dest = "trim_fraction",
00160             default = 0.6,
00161             help = "Columns in the alignments with occupancy lower than this value will be removed.
      Default: 0.6"
00162         )
00163         parser.add_argument(
00164             "-p",
00165             "--retain-only-parsimony-sites",
00166             dest = "parsimony_check",
00167             action = "store_true",
00168             default = False,
00169             help = "Only write parsimony informative columns in trimmed alignment Default: write all
      columns"
00170         )
00171         # add shared arguments
00172         self.add_common_args(parser)
00173         args = parser.parse_args(sys.argv[2:])
00174         return args
00175
```

References amas.AMAS.ParsedArgs.add_common_args().

Here is the call graph for this function:



7.7.4 Member Data Documentation

7.7.4.1 args

`amas.AMAS.ParsedArgs.args`

Definition at line 79 of file AMAS.py.

Referenced by amas.AMAS.ParsedArgs.get_args_dict().

The documentation for this class was generated from the following file:

- amas/AMAS.py

Chapter 8

File Documentation

8.1 amas/__init__.py File Reference

Namespaces

- namespace amas

Variables

- str amas.__author__ = 'Marek Borowiec'
- str amas.__email__ = 'petiolus@gmail.com'
- str amas.__version__ = '1.02'
- amas.__all__ = dir()

8.2 __init__.py

Go to the documentation of this file.

```
00001 # -*- coding: utf-8 -*-
00002
00003 __author__ = 'Marek Borowiec'
00004 __email__ = 'petiolus@gmail.com'
00005 __version__ = '1.02'
00006 __all__ = dir()
```

8.3 amas/AMAS.py File Reference

Classes

- class amas.AMAS.ParsedArgs
- class amas.AMAS.FileHandler
- class amas.AMAS.FileParser
- class amas.AMAS.Alignment
- class amas.AMAS.AminoAcidAlignment
- class amas.AMAS.DNAAlignment
- class amas.AMAS.MetaAlignment

Namespaces

- namespace amas
- namespace amas.AMAS

Functions

- amas.AMAS.proportion (x)
- amas.AMAS.main ()
- amas.AMAS.run ()

8.4 AMAS.py

Go to the documentation of this file.

```
00001 #! /usr/bin/env python3
00002 # -*- coding: utf-8 -*-
00003 # vim:fileencoding=utf-8
00004
00005 #   Program to calculate various statistics on a multiple sequence alignment
00006 #   and allow efficient manipulation of phylogenomic data sets
00007
00008 #   Copyright (C) 2015 Marek Borowiec
00009
00010 #   This program is free software: you can redistribute it and/or modify
00011 #   it under the terms of the GNU General Public License as published by
00012 #   the Free Software Foundation, either version 3 of the License, or
00013 #   (at your option) any later version.
00014
00015 #   This program is distributed in the hope that it will be useful,
00016 #   but WITHOUT ANY WARRANTY; without even the implied warranty of
00017 #   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
00018 #   GNU General Public License for more details.
00019
00020 #   You should have received a copy of the GNU General Public License
00021 #   along with this program.  If not, see <http://www.gnu.org/licenses/>.
00022
00023 """
00024 This stand-alone program allows manipulations of multiple sequence
00025 alignments. It supports sequential FASTA, PHYLIP, NEXUS, and interleaved PHYLIP
00026 and NEXUS formats for DNA and aino acid sequences. It can print summary statistics,
00027 convert among formats, and concatenate alignments.
00028
00029 Current statistics include the number of taxa, alignment length, total number
00030 of matrix cells, overall number of undetermined characters, percent of missing
00031 data, AT and GC contents (for DNA alignments), number and proportion of
00032 variable sites, number and proportion of parsimony informative sites,
00033 and counts of all characters present in the relevant (nucleotide or amino acid) alphabet.
00034 """
00035
00036
00037 import argparse, multiprocessing as mp, re, sys
00038 from random import sample
00039 from os import path, remove
00040 from collections import defaultdict, Counter
00041 from itertools import compress
00042
00043 def proportion(x):
00044     # needed to prevent input of invalid floats in trim mode
00045     x = float(x)
00046     if x < 0.0 or x > 1.0:
00047         raise argparse.ArgumentTypeError("%r not in range [0.0, 1.0]" % (x,))
00048     return x
00049
00050 class ParsedArgs:
00051
00052     def __init__(self):
00053         parser = argparse.ArgumentParser(
00054             usage="'AMAS <command> [<args>]
00055
00056 The AMAS commands are:
00057   concat            Concatenate input alignments.
00058   convert           Convert to other file format.
00059   replicate         Create replicate data sets for phylogenetic jackknife.
00060   split             Split alignment according to a partitions file.
00061   summary           Write alignment summary.
```

```
00062   remove            Remove taxa from alignment.
00063   translate         Translate DNA alignment into protein alignment.
00064   trim              Remove columns from alignment.
00065   metapartitions    Runs `split` and concatenates the output.
00066
00067
00068 Use AMAS <command> -h for help with arguments of the command of interest
00069 "'
00070         )
00071
00072         parser.add_argument(
00073             "command",
00074             help="Subcommand to run"
00075         )
00076
00077         # parse_args defaults to [1:] for args, but you need to
00078         # exclude the rest of the args too, or validation will fail
00079         self.args = parser.parse_args(sys.argv[1:2])
00080         if not hasattr(self, self.args.command):
00081             print('Unrecognized command')
00082             parser.print_help()
00083             exit(1)
00084         # use dispatch pattern to invoke method with same name
00085         getattr(self, self.args.command)()
00086
00087     def add_common_args(self, parser):
00088         # define required arguments for every command
00089         requiredNamed = parser.add_argument_group('required arguments')
00090         parser.add_argument(
00091             "-e",
00092             "--check-align",
00093             dest = "check_align",
00094             action = "store_true",
00095             default = False,
00096             help = "Check if input sequences are aligned. Default: no check"
00097         )
00098         parser.add_argument(
00099             # parallelization is used for file parsing and calculating summary stats
00100             "-c",
00101             "--cores",
00102             dest = "cores",
00103             default = 1,
00104             help = "Number of cores used. Default: 1"
00105         )
00106
00107         requiredNamed.add_argument(
00108             "-i",
00109             "--in-files",
00110             nargs = "+",
00111             type = str,
00112             dest = "in_files",
00113             required = True,
00114             help = """Alignment files to be taken as input.
00115             You can specify multiple files using wildcards (e.g. --in-files *fasta)"""
00116         )
00117         requiredNamed.add_argument(
00118             "-f",
00119             "--in-format",
00120             dest = "in_format",
00121             required = True,
00122             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00123             help = "The format of input alignment"
00124         )
00125         requiredNamed.add_argument(
00126             "-d",
00127             "--data-type",
00128             dest = "data_type",
00129             required = True,
00130             choices = ["aa", "dna"],
00131             help = "Type of data"
00132         )
00133
00134     def trim(self):
00135         # trim command
00136         parser = argparse.ArgumentParser(
00137             formatter_class=argparse.RawDescriptionHelpFormatter,
00138             description="'Trim alignment by occupancy. Optionally removes sites that are not parsimony
    informative."'
00139             "'\nCAUTION: when running on amino acids stop codons marked with * will be treated as
    missing data!"'
00140         )
00141         parser.add_argument(
00142             "-u",
00143             "--out-format",
00144             dest = "out_format",
00145             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00146             default = "fasta",
```

```
00147             help = "File format for the output alignment. Default: fasta"
00148         )
00149         parser.add_argument(
00150             "-o",
00151             "--trim-out",
00152             dest = "trim_out",
00153             help = "File name for the trimmed alignment when providing a single file as input."
00154         )
00155         parser.add_argument(
00156             "-t",
00157             "--trim-fraction",
00158             type = proportion,
00159             dest = "trim_fraction",
00160             default = 0.6,
00161             help = "Columns in the alignments with occupancy lower than this value will be removed.
      Default: 0.6"
00162         )
00163         parser.add_argument(
00164             "-p",
00165             "--retain-only-parsimony-sites",
00166             dest = "parsimony_check",
00167             action = "store_true",
00168             default = False,
00169             help = "Only write parsimony informative columns in trimmed alignment Default: write all
      columns"
00170         )
00171         # add shared arguments
00172         self.add_common_args(parser)
00173         args = parser.parse_args(sys.argv[2:])
00174         return args
00175
00176     def summary(self):
00177        # summary command
00178         parser = argparse.ArgumentParser(
00179             description="Write alignment summary",
00180         )
00181         parser.add_argument(
00182             "-o",
00183             "--summary-out",
00184             dest = "summary_out",
00185             default = "summary.txt",
00186             help = "File name for the alignment summary. Default: 'summary.txt'"
00187         )
00188         parser.add_argument(
00189             "-s",
00190             "--by-taxon",
00191             dest = "by_taxon_summary",
00192             action = "store_true",
00193             default = False,
00194             help = "In addition to alignment summary, write by sequence/taxon summaries. Default:
      Don't write"
00195         )
00196         # add shared arguments
00197         self.add_common_args(parser)
00198         args = parser.parse_args(sys.argv[2:])
00199         return args
00200
00201     def concat(self):
00202         # concat command
00203         parser = argparse.ArgumentParser(
00204             description="Concatenate input alignments"
00205         )
00206         parser.add_argument(
00207             "-p",
00208             "--concat-part",
00209             dest = "concat_part",
00210             default = "partitions.txt",
00211             help = "File name for the concatenated alignment partitions. Default: 'partitions.txt'"
00212         )
00213         parser.add_argument(
00214             "-t",
00215             "--concat-out",
00216             dest = "concat_out",
00217             default = "concatenated.out",
00218             help = "File name for the concatenated alignment. Default: 'concatenated.out'"
00219         )
00220         parser.add_argument(
00221             "-u",
00222             "--out-format",
00223             dest = "out_format",
00224             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00225             default = "fasta",
00226             help = "File format for the output alignment. Default: fasta"
00227         )
00228         parser.add_argument(
00229             "-y",
00230             "--part-format",
```

```
00231             dest = "part_format",
00232             choices = ["nexus", "iqtree-nexus", "raxml", "unspecified"],
00233             default = "unspecified",
00234             help = "Format of the partitions file. Default: 'unspecified'"
00235         )
00236         parser.add_argument(
00237             "-n",
00238             "--codons",
00239             dest = "codons",
00240             choices = ["none", "12", "123"],
00241             default = "none",
00242             help = "Use codon partitioning for 1st and 2nd or all three positions. Default: Don't use"
00243         )
00244         # add shared arguments
00245         self.add_common_args(parser)
00246         args = parser.parse_args(sys.argv[2:])
00247         return args
00248
00249     def convert(self):
00250         # convert command
00251         parser = argparse.ArgumentParser(
00252             description="Convert to other file format",
00253         )
00254         parser.add_argument(
00255             "-u",
00256             "--out-format",
00257             dest = "out_format",
00258             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00259             default = "fasta",
00260             help = "File format for the output alignment. Default: fasta"
00261         )
00262         # add shared arguments
00263         self.add_common_args(parser)
00264         args = parser.parse_args(sys.argv[2:])
00265         return args
00266
00267     def replicate(self):
00268         # replicate command
00269         parser = argparse.ArgumentParser(
00270             description="Create replicate datasets for phylogenetic jackknife",
00271         )
00272         parser.add_argument(
00273             "-r",
00274             "--rep-aln",
00275             nargs = 2,
00276             type = int,
00277             dest = "replicate_args",
00278             help = "Create replicate data sets for phylogenetic jackknife [replicates, no alignments
      for each replicate]",
00279             required = True
00280         )
00281         parser.add_argument(
00282             "-u",
00283             "--out-format",
00284             dest = "out_format",
00285             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00286             default = "fasta",
00287             help = "File format for the output alignment. Default: fasta"
00288         )
00289         # add shared arguments
00290         self.add_common_args(parser)
00291         args = parser.parse_args(sys.argv[2:])
00292         return args
00293
00294     def split(self):
00295         # split command
00296         parser = argparse.ArgumentParser(
00297             description="Split alignment according to a partitions file",
00298         )
00299         parser.add_argument(
00300             "-l",
00301             "--split-by",
00302             dest = "split_by",
00303             help = "File name for partitions to be used for alignment splitting.",
00304             required = True
00305         )
00306         parser.add_argument(
00307             "-j",
00308             "--remove-empty",
00309             dest = "remove_empty",
00310             action = "store_true",
00311             default = False,
00312             help = "Remove taxa with sequences composed of only undetermined characters? Default:
      Don't remove"
00313         )
00314         parser.add_argument(
00315             "-u",
```

```
00316             "--out-format",
00317             dest = "out_format",
00318             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00319             default = "fasta",
00320             help = "File format for the output alignment. Default: fasta"
00321         )
00322         parser.add_argument(
00323             "--no-san",
00324             dest = "no_sup_aln_name",
00325             action = "store_true",
00326             default = False,
00327             help = "'Don't prepend the input (super)alignment filename to the partition-alignment
      filenames output by `split`"'
00328         )
00329         # add shared arguments
00330         self.add_common_args(parser)
00331         args = parser.parse_args(sys.argv[2:])
00332         return args
00333
00334     def metapartitions(self):
00335         # metapartitions command
00336         parser = argparse.ArgumentParser(
00337             formatter_class=argparse.RawTextHelpFormatter,
00338             description="'Split alignment according to a partition file, then concatenate the
      output."'
00339             "'\n\nuse case:\n"'
00340             "'    Some utilities cannot parse partition definitions containing strides (\\) and/or
      discontinuous ranges.\n"'
00341             "'    In such case, running `split` + `concat` in separate passes can convert a
      corresponding (super)alignment it into an\n"'
00342             "'    equivalent compatible form with contiguous (meta)partitions; this may also require
      renaming metapartiton alignments\n"'
00343             "'    and partition file entries in order to remove tags applied by each respective
      operation.\n\n"'
00344             "'    `metapartitions` combines these steps into one command, with the options `--prepend`
      and `--no-mpan`\n"'
00345             "'    providing additional control over the collated (meta)partition names (see their
      respective help entries).\n\n"'
00346             "'    Note: in this mode, the format of the input (super)alignment file determines that of
      all outputs (-u|--out-format is disabled)!\n\n"'
00347         )
00348         parser.add_argument(
00349             "-p",
00350             "--concat-part",
00351             dest = "concat_part",
00352             default = "metapartitions.txt",
00353             help = "Partition file(name) for the final concatenated alignment of metapartitions.
      Default: 'metapartitions.txt'"
00354         )
00355         parser.add_argument(
00356             "-t",
00357             "--concat-out",
00358             dest = "concat_out",
00359             default = "concatenated-meta.out",
00360             help = "File name for the concatenated alignment of metapartitions. Default:
      'concatenated-meta.out'"
00361         )
00362         parser.add_argument(
00363             "-y",
00364             "--part-format",
00365             dest = "part_format",
00366             choices = ["nexus", "iqtree-nexus", "raxml", "unspecified"],
00367             default = "unspecified",
00368             help = "Partitions file format for the final concatenated alignment of metapartitions.
      Default: 'unspecified'"
00369         )
00370         parser.add_argument(
00371             "-l",
00372             "--split-by",
00373             dest = "split_by",
00374             help = "Partition file(name) to be used for splitting the initial concatenated
      alignment.",
00375             required = True
00376         )
00377         parser.add_argument(
00378             "-j",
00379             "--remove-empty",
00380             dest = "remove_empty",
00381             action = "store_true",
00382             default = False,
00383             help = "Remove taxa with sequences composed of only undetermined characters? Default:
      Don't remove"
00384         )
00385         parser.add_argument(
00386             "--no-san",
00387             dest = "no_sup_aln_name",
00388             action = "store_true",
```

```
00389             default = False,
00390             help = "'Don't prepend the input (super)alignment filename to the
    (meta)partition-alignment filenames output by `split`"'
00391         )
00392         parser.add_argument(
00393             "--prepend",
00394             dest = "prepend_label",
00395             default = None,
00396             help = "'Prepend <string> to the partition counter in partition file, e.g."'
00397             "'\n            --prepend <string>:  <string>_p001_metapartition_alignment_name = 1-1200
    ..."'
00398             "'\n               Default (None):           p001_metapartition_alignment_name = 1-1200
    ..."'
00399             "'\n--no-mpan + --prepend <string>:  <string>_p001 = 1-1200 ..."'
00400         )
00401         parser.add_argument(
00402             "--no-mpan",
00403             dest = "no_mpan",
00404             action = "store_true",
00405             default = False,
00406             help = "'Omits (meta)partition alignment names when printing partition file, e.g."'
00407             "'\n                     --no-mpan:          p001 = 1-1200 ..."'
00408             "'\n               Default (False):          p001_metapartition_alignment_name = 1-1200
    ..."'
00409             "'\n--prepend <string> + --no-mpan: <string>_p001 = 1-1200 ..."'
00410         )
00411         # add shared arguments
00412         self.add_common_args(parser)
00413         args = parser.parse_args(sys.argv[2:])
00414         return args
00415
00416     def translate(self):
00417         # translate command
00418         parser = argparse.ArgumentParser(
00419             formatter_class=argparse.RawTextHelpFormatter,
00420             description="Translate a protein-coding DNA alignment into amino acids"
00421         )
00422         parser.add_argument(
00423             "-b",
00424             "--code",
00425             type = int,
00426             dest = "genetic_code",
00427             choices = [1, 2, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 16, 21, 22, 23, 24, 25, 26],
00428             default = 1,
00429             help = "'\nNCBI genetic code to use (Default: 1):"'
00430 "'
00431    1. The Standard Code
00432    2. The Vertebrate Mitochondrial Code
00433    3. The Yeast Mitochondrial Code
00434    4. The Mold, Protozoan, and Coelenterate Mitochondrial Code and the Mycoplasma/Spiroplasma Code
00435    5. The Invertebrate Mitochondrial Code
00436    6. The Ciliate, Dasycladacean and Hexamita Nuclear Code
00437    9. The Echinoderm and Flatworm Mitochondrial Code
00438   10. The Euplotid Nuclear Code
00439   11. The Bacterial, Archaeal and Plant Plastid Code
00440   12. The Alternative Yeast Nuclear Code
00441   13. The Ascidian Mitochondrial Code
00442   14. The Alternative Flatworm Mitochondrial Code
00443   16. Chlorophycean Mitochondrial Code
00444   21. Trematode Mitochondrial Code
00445   22. Scenedesmus obliquus Mitochondrial Code
00446   23. Thraustochytrium Mitochondrial Code
00447   24. Pterobranchia Mitochondrial Code
00448   25. Candidate Division SR1 and Gracilibacteria Code
00449   26. Pachysolen tannophilus Nuclear Code\n
00450 "'
00451         )
00452         parser.add_argument(
00453             "-k",
00454             "--reading-frame",
00455             type = int,
00456             dest = "reading_frame",
00457             choices = [1, 2, 3],
00458             default = 1,
00459             help = "Number specifying reading frame; i.e. '2' means codons start at the second
    character of the alignment. Default: 1",
00460         )
00461         parser.add_argument(
00462             "-u",
00463             "--out-format",
00464             dest = "out_format",
00465             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00466             default = "fasta",
00467             help = "File format for the output alignment. Default: fasta"
00468         )
00469         # add shared arguments
00470         self.add_common_args(parser)
```

```
00471         args = parser.parse_args(sys.argv[2:])
00472         return args
00473
00474     def remove(self):
00475         # remove taxa command
00476         parser = argparse.ArgumentParser(
00477             description="Remove taxa from alignment",
00478         )
00479         parser.add_argument(
00480             "-x",
00481             "--taxa-to-remove",
00482             nargs = "+",
00483             type = str,
00484             dest = "taxa_to_remove",
00485             help = "Taxon/sequence names to be removed.",
00486             required = True
00487         )
00488         parser.add_argument(
00489             "-u",
00490             "--out-format",
00491             dest = "out_format",
00492             choices = ["fasta", "phylip", "nexus", "phylip-int", "nexus-int"],
00493             default = "fasta",
00494             help = "File format for the output alignment. Default: fasta"
00495         )
00496         parser.add_argument(
00497             "-g",
00498             "--out-prefix",
00499             dest = "out_prefix",
00500             default = "reduced_",
00501             help = "File name prefix for the concatenated alignment. Default: 'reduced_'"
00502         )
00503         # add shared arguments
00504         self.add_common_args(parser)
00505         args = parser.parse_args(sys.argv[2:])
00506         return args
00507
00508     def get_args_dict(self):
00509         # store arguments in a dictionary
00510         command = self.args.__dict__
00511         arguments = getattr(self, self.args.command)().__dict__
00512         argument_dictionary = command.copy()
00513         argument_dictionary.update(arguments)
00514
00515         return argument_dictionary
00516
00517
00518 class FileHandler:
00519     """Define file handle that closes when out of scope"""
00520
00521     def __init__(self, file_name):
00522         self.file_name = file_name
00523
00524     def __enter__(self):
00525         try:
00526             self.in_file = open(self.file_name, "r", encoding="utf-8")
00527         except FileNotFoundError:
00528             print("ERROR: File '" + self.file_name + "' not found.")
00529             sys.exit()
00530         return self.in_file
00531
00532     def __exit__(self, *args):
00533         self.in_file.close()
00534
00535     def get_file_name(self):
00536         return self.file_name
00537
00538 class FileParser:
00539     """Parse file contents and return sequences and sequence names"""
00540
00541     def __init__(self, in_file):
00542         self.in_file = in_file
00543         with FileHandler(in_file) as handle:
00544             self.in_file_lines = handle.read().rstrip("\r\n")
00545
00546     def fasta_parse(self):
00547         # use regex to parse names and sequences in sequential fasta files
00548         matches = re.finditer(
00549             r"^>(.*[^$])([^>]*)",
00550             self.in_file_lines, re.MULTILINE
00551         )
00552         records = {}
00553
00554         for match in matches:
00555             name_match = match.group(1).replace("\n", "")
00556             seq_match = match.group(2).replace("\n", "").upper()
00557             seq_match = self.translate_ambiguous(seq_match)
```

```
00558             records[name_match] = seq_match
00559
00560         return records
00561
00562     def phylip_parse(self):
00563         # use regex to parse names and sequences in sequential phylip files
00564         matches = re.finditer(
00565             r"^(\s+)?(\S+)\s+([A-Za-z*?.{}-]+)",
00566             self.in_file_lines, re.MULTILINE
00567         )
00568
00569         records = {}
00570
00571         for match in matches:
00572             name_match = match.group(2).replace("\n", "")
00573             seq_match = match.group(3).replace("\n", "").upper()
00574             seq_match = self.translate_ambiguous(seq_match)
00575             records[name_match] = seq_match
00576
00577         return records
00578
00579     def phylip_interleaved_parse(self):
00580         # use regex to parse names and sequences in interleaved phylip files
00581         tax_chars_matches = re.finditer(
00582             r"^(\s+)?([0-9]+)[ \t]+([0-9]+)",
00583             self.in_file_lines, re.MULTILINE
00584         )
00585         name_matches = re.finditer(
00586             r"^(\s+)?(\S+)[ \t]+[A-Za-z*?.{}-]+",
00587             self.in_file_lines, re.MULTILINE
00588         )
00589         seq_matches = re.finditer(
00590             r"(^(\s+)?\S+[ \t]+|^)([A-Za-z*?.{}-]+)$",
00591             self.in_file_lines, re.MULTILINE
00592         )
00593         # get number of taxa and chars
00594         for match in tax_chars_matches:
00595             tax_match = match.group(2)
00596             chars_match = match.group(3)
00597
00598         # initiate lists for taxa names and sequence strings on separate lines
00599         taxa = []
00600         sequences = []
00601         # initiate a dictionary for the name:sequence records
00602         records = {}
00603         # initiate a counter to keep track of sequences strung together
00604         # from separate lines
00605         counter = 0
00606
00607         for match in name_matches:
00608             name_match = match.group(2).replace("\n", "")
00609             taxa.append(name_match)
00610
00611         for match in seq_matches:
00612             seq_match = match.group(3).replace("\n", "").upper()
00613             seq_match = self.translate_ambiguous(seq_match)
00614             sequences.append(seq_match)
00615         # try parsing PHYLUCE-style interleaved phylip
00616         if len(taxa) != int(tax_match):
00617             taxa = []
00618             sequences = []
00619             matches = re.finditer(
00620                 r"(^(\s+)?(\S+)( ){2,}|^\s+)([ A-Za-z*?.{}-]+)",
00621                 self.in_file_lines, re.MULTILINE
00622             )
00623
00624             for match in matches:
00625                 try:
00626                     name_match = match.group(3).replace("\n", "")
00627                     taxa.append(name_match)
00628                 except AttributeError:
00629                     pass
00630                 seq_match = match.group(5).replace("\n", "").upper()
00631                 seq_match = "".join(seq_match.split())
00632                 seq_match = self.translate_ambiguous(seq_match)
00633                 sequences.append(seq_match)
00634
00635         for taxon_no in range(len(taxa)):
00636             sequence = ""
00637             for index in range(counter,len(sequences),len(taxa)):
00638                 sequence += sequences[index]
00639
00640             records[taxa[taxon_no]] = sequence
00641             counter += 1
00642
00643         return records
00644
```

```
00645     def nexus_parse(self):
00646         # use regex to parse names and sequences in sequential nexus files
00647         # find the matrix block
00648         matches = re.finditer(
00649             r"(\s+)?(MATRIX\n|matrix\n|MATRIX\r\n|matrix\r\n)(.*?;)",
00650             self.in_file_lines, re.DOTALL
00651         )
00652 
00653         records = {}
00654         # get names and sequences from the matrix block
00655 
00656         for match in matches:
00657             matrix_match = match.group(3)
00658             seq_matches = re.finditer(
00659                 r"^(\s+)?[']?(\S+\s\S+|\S+)[']?\s+([A-Za-z*?.{}-]+)($|\s+\[[0-9]+\]$)",
00660                 matrix_match, re.MULTILINE
00661             )
00662 
00663             for match in seq_matches:
00664                 name_match = match.group(2).replace("\n", "")
00665                 seq_match = match.group(3).replace("\n", "").upper()
00666                 seq_match = self.translate_ambiguous(seq_match)
00667                 records[name_match] = seq_match
00668 
00669         return records
00670 
00671     def nexus_interleaved_parse(self):
00672         # use regex to parse names and sequences in sequential nexus files
00673         # find the matrix block
00674         matches = re.finditer(
00675             r"(\s+)?(MATRIX\n|matrix\n|MATRIX\r\n|matrix\r\n)(.*?;)",
00676             self.in_file_lines, re.DOTALL
00677         )
00678         # initiate lists for taxa names and sequence strings on separate lines
00679         taxa = []
00680         sequences = []
00681         # initiate a dictionary for the name:sequence records
00682         records = {}
00683 
00684         for match in matches:
00685             matrix_match = match.group(3)
00686             # get names and sequences from the matrix block
00687             seq_matches = re.finditer(
00688                 r"^(\s+)?[']?(\S+\s\S+|\S+)[']?\s+([A-Za-z*?.{}-]+)($|\s+\[[0-9]+\]$)",
00689                 matrix_match, re.MULTILINE
00690             )
00691 
00692             for match in seq_matches:
00693                 name_match = match.group(2)
00694                 if name_match not in taxa:
00695                     taxa.append(name_match)
00696                 seq_match = match.group(3)
00697 
00698                 sequences.append(seq_match)
00699 
00700         # initiate a counter to keep track of sequences strung together
00701         # from separate lines
00702         counter = 0
00703 
00704         for taxon_no in range(len(taxa)):
00705 
00706             full_length_sequence = "".join([sequences[index] for index in
      range(counter,len(sequences),len(taxa))])
00707             records[taxa[taxon_no]] = self.translate_ambiguous(full_length_sequence).replace("\n",
      "").upper()
00708             counter += 1
00709 
00710         return records
00711 
00712     def translate_ambiguous(self, seq):
00713         # translate ambiguous characters from curly bracket format
00714         # to single letter format
00715         # also remove spaces from sequences
00716         seq = seq.replace("{GT}", "K")
00717         seq = seq.replace("{AC}", "M")
00718         seq = seq.replace("{AG}", "R")
00719         seq = seq.replace("{CT}", "Y")
00720         seq = seq.replace("{CG}", "S")
00721         seq = seq.replace("{AT}", "W")
00722         seq = seq.replace("{CGT}", "B")
00723         seq = seq.replace("{ACG}", "V")
00724         seq = seq.replace("{ACT}", "H")
00725         seq = seq.replace("{AGT}", "D")
00726         seq = seq.replace("{GATC}", "N")
00727         seq = seq.replace(" ", "")
00728 
00729         return seq
```

```
00730
00731     def partitions_parse(self):
00732         # parse partitions file using regex
00733         # original: `matches = re.finditer(r"^(\s+)?([^ =]+)[ =]+([\\0-9, -]+)", self.in_file_lines,
      re.MULTILINE)`
00734         # new version: more permissive -> handles PartionFinder/RAxML/(IQ-TREE 2)best_scheme.nex
      format partition files
00735         matches = re.finditer(
00736             r"""^[ \t]*                                 # start of line w/ zero-or-more (just)
      whitespaces/tabs
00737                 (
00738                  (?P<nexus>charset[ ]+)                 # case 1: (IQ-TREE 2)best_scheme.nex partition
      directive; partition name
00739                  |
00740                  (?P<raxml>[A-Za-z0-9_+\.]+,[ \t]+)     # case 2: RAxML/RAxML-NG model(+other pars);
      partition name
00741                 )?
00742                 (?P<partition_name>[A-Za-z0-9_\-]+)     # case 3: just partition name (including one
      that contain residual '-out'/'-meta' suffixes)
00743                 [ ]*=[ ]*                               # whitespace-padded (or unpadded) '=':
      (IQ-TREE 2)best_scheme.nex compatabiliy
00744                 (?P<numbers>[\\0-9, -]+)                # position ranges w/ stride (multiple
      intervals; from original regex)
00745                 (?P<nexus_term>[ ]*[;])?                # whitespace-prepended (or unprepended) ';'
      (nexus terminator)
00746             """,
00747             self.in_file_lines,
00748             re.MULTILINE | re.VERBOSE
00749         )
00750
00751         # initiate list to store dictionaries with lists
00752         # of slice positions as values
00753         partitions = []
00754         add_to_partitions = partitions.append
00755
00756         for match in matches:
00757             # initiate dictionary of partition name as key
00758             dict_of_dicts = {}
00759             # and list of dictionaries with slice positions
00760             list_of_dicts = []
00761             add_to_list_of_dicts = list_of_dicts.append
00762             # get parition name and numbers from parsed partition strings
00763             partition_name = match.group('partition_name')
00764             numbers = match.group('numbers')
00765             # remove any whitespace padding '-' (to be consistent with partition-writing format)
00766             numbers = re.sub(r"[ ]*-[ ]*", "-", numbers)
00767             # find all numbers that will be used to parse positions
00768             positions = re.findall(r"([^ ,]+)", numbers)
00769
00770             for position in positions:
00771                 # create dictionary for slicing input sequence
00772                 # conditioning on whether positions are represented
00773                 # by range, range with stride, or single number
00774                 pos_dict = {}
00775
00776                 if "-" in position:
00777                     m = re.search(r"([0-9]+)-([0-9]+)", position)
00778                     pos_dict["start"] = int(m.group(1)) - 1
00779                     pos_dict["stop"] = int(m.group(2))
00780                 else:
00781                     pos_dict["start"] = int(position) - 1
00782                     pos_dict["stop"] = int(position)
00783
00784                 if "\\" in position:
00785                     # Note: the value of `N` in `...\N` isn't read: the script simply assumes `N` is
      consistent with the number of
00786                     # increments per interval when the alignment is parsed with a stride of 3
      (designating each cpos).
00787                     # E.g. For the partition file:
00788                     #       ...'1-N\2'
00789                     #       ...'2-N\2'
00790                     #       ...'(N+1)-M\2'
00791                     #       ...'(N+2)-M\2'
00792                     # 3'cpos are ignored due to the absence of intervals `3-N...', `(N+3)-M...', not
      because the associated stride values are'\2'
00793                     pos_dict["stride"] = 3
00794                 elif "\\" not in position:
00795                     pos_dict["stride"] = 1
00796
00797                 add_to_list_of_dicts(pos_dict)
00798
00799             dict_of_dicts[partition_name] = list_of_dicts
00800             add_to_partitions(dict_of_dicts)
00801
00802         return partitions
00803
00804
```

```
00805 class Alignment:
00806     """Base class: Gets in parsed sequences as input and summarizes their stats.
00807     Based on the data type, the subclasses AminoAcidAlignment & DNAAlignment define the attributes:
00808     `alphabet`, `missing_ambiguous_chars`, `missing_chars`, `non_alphabet`
00809     """
00810
00811     def __init__(self, in_file, in_format, data_type):
00812         # initialize alignment class with parsed records and alignment name as arguments,
00813         # create empty lists for list of sequences, sites without
00814         # ambiguous or missing characters, and initialize variable for the number
00815         # of parsimony informative sites
00816         self.in_file = in_file
00817         self.in_format = in_format
00818         self.data_type = data_type
00819
00820         self.parsed_aln = self.get_parsed_aln()
00821
00822     def __str__(self):
00823         #  purpose of override? (originally returned method object)
00824         return self.get_name()
00825
00826     def get_aln_input(self):
00827         # open and parse input file
00828         aln_input = FileParser(self.in_file)
00829         return aln_input
00830
00831     def get_parsed_aln(self):
00832         # parse according to the given format
00833         aln_input = self.get_aln_input()
00834         if self.in_format == "fasta":
00835             parsed_aln = aln_input.fasta_parse()
00836         elif self.in_format == "phylip":
00837             parsed_aln = aln_input.phylip_parse()
00838         elif self.in_format == "phylip-int":
00839             parsed_aln = aln_input.phylip_interleaved_parse()
00840         elif self.in_format == "nexus":
00841             parsed_aln = aln_input.nexus_parse()
00842         elif self.in_format == "nexus-int":
00843             parsed_aln = aln_input.nexus_interleaved_parse()
00844
00845         return parsed_aln
00846
00847     def summarize_alignment(self):
00848         # call methods to create sequences list, matrix, sites without ambiguous or
00849         # missing characters; get and summarize alignment statistics
00850         summary = []
00851         self.length = str(self.get_alignment_length())
00852         self.matrix = self.matrix_creator()
00853         self.no_missing_ambiguous = self.get_sites_no_missing_ambiguous()
00854         self.variable_sites = self.get_variable()
00855         self.prop_variable = self.get_prop_variable()
00856         self.parsimony_informative = self.get_parsimony_informative()
00857         self.prop_parsimony = self.get_prop_parsimony()
00858         self.missing_records = self.get_missing_from_parsed()
00859         name = str(self.get_name())
00860         taxa_no = str(self.get_taxa_no())
00861         cells = str(self.get_matrix_cells())
00862         missing = str(self.get_missing())
00863         missing_percent = str(self.get_missing_percent())
00864         self.check_data_type()
00865         summary = [
00866             name,
00867             taxa_no,
00868             self.length,
00869             cells,
00870             missing,
00871             missing_percent,
00872             str(self.variable_sites),
00873             str(self.prop_variable),
00874             str(self.parsimony_informative),
00875             str(self.prop_parsimony)
00876         ]
00877         return summary
00878
00879     def summarize_alignment_by_taxa(self):
00880         # get summary for all taxa/sequences in alignment
00881         per_taxon_summary = []
00882         taxa_no = self.get_taxa_no()
00883         self.missing_records = self.get_missing_from_parsed()
00884         self.length = self.get_alignment_length()
00885         lengths = (self.length for i in range(taxa_no))
00886         name = self.get_name()
00887         names = (name for i in range(taxa_no))
00888         taxa_names = (
00889             taxon.replace(" ", "_").replace(".", "_").replace("'", "")
00890             for taxon, missing_count, missing_percent in self.missing_records
00891         )
```

```
00892         missing = (missing_count for taxon, missing_count, missing_percent in self.missing_records)
00893         missing_percent = (missing_percent for taxon, missing_count, missing_percent in
      self.missing_records)
00894         self.check_data_type()
00895         per_taxon_summary = (names, taxa_names, lengths, missing, missing_percent)
00896         zipped = list(zip(*per_taxon_summary))
00897         return zipped
00898
00899     def get_char_summary(self):
00900         # get summary of frequencies for all characters
00901         characters = []
00902         counts = []
00903         add_to_chars = characters.append
00904         add_to_counts = counts.append
00905         char_count_dicts = self.get_counts()
00906         for char in self.alphabet:
00907             add_to_chars(char)
00908             if char in char_count_dicts.keys():
00909                 add_to_counts(str(char_count_dicts[char]))
00910             else:
00911                 add_to_counts("0")
00912         return characters, counts
00913
00914     def get_taxon_char_summary(self):
00915         # get summary of frequencies for all characters
00916         records = (self.append_count(char_dict) for taxon, char_dict in self.get_counts_from_parsed())
00917         return records
00918
00919     def append_count(self, char_dict):
00920         count_list = []
00921         for char in self.alphabet:
00922             if char in char_dict.keys():
00923                 count_list.append(char_dict[char])
00924             else:
00925                 count_list.append(0)
00926         return count_list
00927
00928     def matrix_creator(self):
00929         # decompose character matrix into a two-dimensional list
00930         matrix = [list(sequence) for sequence in self.parsed_aln.values()]
00931         return matrix
00932
00933     def get_column(self, i):
00934         # get site from the character matrix
00935         return [row[i] for row in self.matrix]
00936
00937     def all_same(self, site):
00938         # check if all elements of a site are the same
00939         return not site or site.count(site[0]) == len(site)
00940
00941     def get_sites_no_missing_ambiguous(self):
00942         # get each site without missing or ambiguous characters
00943         no_missing_ambiguous_sites = [self.get_site_no_missing_ambiguous(column) for column in
      range(self.get_alignment_length())]
00944         return no_missing_ambiguous_sites
00945
00946     def get_site_no_missing_ambiguous(self, column):
00947         site = self.get_column(column)
00948         return [char for char in site if char not in self.missing_ambiguous_chars]
00949
00950     def replace_missing(self, column):
00951         return ["-" if x in self.missing_chars else x for x in self.get_column(column)]
00952
00953     def get_trim_selection(self, trim_fraction, parsimony_check):
00954         # this checks each column of alignment for minimum occupancy
00955         self.matrix = self.matrix_creator()
00956         trim_vector = []
00957         for column in range(self.get_alignment_length()):
00958             site = self.replace_missing(column)
00959             occ = (len(site) - site.count("-")) / len(site)
00960             if parsimony_check:
00961                 unique_chars = set(site)
00962                 try:
00963                     unique_chars.remove("-")
00964                 except KeyError:
00965                     pass # this occurs if we have no missing data
00966                 pattern = [base for base in unique_chars if site.count(base) >= 2]
00967                 trim_vector.append(len(pattern) >= 2 and occ >= trim_fraction)
00968             else:
00969                 trim_vector.append(occ >= trim_fraction)
00970         return trim_vector
00971
00972     def get_variable(self):
00973         # if all elements of a site without missing or ambiguous characters
00974         # are not the same, consider it variable
00975         variable = len([site for site in self.no_missing_ambiguous if not self.all_same(site)])
00976         return variable
```

```
00977
00978     def get_parsimony_informative(self):
00979         # if the count for a unique character in a site is at least two,
00980         # and there are at least two such characters in a site without missing
00981         # or ambiguous characters, consider it parsimony informative
00982         parsimony_informative = 0
00983         for site in self.no_missing_ambiguous:
00984             unique_chars = set(site)
00985             pattern = [base for base in unique_chars if site.count(base) >= 2]
00986             no_patterns = len(pattern)
00987
00988             if no_patterns >= 2:
00989                 parsimony_informative += 1
00990         return parsimony_informative
00991
00992     def get_prop_variable(self):
00993         # get proportion of variable sites to all sites
00994         prop_variable = self.variable_sites / int(self.length)
00995         return round(prop_variable, 3)
00996
00997     def get_prop_parsimony(self):
00998         # get proportion of parsimony informative sites to all sites
00999         prop_parsimony = self.parsimony_informative / int(self.length)
01000         return round(prop_parsimony, 3)
01001
01002     def get_name(self):
01003         # get input file name
01004         in_filename = path.basename(self.in_file)
01005         return in_filename
01006
01007     def get_taxa_no(self):
01008         # get number of taxa
01009         return len(self.parsed_aln.values())
01010
01011     def get_alignment_length(self):
01012         # get alignment length by just checking the first seq length
01013         # this assumes that all sequences are of equal length
01014         return len(next(iter(self.parsed_aln.values())))
01015
01016     def get_matrix_cells(self):
01017         # count all matrix cells
01018         self.all_matrix_cells = len(self.parsed_aln.values()) * int(self.length)
01019         return self.all_matrix_cells
01020
01021     def get_missing(self):
01022         # count missing characters from the list of missing for all sequences
01023         self.missing = sum(count for taxon, count, percent in self.missing_records)
01024         return self.missing
01025
01026     def get_missing_percent(self):
01027         # get missing percent
01028         missing_percent = round((self.missing / self.all_matrix_cells * 100), 3)
01029         return missing_percent
01030
01031     def get_missing_from_parsed(self):
01032         # get missing count and percent from parsed alignment
01033         # return a list of tuples with taxon name, count, and percent missing
01034         self.missing_records = sorted(
01035             [
01036                 (taxon, self.get_missing_from_seq(seq), self.get_missing_percent_from_seq(seq))
01037                 for taxon, seq in self.parsed_aln.items()
01038             ]
01039         )
01040         return self.missing_records
01041
01042     def get_missing_from_seq(self, seq):
01043         # count missing characters for individual sequence
01044         missing_count = sum(seq.count(char) for char in self.missing_chars)
01045         return missing_count
01046
01047     def get_missing_percent_from_seq(self, seq):
01048         # get missing percent from individual sequence
01049         missing_seq_percent = round((self.get_missing_from_seq(seq) / self.get_alignment_length() *
    100), 3)
01050         return missing_seq_percent
01051
01052     def get_counts(self):
01053         # get counts of each character in the used alphabet for all sequences
01054         counters = [Counter(chars) for taxon, chars in self.get_counts_from_parsed()]
01055         all_counts = sum(counters, Counter())
01056         counts_dict = dict(all_counts)
01057         return counts_dict
01058
01059     def get_counts_from_parsed(self):
01060         # get counts of all characters from parsed alignment
01061         # return a list of tuples with taxon name and counts
01062         return sorted(
```

```
01063             [
01064                 (taxon, self.get_counts_from_seq(seq))
01065                 for taxon, seq in self.parsed_aln.items()
01066             ]
01067         )
01068
01069     def get_counts_from_seq(self, seq):
01070         # get all alphabet chars count for individual sequence
01071         char_counts = {char : seq.count(char) for char in self.alphabet}
01072         return char_counts
01073
01074     def check_data_type(self):
01075         # check if the data type is correct; only one seq to save on computation
01076         seq = next(iter(self.parsed_aln.values()))
01077         self.check = any(char in self.non_alphabet for char in seq)
01078         if self.check is True:
01079             print(
01080                 "WARNING: found non-" + self.data_type + " characters. "
01081                 "Are you sure you specified the right data type?"
01082             )
01083
01084
01085 class AminoAcidAlignment(Alignment):
01086     """Alphabets specific to amino acid alignments"""
01087
01088     alphabet = ["A", "C", "D", "E", "F", "G", "H", "I", "K", "L", "M", "N", "P", "Q", "R", "S", "T",
      "V", "W", "Y", "B", "J", "Z", "X", ".", "*", "-", "?"]
01089     missing_ambiguous_chars = ["B", "J", "Z", "X", ".", "*", "-", "?"]
01090     missing_chars = ["X", ".", "*", "-", "?"]
01091     non_alphabet = ["O"]
01092
01093     def get_summary(self):
01094         # get alignment summary specific to amino acids
01095         data = self.summarize_alignment()
01096         new_data = data + list(self.get_char_summary()[1])
01097         return new_data
01098
01099     def get_taxa_summary(self):
01100         # get per-taxon/sequence alignment summary specific to amino acids
01101         data = self.summarize_alignment_by_taxa()
01102         aa_summary = (data, self.get_taxon_char_summary())
01103         zipped_list = list(zip(*aa_summary))
01104         new_data = [list(data_tupl) + chars for data_tupl, chars in zipped_list]
01105         return new_data
01106
01107 class DNAAlignment(Alignment):
01108     """Alphabets specific to DNA alignments"""
01109
01110     alphabet = ["A", "C", "G", "T", "K", "M", "R", "Y", "S", "W", "B", "V", "H", "D", "X", "N", "O",
      "-", "?"]
01111     missing_ambiguous_chars = ["K", "M", "R", "Y", "S", "W", "B", "V", "H", "D", "X", "N", "O", "-",
      "?"]
01112     missing_chars = ["X", "N", "O", "-", "?"]
01113     non_alphabet = ["E", "F", "I", "L", "P", "Q", "J", "Z", ".", "*"]
01114
01115     def get_summary(self):
01116         # get alignment summarry specific to nucleotide
01117         data = self.summarize_alignment()
01118         new_data = data + self.get_atgc_content() + list(self.get_char_summary()[1])
01119         return new_data
01120
01121     def get_taxa_summary(self):
01122         # get per-taxon/sequence alignment summary specific to nucleotides
01123         data = self.summarize_alignment_by_taxa()
01124         dna_summary = (data, self.get_list_from_atgc(), self.get_taxon_char_summary())
01125         zipped_list = list(zip(*dna_summary))
01126         new_data = [list(data_tupl) + list(atgc) + chars for data_tupl, atgc, chars in zipped_list]
01127         return new_data
01128
01129     def get_atgc_content(self):
01130         # get AC and GC contents for all sequences
01131         # AT content is the first element of AT, GC content tuple
01132         # returned by get_atgc_from_seq()
01133         atgc_records = self.get_atgc_from_parsed()
01134         at_content = round(sum(atgc[0] for taxon, atgc in atgc_records) / self.get_taxa_no(), 3)
01135         gc_content = round(1 - float(at_content), 3)
01136
01137         atgc_content = [str(at_content), str(gc_content)]
01138         return atgc_content
01139
01140     def get_list_from_atgc(self):
01141         records = (atgc for taxon, atgc in self.get_atgc_from_parsed())
01142         return records
01143
01144     def get_atgc_from_parsed(self):
01145         # get AT and GC contents from parsed alignment dictionary
01146         # return a list of tuples with taxon name, AT content, and GC content
```

```
01147         return sorted([(taxon, self.get_atgc_from_seq(seq)) for taxon, seq in
      self.parsed_aln.items()])
01148
01149     def get_atgc_from_seq(self, seq):
01150         # get AT and GC contents from individual sequences
01151
01152         at_count = seq.count("A") + seq.count("T") + seq.count("W")
01153         gc_count = seq.count("G") + seq.count("C") + seq.count("S")
01154
01155         try:
01156             at_content = round(at_count / (at_count + gc_count), 3)
01157             gc_content = round(1 - float(at_content), 3)
01158
01159         except ZeroDivisionError:
01160             at_content = 0
01161             gc_content = 0
01162
01163         return at_content, gc_content
01164
01165 class MetaAlignment:
01166     """Class of multiple sequence alignments"""
01167
01168     def __init__(self, **kwargs):
01169         # set defaults and get values from kwargs
01170         self.in_files = kwargs.get("in_files")
01171         self.in_format = kwargs.get("in_format")
01172         self.data_type = kwargs.get("data_type")
01173         self.command = kwargs.get("command")
01174         self.concat_out = kwargs.get("concat_out", "concatenated.out")
01175         self.using_metapartitions = False
01176         self.check_align = kwargs.get("check_align", False)
01177         self.cores = kwargs.get("cores")
01178         self.by_taxon_summary = kwargs.get("by_taxon_summary")
01179         self.no_sup_aln_name = False
01180         self.no_mpan = False
01181
01182         if self.command == "concat":
01183             self.codons = kwargs.get("codons", "none")
01184             if self.data_type == "aa" and self.codons != "none":
01185                 print("ERROR: when option -d|--data-type is set to 'aa', option -n|--codons must be
      set to 'none'.")
01186                 sys.exit(1)
01187
01188         if self.command == "replicate":
01189             self.no_replicates = kwargs.get("replicate_args")[0]
01190             self.no_loci = kwargs.get("replicate_args")[1]
01191
01192         if self.command == "split":
01193             self.split = kwargs.get("split_by")
01194             self.remove_empty = kwargs.get("remove_empty", False)
01195             self.no_sup_aln_name = kwargs.get("no_sup_aln_name", False)
01196
01197         if self.command == "metapartitions":
01198             self.using_metapartitions = True
01199             self.split = kwargs.get("split_by")
01200             self.remove_empty = kwargs.get("remove_empty", False)
01201             self.no_sup_aln_name = kwargs.get("no_sup_aln_name", False)
01202             self.no_mpan = kwargs.get("no_mpan", False)
01203             self.prepend_label = kwargs.get("prepend_label")
01204             if self.prepend_label is not None and isinstance(self.prepend_label, str):
01205                 self.prepend_label = self.prepend_label + "_"
01206             else:
01207                 self.prepend_label = ""
01208
01209         if self.command == "remove":
01210             self.species_to_remove = kwargs.get("taxa_to_remove")
01211             self.species_to_remove_set = set(self.species_to_remove)
01212             self.reduced_file_prefix = kwargs.get("out_prefix")
01213             self.check_taxa = kwargs.get("check_taxa", False)
01214
01215         if self.command == "translate":
01216             self.reading_frame = kwargs.get("reading_frame")
01217             self.genetic_code = kwargs.get("genetic_code")
01218
01219         if self.command == "trim":
01220             self.trim_fraction = kwargs.get("trim_fraction")
01221             self.trim_out = kwargs.get("trim_out")
01222             self.parsimony_check = kwargs.get("parsimony_check", False)
01223
01224         self.alignment_objects = self.get_alignment_objects()
01225         self.parsed_alignments = self.get_parsed_alignments()
01226
01227         # The code list:
01228         self.codes_list = """
01229          1. The Standard Code
01230          2. The Vertebrate Mitochondrial Code
01231          3. The Yeast Mitochondrial Code
```

```
01232           4. The Mold, Protozoan, and Coelenterate Mitochondrial Code and the Mycoplasma/Spiroplasma
      Code
01233           5. The Invertebrate Mitochondrial Code
01234           6. The Ciliate, Dasycladacean and Hexamita Nuclear Code
01235           9. The Echinoderm and Flatworm Mitochondrial Code
01236          10. The Euplotid Nuclear Code
01237          11. The Bacterial, Archaeal and Plant Plastid Code
01238          12. The Alternative Yeast Nuclear Code
01239          13. The Ascidian Mitochondrial Code
01240          14. The Alternative Flatworm Mitochondrial Code
01241          16. Chlorophycean Mitochondrial Code
01242          21. Trematode Mitochondrial Code
01243          22. Scenedesmus obliquus Mitochondrial Code
01244          23. Thraustochytrium Mitochondrial Code
01245          24. Pterobranchia Mitochondrial Code
01246          25. Candidate Division SR1 and Gracilibacteria Code
01247          26. Pachysolen tannophilus Nuclear Code
01248          """
01249
01250          # 1: The Standard Code
01251          self.gencode_NCBI_1 = {
01252          "TTT" : "F", # Phe
01253          "TCT" : "S", # Ser
01254          "TAT" : "Y", # Tyr
01255          "TGT" : "C", # Cys
01256          "TTC" : "F", # Phe
01257          "TCC" : "S", # Ser
01258          "TAC" : "Y", # Tyr
01259          "TGC" : "C", # Cys
01260          "TTA" : "L", # Leu
01261          "TCA" : "S", # Ser
01262          "TAA" : "*", # Ter
01263          "TGA" : "*", # Ter
01264          "TTG" : "L", # Leu i
01265          "TCG" : "S", # Ser
01266          "TAG" : "*", # Ter
01267          "TGG" : "W", # Trp
01268          "CTT" : "L", # Leu
01269          "CCT" : "P", # Pro
01270          "CAT" : "H", # His
01271          "CGT" : "R", # Arg
01272          "CTC" : "L", # Leu
01273          "CCC" : "P", # Pro
01274          "CAC" : "H", # His
01275          "CGC" : "R", # Arg
01276          "CTA" : "L", # Leu
01277          "CCA" : "P", # Pro
01278          "CAA" : "Q", # Gln
01279          "CGA" : "R", # Arg
01280          "CTG" : "L", # Leu i
01281          "CCG" : "P", # Pro
01282          "CAG" : "Q", # Gln
01283          "CGG" : "R", # Arg
01284          "ATT" : "I", # Ile
01285          "ACT" : "T", # Thr
01286          "AAT" : "N", # Asn
01287          "AGT" : "S", # Ser
01288          "ATC" : "I", # Ile
01289          "ACC" : "T", # Thr
01290          "AAC" : "N", # Asn
01291          "AGC" : "S", # Ser
01292          "ATA" : "I", # Ile
01293          "ACA" : "T", # Thr
01294          "AAA" : "K", # Lys
01295          "AGA" : "R", # Arg
01296          "ATG" : "M", # Met i
01297          "ACG" : "T", # Thr
01298          "AAG" : "K", # Lys
01299          "AGG" : "R", # Arg
01300          "GTT" : "V", # Val
01301          "GCT" : "A", # Ala
01302          "GAT" : "D", # Asp
01303          "GGT" : "G", # Gly
01304          "GTC" : "V", # Val
01305          "GCC" : "A", # Ala
01306          "GAC" : "D", # Asp
01307          "GGC" : "G", # Gly
01308          "GTA" : "V", # Val
01309          "GCA" : "A", # Ala
01310          "GAA" : "E", # Glu
01311          "GGA" : "G", # Gly
01312          "GTG" : "V", # Val
01313          "GCG" : "A", # Ala
01314          "GAG" : "E", # Glu
01315          "GGG" : "G", # Gly
01316          "---" : "-", # Gap
01317          "???" : "?", # Unk
```

```
01318         "NNN" : "X", # Unk
01319         }
01320
01321         # 2: The Vertebrate Mitochondrial Code
01322         self.gencode_NCBI_2 = self.gencode_NCBI_1.copy()
01323         self.gencode_NCBI_2["AGA"] = "*" # Ter
01324         self.gencode_NCBI_2["AGG"] = "*" # Ter
01325         self.gencode_NCBI_2["ATA"] = "M" # Met
01326         self.gencode_NCBI_2["TGA"] = "W" # Trp
01327
01328         # 3: The Yeast Mitochondrial Code
01329         self.gencode_NCBI_3 = self.gencode_NCBI_1.copy()
01330         self.gencode_NCBI_3["ATA"] = "M" # Met
01331         self.gencode_NCBI_3["CTT"] = "T" # Thr
01332         self.gencode_NCBI_3["CTC"] = "T" # Thr
01333         self.gencode_NCBI_3["CTA"] = "T" # Thr
01334         self.gencode_NCBI_3["CTG"] = "T" # Thr
01335         self.gencode_NCBI_3["TGA"] = "W" # Trp
01336
01337         del self.gencode_NCBI_3["CGA"]
01338         del self.gencode_NCBI_3["CGC"]
01339
01340         # 4: The Mold, Protozoan, and Coelenterate Mitochondrial Code and the Mycoplasma/Spiroplasma
    Code
01341         self.gencode_NCBI_4 = self.gencode_NCBI_1.copy()
01342         self.gencode_NCBI_4["TGA"] = "W" # Trp
01343
01344         # 5: The Invertebrate Mitochondrial Code
01345         self.gencode_NCBI_5 = self.gencode_NCBI_1.copy()
01346         self.gencode_NCBI_5["AGA"] = "S" # Ser
01347         self.gencode_NCBI_5["AGG"] = "S" # Ser
01348         self.gencode_NCBI_5["ATA"] = "M" # Met
01349         self.gencode_NCBI_5["TGA"] = "W" # Trp
01350
01351         # 6: The Ciliate, Dasycladacean and Hexamita Nuclear Code
01352         self.gencode_NCBI_6 = self.gencode_NCBI_1.copy()
01353         self.gencode_NCBI_6["TAA"] = "Q" # Gln
01354         self.gencode_NCBI_6["TAG"] = "Q" # Gln
01355
01356         # 9: The Echinoderm and Flatworm Mitochondrial Code
01357         self.gencode_NCBI_9 = self.gencode_NCBI_1.copy()
01358         self.gencode_NCBI_9["AAA"] = "N" # Asn
01359         self.gencode_NCBI_9["AGA"] = "S" # Ser
01360         self.gencode_NCBI_9["AGG"] = "S" # Ser
01361         self.gencode_NCBI_9["TGA"] = "W" # Trp
01362
01363         # 10: The Euplotid Nuclear Code
01364         self.gencode_NCBI_10 = self.gencode_NCBI_1.copy()
01365         self.gencode_NCBI_10["TGA"] = "C" # Cys
01366
01367         # 11: The Bacterial, Archaeal and Plant Plastid Code
01368         self.gencode_NCBI_11 = self.gencode_NCBI_1.copy()
01369
01370         # 12: The Alternative Yeast Nuclear Code
01371         self.gencode_NCBI_12 = self.gencode_NCBI_1.copy()
01372         self.gencode_NCBI_12["CTG"] = "S" # Ser
01373
01374         # 13: The Ascidian Mitochondrial Code
01375         self.gencode_NCBI_13 = self.gencode_NCBI_1.copy()
01376         self.gencode_NCBI_13["AGA"] = "G" # Gly
01377         self.gencode_NCBI_13["AGG"] = "G" # Gly
01378         self.gencode_NCBI_13["ATA"] = "M" # Met
01379         self.gencode_NCBI_13["TGA"] = "W" # Trp
01380
01381         # 14: The Alternative Flatworm Mitochondrial Code
01382         self.gencode_NCBI_14 = self.gencode_NCBI_1.copy()
01383         self.gencode_NCBI_14["AAA"] = "N" # Asn
01384         self.gencode_NCBI_14["AGA"] = "S" # Ser
01385         self.gencode_NCBI_14["AGG"] = "S" # Ser
01386         self.gencode_NCBI_14["TAA"] = "Y" # Tyr
01387         self.gencode_NCBI_14["TGA"] = "W" # Trp
01388
01389         # 16: Chlorophycean Mitochondrial Code
01390         self.gencode_NCBI_16 = self.gencode_NCBI_1.copy()
01391         self.gencode_NCBI_16["TAG"] = "L" # Leu
01392
01393         # 21: Trematode Mitochondrial Code
01394         self.gencode_NCBI_21 = self.gencode_NCBI_1.copy()
01395         self.gencode_NCBI_21["TGA"] = "W" # Trp
01396         self.gencode_NCBI_21["ATA"] = "M" # Met
01397         self.gencode_NCBI_21["AGA"] = "S" # Ser
01398         self.gencode_NCBI_21["AGG"] = "S" # Ser
01399         self.gencode_NCBI_21["AAA"] = "N" # Asn
01400
01401         # 22: Scenedesmus obliquus Mitochondrial Code
01402         self.gencode_NCBI_22 = self.gencode_NCBI_1.copy()
01403         self.gencode_NCBI_22["TCA"] = "*" # Ter
```

```
01404         self.gencode_NCBI_22["TAG"] = "L" # Leu
01405
01406         # 23: Thraustochytrium Mitochondrial Code
01407         self.gencode_NCBI_23 = self.gencode_NCBI_1.copy()
01408         self.gencode_NCBI_23["TTA"] = "*" # Ter
01409
01410         # 24: Pterobranchia Mitochondrial Code
01411         self.gencode_NCBI_24 = self.gencode_NCBI_1.copy()
01412         self.gencode_NCBI_24["AGA"] = "S" # Ser
01413         self.gencode_NCBI_24["AGG"] = "K" # Lys
01414         self.gencode_NCBI_24["TGA"] = "W" # Trp
01415
01416         # 25: Candidate Division SR1 and Gracilibacteria Code
01417         self.gencode_NCBI_25 = self.gencode_NCBI_1.copy()
01418         self.gencode_NCBI_25["TGA"] = "G" # Gly
01419
01420         # 26: Pachysolen tannophilus Nuclear Code
01421         self.gencode_NCBI_26 = self.gencode_NCBI_1.copy()
01422         self.gencode_NCBI_26["CTG"] = "A" # Ala
01423
01424         self.codes = {
01425         1 : self.gencode_NCBI_1,
01426         2 : self.gencode_NCBI_2,
01427         3 : self.gencode_NCBI_3,
01428         4 : self.gencode_NCBI_4,
01429         5 : self.gencode_NCBI_5,
01430         6 : self.gencode_NCBI_6,
01431         9 : self.gencode_NCBI_9,
01432         10 : self.gencode_NCBI_10,
01433         11 : self.gencode_NCBI_11,
01434         12 : self.gencode_NCBI_12,
01435         13 : self.gencode_NCBI_13,
01436         14 : self.gencode_NCBI_14,
01437         16 : self.gencode_NCBI_16,
01438         21 : self.gencode_NCBI_21,
01439         22 : self.gencode_NCBI_22,
01440         23 : self.gencode_NCBI_23,
01441         24 : self.gencode_NCBI_24,
01442         25 : self.gencode_NCBI_25,
01443         26 : self.gencode_NCBI_26
01444         }
01445
01446     def translate_dna_to_aa(self, seq, translation_table, frame):
01447         # translate DNA string into amino acids
01448         # where the last codon starts
01449         last_codon_start = len(seq) - 2
01450         # where the first codon starts
01451         if frame == 1:
01452             first = 0
01453         elif frame == 2:
01454             first = 1
01455         elif frame == 3:
01456             first = 2
01457         # create protein sequence by growing list
01458         protein = []
01459         add_to_protein = protein.append
01460         for start in range(first, last_codon_start, 3):
01461             codon = seq[start : start + 3]
01462             aa = translation_table.get(codon.upper(), 'X')
01463             add_to_protein(aa)
01464
01465         return "".join(protein)
01466
01467     def translate_dict(self, source_dict):
01468         translation_table = self.codes.get(self.genetic_code)
01469         translated_dict = {}
01470         for taxon, seq in sorted(source_dict.items()):
01471             translated_seq = self.translate_dna_to_aa(seq, translation_table, self.reading_frame)
01472             if "*" in translated_seq:
01473                 print("WARNING: stop codon(s), indicated as *, found in {} sequence".format(taxon))
01474             translated_dict[taxon] = translated_seq
01475
01476         return translated_dict
01477
01478     def get_translated(self, translation_table, reading_frame):
01479         if int(self.cores) == 1:
01480             translated_alignments = [self.translate_dict(alignment) for alignment in
    self.parsed_alignments]
01481         elif int(self.cores) > 1:
01482             pool = mp.Pool(int(self.cores))
01483             translated_alignments = pool.map(self.translate_dict, self.parsed_alignments)
01484
01485         return translated_alignments
01486
01487     def trim_dict(self, alignment):
01488         trim_vector = alignment.get_trim_selection(self.trim_fraction, self.parsimony_check)
01489         aln_dict = alignment.parsed_aln
```

```
01490         for key in aln_dict:
01491             aln_dict[key] = ''.join(list(compress(aln_dict[key], trim_vector)))
01492 
01493         return aln_dict
01494 
01495     def get_trimmed(self, trim_fraction, parsimony_check):
01496         if int(self.cores) == 1:
01497             trimmed_alignments = [self.trim_dict(alignment) for alignment in self.alignment_objects]
01498         elif int(self.cores) > 1:
01499             pool = mp.Pool(int(self.cores))
01500             trimmed_alignments = pool.map(self.trim_dict, self.alignment_objects)
01501 
01502         return trimmed_alignments
01503 
01504     def remove_unknown_chars(self, seq):
01505         # remove unknown characters from sequence
01506         new_seq = seq.replace("?", "").replace("-", "")
01507 
01508         return new_seq
01509 
01510     def remove_empty_sequences(self, split_alignment):
01511         # remove taxa from alignment if they are composed of only empty sequences
01512         new_alignment = {taxon : seq for taxon, seq in split_alignment.items() if
      self.remove_unknown_chars(seq)}
01513 
01514         return new_alignment
01515 
01516     def get_partitions(self, partitions_file):
01517         # parse and get partitions from partitions file
01518         partitions = FileParser(partitions_file)
01519         parsed_partitions = partitions.partitions_parse()
01520 
01521         return parsed_partitions
01522 
01523     def get_alignment_object(self, alignment):
01524         # parse according to the given alphabet;
01525         # Note:(`alignment`) <=> `in_file` outside MetaAlignment, e.g.
01526         #
      AminoAcidAlignment(Alignment<-.get_parsed_aln<-.get_aln_input)<-FileParser.__init__(in_file)<-FileHandler(...open(self.
01527         if self.data_type == "aa":
01528             aln = AminoAcidAlignment(alignment, self.in_format, self.data_type)
01529         elif self.data_type == "dna":
01530             aln = DNAAlignment(alignment, self.in_format, self.data_type)
01531         return aln
01532 
01533     def get_alignment_objects(self):
01534         # get alignment objects on which statistics can be computed
01535         # use multiprocessing if more than one core specified
01536         if int(self.cores) == 1:
01537             alignments = [self.get_alignment_object(alignment) for alignment in self.in_files]
01538         elif int(self.cores) > 1:
01539             pool = mp.Pool(int(self.cores))
01540             alignments = pool.map(self.get_alignment_object, self.in_files)
01541         return alignments
01542 
01543     def get_parsed_alignments(self):
01544         # get parsed dictionaries with taxa and sequences
01545         parsed_alignments = []
01546         add_to_parsed_alignments = parsed_alignments.append
01547         for alignment in self.alignment_objects:
01548             parsed = alignment.parsed_aln
01549             add_to_parsed_alignments(parsed)
01550             # checking if every seq has the same length or if parsed is not empty; exit if false
01551             if self.check_align:
01552                 equal = all(
01553                     x == [len(list(parsed.values())[i]) for i in
      range(0,len(list(parsed.values())))][0]
01554                     for x in [len(list(parsed.values())[i]) for i in
      range(0,len(list(parsed.values())))]
01555                 )
01556                 if equal is False:
01557                     print("ERROR: Sequences in input are of varying lengths. Be sure to align them
      first.")
01558                     sys.exit()
01559 
01560             if not parsed.keys() or not any(parsed.values()):
01561                 print(
01562                     "ERROR: Parsed sequences of " + alignment.in_file + " are empty. "
01563                     "Are you sure you specified the right input format and/or that input is a valid
      alignment?"
01564                 )
01565                 sys.exit()
01566 
01567         return parsed_alignments
01568 
01569     def get_partitioned(self, partitions_file):
01570         # partition alignment according to a partitions file
```

```
01571         partitions = self.get_partitions(partitions_file)
01572         alignment = self.parsed_alignments[0]
01573 
01574         # initiate list of newly partitioned alignments
01575         list_of_parts = []
01576         add_to_list_of_parts = list_of_parts.append
01577         for partition in partitions:
01578             # loop over all parsed partitions, adding taxa and sliced sequences
01579             for name, elements in partition.items():
01580                 new_dict = {}
01581 
01582                 for taxon, seq in alignment.items():
01583                     new_seq = ""
01584 
01585                     for dictionary in elements:
01586                         new_seq = new_seq +
      seq[dictionary["start"]:dictionary["stop"]:dictionary["stride"]]
01587                         new_dict[taxon] = new_seq
01588 
01589                 if self.remove_empty:
01590                     # check if remove empty sequences
01591                     no_empty_dict = self.remove_empty_sequences(new_dict)
01592                     add_to_list_of_parts({name : no_empty_dict})
01593                 else:
01594                     # add partition name : dict of taxa and sequences to the list
01595                     add_to_list_of_parts({name : new_dict})
01596 
01597         return list_of_parts
01598 
01599     def get_summaries(self):
01600         # get summaries for all alignment objects
01601 
01602         # define different headers for aa and dna alignments
01603         aa_header = [
01604             "Alignment_name",
01605             "No_of_taxa",
01606             "Alignment_length",
01607             "Total_matrix_cells",
01608             "Undetermined_characters",
01609             "Missing_percent",
01610             "No_variable_sites",
01611             "Proportion_variable_sites",
01612             "Parsimony_informative_sites",
01613             "Proportion_parsimony_informative"
01614         ]
01615 
01616         dna_header = [
01617             "Alignment_name",
01618             "No_of_taxa",
01619             "Alignment_length",
01620             "Total_matrix_cells",
01621             "Undetermined_characters",
01622             "Missing_percent",
01623             "No_variable_sites",
01624             "Proportion_variable_sites",
01625             "Parsimony_informative_sites",
01626             "Proportion_parsimony_informative",
01627             "AT_content",
01628             "GC_content"
01629         ]
01630 
01631         alignments = self.alignment_objects
01632         parsed_alignments = self.parsed_alignments
01633         freq_header = [char for char in alignments[0].alphabet]
01634 
01635         if self.data_type == "aa":
01636             header = aa_header + freq_header
01637         elif self.data_type == "dna":
01638             header = dna_header + freq_header
01639 
01640         # use multiprocessing if more than one core specified
01641         if int(self.cores) == 1:
01642             summaries = [alignment.get_summary() for alignment in alignments]
01643         elif int(self.cores) > 1:
01644             pool = mp.Pool(int(self.cores))
01645             summaries = pool.map(self.summarize_alignments, alignments)
01646         return header, summaries
01647 
01648     def summarize_alignments(self, alignment):
01649         # helper function to summarize alignments
01650         summary = alignment.get_summary()
01651         return summary
01652 
01653     def get_taxon_summaries(self):
01654         # get per-sequence summaries for all alignment objects
01655 
01656         # define different headers for aa and dna alignments
```

```
01657         aa_header = [
01658             "Alignment_name",
01659             "Taxon_name",
01660             "Sequence_length",
01661             "Undetermined_characters",
01662             "Missing_percent"
01663         ]
01664 
01665         dna_header = [
01666             "Alignment_name",
01667             "Taxon_name",
01668             "Sequence_length",
01669             "Undetermined_characters",
01670             "Missing_percent",
01671             "AT_content",
01672             "GC_content"
01673         ]
01674 
01675         alignments = self.alignment_objects
01676         parsed_alignments = self.parsed_alignments
01677         freq_header = alignments[0].alphabet
01678 
01679         if self.data_type == "aa":
01680             header = aa_header + freq_header
01681         elif self.data_type == "dna":
01682             header = dna_header + freq_header
01683 
01684         # use multiprocessing if more than one core specified
01685         if int(self.cores) == 1:
01686             summaries = [alignment.get_taxa_summary() for alignment in alignments]
01687         elif int(self.cores) > 1:
01688             pool = mp.Pool(int(self.cores))
01689             summaries = pool.map(self.summarize_alignments_taxa, alignments)
01690 
01691         return header, summaries
01692 
01693     def summarize_alignments_taxa(self, alignment):
01694         # helper function to summarize alignments by taxon
01695         summary = alignment.get_taxa_summary()
01696         return summary
01697 
01698     def write_summaries(self, file_name):
01699         # write summaries to file
01700 
01701         self.file_overwrite_error(file_name)
01702 
01703         with open(file_name, "w", encoding="utf-8") as summary_file:
01704             summary_out = self.get_summaries()
01705             header = '\t'.join(summary_out[0])
01706             new_summ = ['\t'.join(summary) for summary in summary_out[1]]
01707             summary_file.write(header + '\n')
01708             summary_file.write('\n'.join(new_summ))
01709             summary_file.write('\n')
01710             print("Wrote summaries to file '" + file_name + "'")
01711 
01712     def write_taxa_summaries(self):
01713         # write by-taxon summaries to file
01714         for index, in_file_name in enumerate(self.in_files):
01715             out_file_name = in_file_name + "-seq-summary.txt"
01716             self.file_overwrite_error(out_file_name)
01717             with open(out_file_name, "w", encoding="utf-8") as summary_file:
01718                 summary_out = self.get_taxon_summaries()
01719                 header = '\t'.join(summary_out[0])
01720                 summ = [[str(col) for col in element] for element in summary_out[1][index]]
01721                 new_summ = ['\t'.join(row) for row in summ]
01722                 summary_file.write(header + '\n')
01723                 summary_file.write('\n'.join(new_summ))
01724                 summary_file.write('\n')
01725 
01726     def get_replicate(self, no_replicates, no_loci):
01727         # construct replicate data sets for phylogenetic jackknife
01728         replicates = []
01729         add_to_replicates = replicates.append
01730         counter = 1
01731         for replicate in range(no_replicates):
01732 
01733             try:
01734                 random_alignments = sample(self.parsed_alignments, no_loci)
01735             except ValueError:
01736                 print("ERROR: You specified more loci per replicate than there are in your input.")
01737                 sys.exit()
01738 
01739             random_alignments = sample(self.parsed_alignments, no_loci)
01740             concat_replicate = self.get_concatenated(random_alignments)[0]
01741             add_to_replicates(concat_replicate)
01742             counter += 1
01743 
```

```
01744         return replicates
01745
01746     def get_concatenated(self, alignments):
01747         # concatenate muntiple input alignments
01748         # create empty dictionary of lists
01749         concatenated = defaultdict(list)
01750
01751         # first create list of taxa in all alignments
01752         # you need this to insert empty seqs in
01753         # the concatenated alignment
01754         all_taxa = []
01755         for alignment in alignments:
01756             for taxon in alignment.keys():
01757                 if taxon not in all_taxa:
01758                     all_taxa.append(taxon)
01759
01760         # start counters to keep track of partitions
01761         partition_counter = 1
01762         position_counter = 1
01763         # get dict for alignment name and partition
01764         partitions = {}
01765         digits_to_pad = len(str(len(alignments)))
01766
01767         for alignment in alignments:
01768             # get alignment length from a random taxon
01769             partition_length = len(alignment[list(alignment.keys())[0]])
01770             # get base name of each alignment for use when writing partitions file
01771             # NOTE: the base name here is whatever comes before fist period in the file name
01772             alignment_name = self.alignment_objects[partition_counter - 1].get_name().split('.')[0]
01773
01774             if self.using_metapartitions:
01775                 # implementation of option --no-mpan; option --prepend(-label) will assign a string or
      "" (see class definition)
01776                 if self.no_mpan:
01777                     # omit original alignment names from the printed partition file
01778                     partition_name = self.prepend_label + "p" +
      str(partition_counter).zfill(digits_to_pad)
01779                 else:
01780                     # keep original alignment names in the printed partition file
01781                     partition_name = self.prepend_label + "p" +
      str(partition_counter).zfill(digits_to_pad) + "_" + alignment_name
01782             else:
01783                 partition_name = "p" + str(partition_counter) + "_" + alignment_name
01784
01785             start = position_counter
01786             position_counter += partition_length
01787             end = position_counter - 1
01788             partitions[partition_name] = str(start) + "-" + str(end)
01789             partition_counter += 1
01790
01791             # get empty sequence if there is missing taxon
01792             # getting length from first element of list of keys
01793             # created from the original dict for this alignment
01794             empty_seq = '?' * partition_length
01795
01796             for taxon in all_taxa:
01797
01798                 if taxon not in alignment.keys():
01799                     concatenated[taxon].append(empty_seq)
01800                 else:
01801                     concatenated[taxon].append(alignment[taxon])
01802
01803         concatenated = {taxon:''.join(seqs) for taxon, seqs in concatenated.items()}
01804
01805         return concatenated, partitions
01806
01807     def remove_from_alignment(self, alignment, species_to_remove_set, index):
01808         # remove taxa from alignment
01809         aln_name = self.get_alignment_name_no_ext(index)
01810         for taxon in species_to_remove_set:
01811             if taxon not in alignment.keys():
01812                 print(
01813                     "WARNING: Taxon '" + taxon + "' not found in '" + aln_name + "'.\nIf you expected
      it to be there, "
01814                     "make sure to replace all taxon name spaces with underscores and that you are not
      using quotes."
01815                 )
01816         # originally within for-loop scope (redundancy)
01817         new_alignment = {species: seq for species, seq in alignment.items() if species not in
      species_to_remove_set}
01818
01819         return aln_name, new_alignment
01820
01821     def remove_taxa(self, species_to_remove_set):
01822         new_alns = {}
01823         for index, alignment in enumerate(self.parsed_alignments):
01824             aln_name, aln_dict = self.remove_from_alignment(alignment, species_to_remove_set, index)
```

```
01825             # check if alignment is not empty:
01826             if aln_dict:
01827                 new_alns[aln_name] = aln_dict
01828             else:
01829                 print("ERROR: You asked to remove all taxa from the alignment " + aln_name + ". No
      output file will be written.")
01830
01831         return new_alns
01832
01833     def print_fasta(self, source_dict):
01834         # print fasta-formatted string from a dictionary
01835         fasta_string = ""
01836         # each sequence line will have 80 characters
01837         n = 80
01838
01839         for taxon, seq in sorted(source_dict.items()):
01840             # split dictionary values to a list of string, each n chars long
01841             seq = [seq[i:i+n] for i in range(0, len(seq), n)]
01842             # in case there are unwanted spaces in taxon names
01843             taxon = taxon.replace(" ", "_").strip("'")
01844             fasta_string += ">" + taxon + "\n"
01845             for element in seq:
01846                 fasta_string += element + "\n"
01847
01848         return fasta_string
01849
01850     def print_phylip(self, source_dict):
01851         # print phylip-formatted string from a dictionary
01852         taxa_list = list(source_dict.keys())
01853         no_taxa = len(taxa_list)
01854         # figure out the max length of a taxon for nice padding of sequences
01855         pad_longest_name = len(max(taxa_list, key=len)) + 3
01856         # get sequence length from a random value
01857         seq_length = len(next(iter(source_dict.values())))
01858         header = str(len(source_dict)) + " " + str(seq_length)
01859         phylip_string = header + "\n"
01860         for taxon, seq in sorted(source_dict.items()):
01861             taxon = taxon.replace(" ", "_").strip("'")
01862             # left-justify taxon names relative to sequences
01863             phylip_string += taxon.ljust(pad_longest_name, ' ') + seq + "\n"
01864
01865         return phylip_string
01866
01867     def print_phylip_int(self, source_dict):
01868         # print phylip interleaved-formatted string from a dictionary
01869         taxa_list = list(source_dict.keys())
01870         no_taxa = len(taxa_list)
01871         pad_longest_name = len(max(taxa_list, key=len)) + 3
01872         seq_length = len(next(iter(source_dict.values())))
01873         header = str(len(source_dict)) + " " + str(seq_length)
01874         phylip_int_string = header + "\n\n"
01875         # this will be a list of tuples to hold taxa names and sequences
01876         seq_matrix = []
01877
01878         # each sequence line will have 500 characters
01879         n = 500
01880
01881         # recreate sequence matrix
01882         add_to_matrix = seq_matrix.append
01883         for taxon, seq in sorted(source_dict.items()):
01884             add_to_matrix((taxon, [seq[i:i+n] for i in range(0, len(seq), n)]))
01885
01886         first_seq = seq_matrix[0][1]
01887         for index, item in enumerate(first_seq):
01888             for taxon, sequence in seq_matrix:
01889                 if index == 0:
01890                     phylip_int_string += taxon.ljust(pad_longest_name, ' ') + sequence[index] + "\n"
01891                 else:
01892                     phylip_int_string += sequence[index] + "\n"
01893             phylip_int_string += "\n"
01894
01895         return phylip_int_string
01896
01897     def print_nexus(self, source_dict):
01898         # print nexus-formatted string from a dictionary
01899         if self.data_type == "aa" or self.command == "translate":
01900             data_type = "PROTEIN"
01901         elif self.data_type == "dna":
01902             data_type = "DNA"
01903
01904         taxa_list = list(source_dict.keys())
01905         no_taxa = len(taxa_list)
01906         pad_longest_name = len(max(taxa_list, key=len)) + 3
01907         seq_length = len(next(iter(source_dict.values())))
01908         header = str(len(source_dict)) + " " + str(seq_length)
01909         nexus_string = (
01910             "#NEXUS\n\nBEGIN DATA;\n\tDIMENSIONS  NTAX=" + str(no_taxa) + " NCHAR=" + str(seq_length)
```

```
01911             + ";\n\tFORMAT DATATYPE=" + data_type + "  GAP = - MISSING = ?;\n\tMATRIX\n"
01912         )
01913
01914         for taxon, seq in sorted(source_dict.items()):
01915             taxon = taxon.replace(" ", "_").strip("'")
01916             nexus_string += "\t" + taxon.ljust(pad_longest_name, ' ') + seq + "\n"
01917         nexus_string += "\n;\n\nEND;"
01918
01919         return nexus_string
01920
01921     def print_nexus_int(self, source_dict):
01922         # print nexus interleaved-formatted string from a dictionary
01923         if self.data_type == "aa":
01924             data_type = "PROTEIN"
01925         elif self.data_type == "dna":
01926             data_type = "DNA"
01927
01928         taxa_list = list(source_dict.keys())
01929         no_taxa = len(taxa_list)
01930         pad_longest_name = len(max(taxa_list, key=len)) + 3
01931         seq_length = len(next(iter(source_dict.values())))
01932         header = str(len(source_dict)) + " " + str(seq_length)
01933         # this will be a list of tuples to hold taxa names and sequences
01934         seq_matrix = []
01935         nexus_int_string = (
01936             "#NEXUS\n\nBEGIN DATA;\n\tDIMENSIONS  NTAX=" + str(no_taxa) + " NCHAR=" + str(seq_length)
01937             + ";\n\tFORMAT   INTERLEAVE" + "   DATATYPE=" + data_type + "  GAP = - MISSING =
    ?;\n\tMATRIX\n"
01938         )
01939         # each sequence line will have 500 characters
01940         n = 500
01941
01942         # recreate sequence matrix
01943         add_to_matrix = seq_matrix.append
01944         for taxon, seq in sorted(source_dict.items()):
01945             add_to_matrix((taxon, [seq[i:i+n] for i in range(0, len(seq), n)]))
01946
01947         first_seq = seq_matrix[0][1]
01948         for index, item in enumerate(first_seq):
01949             for taxon, sequence in seq_matrix:
01950                 if index == 0:
01951                     nexus_int_string += taxon.ljust(pad_longest_name, ' ') + sequence[index] + "\n"
01952                 else:
01953                     nexus_int_string += sequence[index] + "\n"
01954             nexus_int_string += "\n"
01955
01956         nexus_int_string += "\n;\n\nEND;"
01957
01958         return nexus_int_string
01959
01960     def natural_sort(self, a_list):
01961         # create a function that does 'human sort' on a list
01962         convert = lambda text: int(text) if text.isdigit() else text.lower()
01963         alphanum_key = lambda key: [convert(c) for c in re.split('([0-9]+)', key)]
01964         return sorted(a_list, key = alphanum_key)
01965
01966     def print_unspecified_partitions(self, data_type, codons):
01967         # print partitions for concatenated alignment
01968         part_string = ""
01969         part_dict = self.get_concatenated(self.parsed_alignments)[1]
01970         part_list = self.natural_sort(part_dict.keys())
01971
01972         if data_type == "dna":
01973             if codons == "none":
01974                 for key in part_list:
01975                     part_string += key + " = " + str(part_dict[key]) + "\n"
01976             elif codons == "12":
01977                 for key in part_list:
01978                     start, end = str(part_dict[key]).split("-")
01979                     part_string += key + "_pos1" + " = " + start + "-" + end + "\\2" + "\n"
01980                     part_string += key + "_pos2" + " = " + str(int(start) + 1) + "-" + end + "\\2" +
    "\n"
01981             elif codons == "123":
01982                 for key in part_list:
01983                     start, end = str(part_dict[key]).split("-")
01984                     part_string += key + "_pos1" + " = " + start + "-" + end + "\\3" + "\n"
01985                     part_string += key + "_pos2" + " = " + str(int(start) + 1) + "-" + end + "\\3" +
    "\n"
01986                     part_string += key + "_pos3" + " = " + str(int(start) + 2) + "-" + end + "\\3" +
    "\n"
01987
01988         elif data_type == "aa":
01989             for key in part_list:
01990                 part_string += key + " = " + str(part_dict[key]) + "\n"
01991
01992         return part_string
01993
```

```
01994     def print_nexus_partitions(self, data_type, codons):
01995         # print partitions for concatenated alignment
01996         part_string = ""
01997         part_dict = self.get_concatenated(self.parsed_alignments)[1]
01998         part_list = self.natural_sort(part_dict.keys())
01999         # write beginning of nexus sets
02000         part_string += "#NEXUS\n\n"
02001         part_string += "BEGIN SETS;\n"
02002 
02003         if data_type == "dna":
02004             if codons == "none":
02005                 for key in part_list:
02006                     part_string += "\tcharset " + key + " = " + str(part_dict[key]) + ";\n"
02007             elif codons == "12":
02008                 for key in part_list:
02009                     start, end = str(part_dict[key]).split("-")
02010                     part_string += "\tcharset " + key + "_pos1" + " = " + start + "-" + end + "\\2" +
      ";\n"
02011                     part_string += "\tcharset " + key + "_pos2" + " = " + str(int(start) + 1) + "-" +
      end + "\\2" + ";\n"
02012             elif codons == "123":
02013                 for key in part_list:
02014                     start, end = str(part_dict[key]).split("-")
02015                     part_string += "\tcharset " + key + "_pos1" + " = " + start + "-" + end + "\\3" +
      ";\n"
02016                     part_string += "\tcharset " + key + "_pos2" + " = " + str(int(start) + 1) + "-" +
      end + "\\3" + ";\n"
02017                     part_string += "\tcharset " + key + "_pos3" + " = " + str(int(start) + 2) + "-" +
      end + "\\3" + ";\n"
02018             part_string += "END;"
02019 
02020         elif data_type == "aa":
02021             for key in part_list:
02022                 part_string += "\tcharset " + key + " = " + str(part_dict[key]) + ";\n"
02023             part_string += "END;"
02024 
02025         return part_string
02026 
02027     def print_iqtree_nexus_partitions(self, data_type, codons):
02028         # print partitions for concatenated alignment
02029         part_string = ""
02030         part_dict = self.get_concatenated(self.parsed_alignments)[1]
02031         part_list = self.natural_sort(part_dict.keys())
02032         # write beginning of nexus sets
02033         part_string += "#nexus\n"
02034         part_string += "begin sets;\n"
02035 
02036         if data_type == "dna":
02037             if codons == "none":
02038                 for key in part_list:
02039                     part_string += "  charset " + key + " = " + str(part_dict[key]) + ";\n"
02040             elif codons == "12":
02041                 for key in part_list:
02042                     start, end = str(part_dict[key]).split("-")
02043                     part_string += "  charset " + key + "_pos1" + " = " + start + " - " + end + "\\2"
      + ";\n"
02044                     part_string += "  charset " + key + "_pos2" + " = " + str(int(start) + 1) + " - "
      + end + "\\2" + ";\n"
02045             elif codons == "123":
02046                 for key in part_list:
02047                     start, end = str(part_dict[key]).split("-")
02048                     part_string += "  charset " + key + "_pos1" + " = " + start + " - " + end + "\\3"
      + ";\n"
02049                     part_string += "  charset " + key + "_pos2" + " = " + str(int(start) + 1) + " - "
      + end + "\\3" + ";\n"
02050                     part_string += "  charset " + key + "_pos3" + " = " + str(int(start) + 2) + " - "
      + end + "\\3" + ";\n"
02051             part_string += "end;\n"
02052 
02053         elif data_type == "aa":
02054             for key in part_list:
02055                 part_string += "  charset " + key + " = " + str(part_dict[key]) + ";\n"
02056             part_string += "end;\n"
02057 
02058         return part_string
02059 
02060     def print_raxml_partitions(self, data_type, codons):
02061         # print partitions for concatenated alignment
02062         part_string = ""
02063         part_dict = self.get_concatenated(self.parsed_alignments)[1]
02064         part_list = self.natural_sort(part_dict.keys())
02065 
02066         if data_type == "dna":
02067             if codons == "none":
02068                 for key in part_list:
02069                     part_string += "DNA, " + key + " = " + str(part_dict[key]) + "\n"
02070             elif codons == "12":
```

```
02071                 for key in part_list:
02072                     start, end = str(part_dict[key]).split("-")
02073                     part_string += "DNA, " + key + "_pos1" + " = " + start + "-" + end + "\\2" + "\n"
02074                     part_string += "DNA, " + key + "_pos2" + " = " + str(int(start) + 1) + "-" + end +
    "\\2" + "\n"
02075             elif codons == "123":
02076                 for key in part_list:
02077                     start, end = str(part_dict[key]).split("-")
02078                     part_string += "DNA, " + key + "_pos1" + " = " + start + "-" + end + "\\3" + "\n"
02079                     part_string += "DNA, " + key + "_pos2" + " = " + str(int(start) + 1) + "-" + end +
    "\\3" + "\n"
02080                     part_string += "DNA, " + key + "_pos3" + " = " + str(int(start) + 2) + "-" + end +
    "\\3" + "\n"
02081
02082         elif data_type == "aa":
02083             for key in part_list:
02084                 part_string += "WAG, " + key + " = " + str(part_dict[key]) + "\n"
02085
02086             #  aa-partition files with strides are probably not useful? (original below)
02087 #             elif codons == "12":
02088 #                 for key in part_list:
02089 #                     start, end = str(part_dict[key]).split("-")
02090 #                     part_string += "WAG, " + key + "_pos1" + " = " + start + "-" + end + "\\2" + "\n"
02091 #                     part_string += "WAG, " + key + "_pos2" + " = " + str(int(start) + 1) + "-" + end
    + "\\2" + "\n"
02092 #             elif codons == "123":
02093 #                 for key in part_list:
02094 #                     start, end = str(part_dict[key]).split("-")
02095 #                     part_string += "WAG, " + key + "_pos1" + " = " + start + "-" + end + "\\3" + "\n"
02096 #                     part_string += "WAG, " + key + "_pos2" + " = " + str(int(start) + 1) + "-" + end
    + "\\3" + "\n"
02097 #                     part_string += "WAG, " + key + "_pos3" + " = " + str(int(start) + 2) + "-" + end
    + "\\3" + "\n"
02098         return part_string
02099
02100     def replace_string_in_file(self, file_name, old_string, new_string):
02101         # global string replacement in file
02102         with open(file_name, "r", encoding="utf-8") as file:
02103             file_content = file.read()
02104         # write globally replaced content back to file
02105         glb_replaced_content = file_content.replace(old_string, new_string)
02106         with open(file_name, "w", encoding="utf-8") as file:
02107             file.write(glb_replaced_content)
02108
02109     def write_partitions(self, file_name, part_format, data_type, codons):
02110         # write partitions file for concatenated alignment
02111         self.file_overwrite_error(file_name)
02112         with open(file_name, "w", encoding="utf-8") as part_file:
02113             if part_format == "nexus":
02114                 part_file.write(self.print_nexus_partitions(data_type, codons))
02115             if part_format == "iqtree-nexus":
02116                 part_file.write(self.print_iqtree_nexus_partitions(data_type, codons))
02117             if part_format == "raxml":
02118                 part_file.write(self.print_raxml_partitions(data_type, codons))
02119             if part_format == "unspecified":
02120                 part_file.write(self.print_unspecified_partitions(data_type, codons))
02121
02122             if self.using_metapartitions:
02123                 self.replace_string_in_file(file_name, '-meta =', ' =')
02124
02125         print("Wrote partitions for the concatenated file to '" + file_name + "'")
02126
02127     def get_extension(self, file_format):
02128         # get proper extension string
02129         if file_format == "phylip":
02130             extension = "-out.phy"
02131         elif file_format == "phylip-int":
02132             extension = "-out.int-phy"
02133         elif file_format == "fasta":
02134             extension = "-out.fas"
02135         elif file_format == "nexus":
02136             extension = "-out.nex"
02137         elif file_format == "nexus-int":
02138             extension = "-out.int-nex"
02139
02140         return extension
02141
02142     def get_metapartition_extension(self, file_format):
02143         # get proper metapartition_extension string
02144         if file_format == "phylip":
02145             metapartition_extension = "-meta.phy"
02146         elif file_format == "phylip-int":
02147             metapartition_extension = "-meta.int-phy"
02148         elif file_format == "fasta":
02149             metapartition_extension = "-meta.fas"
02150         elif file_format == "nexus":
02151             metapartition_extension = "-meta.nex"
```

```
02152         elif file_format == "nexus-int":
02153             metapartition_extension = "-meta.int-nex"
02154 
02155         return metapartition_extension
02156 
02157     def file_overwrite_error(self, file_name):
02158         # print warning when overwriting a file
02159         if path.exists(file_name):
02160             print("WARNING: You are overwriting '" + file_name + "'")
02161 
02162     def write_formatted_file(self, file_format, file_name, alignment):
02163         # write the correct format string into a file
02164         with open(file_name, "w", encoding="utf-8") as out_file:
02165             if file_format == "phylip":
02166                 out_file.write(self.print_phylip(alignment))
02167             elif file_format == "fasta":
02168                 out_file.write(self.print_fasta(alignment))
02169             elif file_format == "phylip-int":
02170                 out_file.write(self.print_phylip_int(alignment))
02171             elif file_format == "nexus":
02172                 out_file.write(self.print_nexus(alignment))
02173             elif file_format == "nexus-int":
02174                 out_file.write(self.print_nexus_int(alignment))
02175 
02176     def get_alignment_name(self, i, extension):
02177         # get file name
02178         file_name = self.alignment_objects[i].get_name() + extension
02179 
02180         return file_name
02181 
02182     def get_alignment_name_no_ext(self, i):
02183         # get file name without extension
02184         file_name = self.alignment_objects[i].get_name()
02185 
02186         return file_name
02187 
02188     def write_concat(self, file_format):
02189         # write concatenated alignment into a file
02190         concatenated_alignment = self.get_concatenated(self.parsed_alignments)[0]
02191         file_name = self.concat_out
02192         self.file_overwrite_error(file_name)
02193         self.write_formatted_file(file_format, file_name, concatenated_alignment)
02194 
02195         print("Wrote concatenated sequences to " + file_format + " file '" + file_name + "'")
02196 
02197     def write_convert(self, index, alignment, file_format, extension):
02198         # write converted alignment into a file
02199         file_name = self.get_alignment_name(index, extension)
02200         self.file_overwrite_error(file_name)
02201         self.write_formatted_file(file_format, file_name, alignment)
02202 
02203     def write_replicate(self, index, alignment, file_format, extension):
02204         # write replicate alignment into a file
02205         file_name = "replicate" + str(index + 1) + "_" + str(self.no_loci) + "-loci" + extension
02206         self.file_overwrite_error(file_name)
02207         self.write_formatted_file(file_format, file_name, alignment)
02208 
02209     def write_split(self, item, file_format, extension):
02210         # write split alignments from partitions file
02211         # bad practice with the dicts; figure out better solution
02212         partition_name = list(item.keys())[0]
02213         alignment = item[partition_name]
02214 
02215         if not alignment:
02216             # If the alignment dict is empty, i.e. no alignment associated with partition name, raise
      error
02217             raise ValueError("Partition '%s' is empty. No sequences to write." % partition_name)
02218 
02219         # implementation of option --no-san (don't prepend input superalignment filename to the
      `split' outputs)
02220         if self.no_sup_aln_name:
02221             file_name = partition_name + extension
02222         else:
02223             file_name = str(self.in_files[0].split('.')[0]) + "_" + partition_name + extension
02224 
02225         try:
02226             self.file_overwrite_error(file_name)
02227             self.write_formatted_file(file_format, file_name, alignment)
02228             yield file_name
02229         except ValueError as e:
02230             print("There was an issue writing file '%s': %s" % (file_name, str(e)))
02231             remove(file_name)
02232             raise
02233 
02234     def write_reduced(self, file_format, extension):
02235         # write alignment with taxa removed into a file
02236         prefix = self.reduced_file_prefix
```

```
02237         alns = self.remove_taxa(self.species_to_remove)
02238         for file_name, aln_dict in alns.items():
02239             out_file_name = prefix + file_name + extension
02240             self.file_overwrite_error(out_file_name)
02241             self.write_formatted_file(file_format, out_file_name, aln_dict)
02242         return len(alns)
02243
02244     def write_translated(self, index, alignment, file_format, extension):
02245         # write alignments translated into amino acids
02246         prefix = "translated_"
02247         file_name = self.get_alignment_name(index, extension)
02248         out_file_name = prefix + file_name + extension
02249         self.file_overwrite_error(out_file_name)
02250         self.write_formatted_file(file_format, out_file_name, alignment)
02251
02252     def write_trimmed(self, index, alignment, file_format, extension):
02253         # write trimmed alignments
02254         if self.trim_out:
02255             out_file_name = self.trim_out
02256         else:
02257             prefix = "trimmed_"
02258             file_name = self.get_alignment_name(index, extension)
02259             out_file_name = prefix + file_name
02260         self.file_overwrite_error(out_file_name)
02261         self.write_formatted_file(file_format, out_file_name, alignment)
02262
02263     def write_metapartitions(self, file_format):
02264         # write metapartitions - combines split and concat
02265         print("write_out elif action == metapartitions")
02266         metapartition_extension = self.get_metapartition_extension(file_format)
02267         list_of_alignments = self.get_partitioned(self.split)
02268         written_split_files = []
02269         err_indx = 0
02270
02271         for item in list_of_alignments:
02272             try:
02273                 for split_file in self.write_split(item, file_format, metapartition_extension):
02274                     written_split_files.append(split_file)
02275             except ValueError as e:
02276                     print("WARNING: ", e)
02277                     err_indx += 1
02278         if len(written_split_files) > 0:
02279             print("Wrote %d %s metapartition files from partitions provided" %
      (len(written_split_files), file_format))
02280         if err_indx > 0:
02281             print("WARNING: %d file(s) raised an error while writing (see above)." % err_indx)
02282
02283         # now set inputs to be the collated metapartition alignment files
02284         self.in_files = written_split_files
02285         self.alignment_objects = self.get_alignment_objects()
02286         self.parsed_alignments = self.get_parsed_alignments()
02287
02288         # concat metapartition alignment files
02289         self.write_concat(file_format)
02290
02291     def write_out(self, action, file_format):
02292         # write other output files depending on command (action)
02293         extension = self.get_extension(file_format)
02294
02295         if action == "concat":
02296             self.write_concat(file_format)
02297
02298         elif action == "convert":
02299             length = len(self.alignment_objects)
02300             [
02301                 self.write_convert(i, alignment, file_format, extension)
02302                 for i, alignment in enumerate(self.parsed_alignments)
02303             ]
02304             print("Converted " + str(length) + " files from " + self.in_format + " to " + file_format)
02305
02306         elif action == "replicate":
02307             [
02308                 self.write_replicate(i, alignment, file_format, extension)
02309                 for i, alignment in enumerate(self.get_replicate(self.no_replicates, self.no_loci))
02310             ]
02311
02312             print("Constructed " + str(self.no_replicates) + " replicate data sets, each from " +
      str(self.no_loci) + " alignments")
02313
02314         elif action == "split":
02315             list_of_alignments = self.get_partitioned(self.split)
02316             written_split_files = []
02317             err_indx = 0
02318
02319             for item in list_of_alignments:
02320                 try:
02321                     for split_file in self.write_split(item, file_format, extension):
```

```
02322                         written_split_files.append(split_file)
02323                 except ValueError as e:
02324                         print("WARNING: ", e)
02325                         err_indx += 1
02326             if len(written_split_files) > 0:
02327                 print("Wrote %d %s files from partitions provided" % (len(written_split_files),
      file_format))
02328             if err_indx > 0:
02329                 print("WARNING: %d file(s) raised an error while writing (see above)." % err_indx)
02330 
02331         elif action == "metapartitions":
02332             self.write_metapartitions(file_format)
02333 
02334         elif action == "remove":
02335             aln_no = self.write_reduced(file_format, extension)
02336             if aln_no:
02337                 print("Wrote " + str(aln_no) + " " + str(file_format) + " files with reduced taxon
      set")
02338 
02339         elif action == "translate":
02340             if self.data_type == "aa":
02341                 print("ERROR: cannot translate; you said your alignment already contains amino acids")
02342                 sys.exit()
02343             translated_alignment_dicts = self.get_translated(self.genetic_code, self.reading_frame)
02344             length = len(self.alignment_objects)
02345             [
02346                 self.write_translated(i, alignment, file_format, extension)
02347                 for i, alignment in enumerate(translated_alignment_dicts)
02348             ]
02349             print("Translated " + str(length) + " files to amino acid sequences")
02350 
02351         elif action == "trim": # self.trim_fraction, self.parsimony_check
02352             trimmed_alignment_dicts = self.get_trimmed(self.trim_fraction, self.parsimony_check)
02353             length = len(self.alignment_objects)
02354             [
02355                 self.write_trimmed(i, alignment, file_format, extension)
02356                 for i, alignment in enumerate(trimmed_alignment_dicts)
02357             ]
02358             print("Trimmed", str(length), "file(s) to have", self.trim_fraction, "minimum occupancy
      per alignment column")
02359 
02360 
02361 def main():
02362 
02363     # initialize parsed arguments and meta alignment objects
02364     kwargs = run()
02365     meta_aln = MetaAlignment(**kwargs)
02366 
02367     if meta_aln.command == "summary":
02368         meta_aln.write_summaries(kwargs["summary_out"])
02369     if meta_aln.by_taxon_summary:
02370         print("Printing taxon summaries")
02371         meta_aln.write_taxa_summaries()
02372     if meta_aln.command == "convert":
02373         meta_aln.write_out("convert", kwargs["out_format"])
02374     if meta_aln.command == "concat":
02375         meta_aln.write_out("concat", kwargs["out_format"])
02376         meta_aln.write_partitions(kwargs["concat_part"], kwargs["part_format"], kwargs["data_type"],
      kwargs["codons"])
02377     if meta_aln.command == "replicate":
02378         meta_aln.write_out("replicate", kwargs["out_format"])
02379     if meta_aln.command == "split":
02380         meta_aln.write_out("split", kwargs["out_format"])
02381     if meta_aln.command == "remove":
02382         meta_aln.write_out("remove", kwargs["out_format"])
02383     if meta_aln.command == "translate":
02384         meta_aln.write_out("translate", kwargs["out_format"])
02385     if meta_aln.command == "trim":
02386         meta_aln.write_out("trim", kwargs["out_format"])
02387 
02388     if meta_aln.command == "metapartitions":
02389         # `metapartitions` is essentially `split` + `concat`. Currently you can't set an out_format:
02390         # it's automatically set to match the in_format because the intermediate `split` outputs
      become
02391         # the 'new' in_files for the `concat` operation, and then calling either:
02392         #  -> AminoAcidAlignment(Alignment.__init__(self, in_file, in_format, data_type))
02393         #  -> DNAAlignment(Alignment.__init__(self, in_file, in_format, data_type))
02394         # through MetaAlignment.get_alignment_object(alignment, self.in_format, self.data_type)
02395         meta_aln.write_out("metapartitions", kwargs["in_format"])
02396         meta_aln.write_partitions(kwargs["concat_part"], kwargs["part_format"], kwargs["data_type"],
      "none")
02397 
02398         # meta_aln.write_out("translate", kwargs["out_format"])
02399 
02400 def run():
02401 
02402     # initialize parsed arguments
```

```
02403     config = ParsedArgs()
02404     # get arguments
02405     config_dict = config.get_args_dict()
02406     return config_dict
02407
02408 if __name__ == '__main__':
02409     main()
```

8.5 md/README.md File Reference

Index