synthaser

Release 1.0

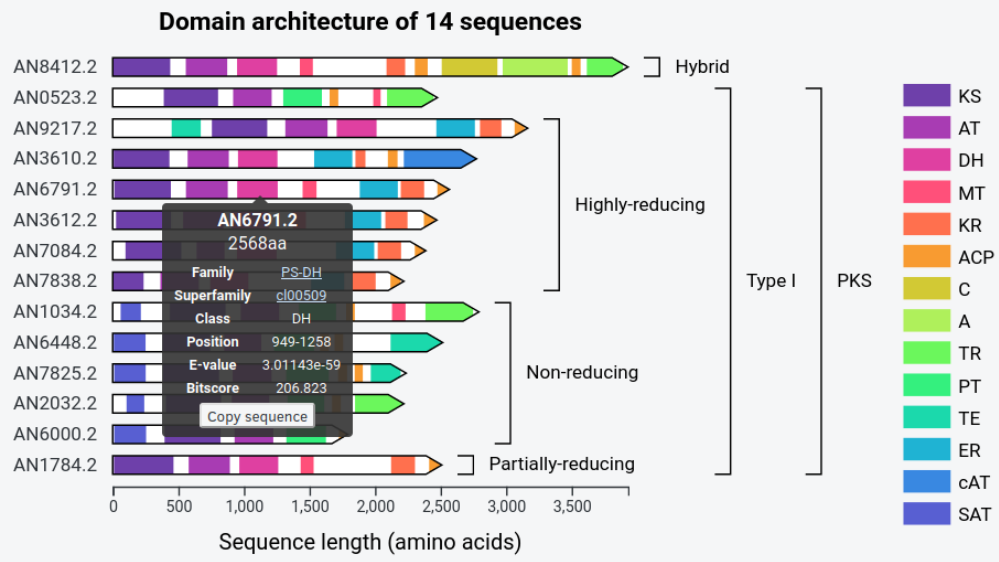
Oct 13, 2021

Contents

Welcome to synthaser's documentation!

synthaser is a tool for the characterisation and classificiation of the domain architecture of protein sequences. It leverages the NCBI's conserved domain (CD) search tool, CD-Search, for comphrehensive, fully remote domain searches, and automatically characterises and classifies sequences based on user-defined rules.

To view an example of what synthaser can produce, click the image below:



CHAPTER 1

Features

- Fully remote searches using the NCBI's CD-Search API
- Automatic characterisation and classification of protein sequences based on user-defined rules
- Fully interactive visualisations which can be tweaked and saved to SVG

CHAPTER 2

User guide

Get started using synthaser.

2.1 User guide

2.1.1 Installation

This section of the documentation covers the installation of *synthaser*.

Python version

`synthaser` is written using Python 3, and should work with any version above 3.3.

Dependencies

These packages are automatically installed when installing cblaster:

- requests
- biopython

Other dependencies

- RPS-BLAST is the search tool used in local cblaster searches
- rpsbproc is used to post-process RPS-BLAST results to remove redundant hits and fill in information about domain families like in the web CD-Search tool

Installation

1. (Optional) Create a new virtual environment

```
python3 -m virtualenv venv
source venv/bin/activate
```

This will create (and activate) a sandboxed environment where you can install Python packages separately to those available on your system. This isn't necessarily required, but is recommended.

2. Install `synthaser`

The easiest way to obtain `synthaser` is to install it directly from PyPI using `pip`:

```
pip install synthaser
```

This will install `synthaser`, as well as all of its required dependencies. Alternatively, you could clone the cblaster repository from GitHub and install it like so:

```
git clone https://www.github.com/gamcil/synthaser
cd synthaser
pip install .
```

This will download the latest version of cblaster and install it from the downloaded folder, rather than from PyPI.

`synthaser` should now be available directly on your terminal:

```
$ synthaser -h
usage: synthaser [-h] [--version] {getdb,getseq,search} ...
synthaser: a Python toolkit for analysing domain architecture of secondary metabolite␣
↪megasynth (et) ases with NCBI CD-Search.

positional arguments:
  {getdb,getseq,search}
    getdb               Download a CDD database for local searches
    getseq              Download sequences from NCBI
    search              Run a synthaser search

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit

Cameron L.M. Gilchrist 2020
```

Installing RPS-BLAST and rpsbproc

`RPS-BLAST` is a distributed in the NCBI's BLAST+ toolkit. This can be acquired either directly from NCBI's FTP or from your distributions repositories, for example in Ubuntu: `sudo apt install ncbi-blast+`.

To install `rpsbproc`, follow these steps:

1. Acquire the relevant archive for your system from the CDD FTP.

2. Extract the contents

3. Acquire the data files required by rpsbproc either by running `utils/getcdddata.sh`, or directly from the FTP as detailed by the README (see: domain-annotation files). The program does NOT require you to download all of the domain databases. So, if doing the former, you can chttps://www.circles.life/au/plan/ancel the run after the necessary files are in `data/`, then delete `db/` and the database `.tar.gz` files.

4. Make sure the rpsbproc binary file is on your system $PATH. This is a requirement of `synthaser`, as it will throw an error if it cannot find `rpsbproc` directly on the $PATH (i.e. accessable in terminal just by typing 'rpsbproc').

2.1.2 Quickstart

This section of the documentation gives a brief overview of how to get started using *synthaser* and a broad overview of its features.

Pre-search configuration using the `config` module

The NCBI requires that you provide some identification before using their services in order to prevent abuse. This can be an e-mail address, or more recently, an API key (https://ncbiinsights.ncbi.nlm.nih.gov/2017/11/02/new-api-keys-for-the-e-utilities/).

You can use the `config` module to set these parameters for `synthaser` searches (you'll only have to do this once!). This module will save a file, `config.ini`, wherever your operating system stores configuration files (for example, in Linux it will be saved in ~/.local/config/synthaser). When you run remote searches in `synthaser`, it will first check to see if it can find this file, and then if an e-mail address or API key is saved; if they are not found, `synthaser` will throw an error.

To set an e-mail address:

```
$ synthaser config --email "foo@bar.com"
```

. . . or an API key:

```
$ synthaser config --api_key <your API key>
```

Running a synthaser search using the `search` module

Once configured, a synthaser search is as easy as:

```
synthaser search --query_file sequences.fasta
```

This will take all sequences in `sequences.fasta` and start a remote CD-Search run.

Note: Most of the arguments used with synthaser have shorter forms which can be found in the help menus (e.g. `synthaser search --help`). For example, the short version of `--query_file` here is `-qf`.

Alternatively, you can use the `-qi/--query_ids` argument to start a search using just a collection of NCBI identifiers. For example:

```
synthaser search -qi KAF4294870.1 KAF4294328.1 KAF4293514.1
```

This will retrieve each sequence and start a remote CD-Search run.

There are several optional arguments you can use which control the output synthaser will generate. To generate a synthaser plot of your sequences, add the `-p/--plot` argument, optionally specifying a name to generate a portable HTML file:

```
synthaser search -qf sequences.fasta -p <plot.html>
```

The plot will then open directly in your web browser.

You can save a file containing the data of a synthaser search by using the `-json, --json_file` argument:

```
synthaser search -qf sequences.fasta -json session.json
```

This is particularly useful in larger searches, as the search won't have to be completely redone if you want to e.g. generate a new visualisation or output table. If the file specified does not exist, synthaser will create it; if it does exist, synthaser will attempt to load it.

The default output will show your query sequences and their domain architectures, grouped by their classifications. For example, the search using `--query_ids` from above produces:

```
PKS --> Type I --> Partially-reducing
--------------------------------------
KAF4294870.1    KS-AT-DH-MT-KR-ACP

Thiolase
--------
KAF4293514.1    KS
KAF4294328.1    KS
```

You can make this tabular by using the `-lf, --long_form` argument, which will produce a comma-separated output like:

```
Synthase,Length (aa),Architecture,Classification
KAF4294870.1,2445,KS-AT-DH-MT-KR-ACP,PKS|Type I|Partially-reducing
KAF4294328.1,413,KS,Thiolase
KAF4293514.1,419,KS,Thiolase
```

Where each row contains the sequence, its length, domain architecture and classification. This can then be directly imported into spreadsheet software.

Another very useful argument is `--cdsid`. This allows you to resume or load a CD-Search run at a later time. The CDSID (CD-Search identifier) is reported by synthaser at the start of every search, and takes the form:

`QM3-qcdsearch-XXXXXXXXXXXXXXX-YYYYYYYYYYYYYYY`

For example, in the output of the above search:

```
[14:57:52] INFO - Starting synthaser
[14:57:56] INFO - Launching new CD-Search run
[14:57:58] INFO - Run ID: QM3-qcdsearch-894E2B07233244A-1C6342BEDF36CB85
```

When I then wanted the tabular output, I could simply re-use the CDSID:

```
synthaser search \
        --query_ids KAF4294870.1 KAF4294328.1 KAF4293514.1 \
        --cdsid QM3-qcdsearch-894E2B07233244A-1C6342BEDF36CB85 \
        --long_form
```

CD-Search parameters can be altered using the following arguments:

Argument	Description
`--cdsid`	CD-Search run ID (e.g. `QM3-qcdsearch-XXXXXXXXXXXXXXX-YYYYYYYYYYYYYYY`).
`--smode`	Search mode (auto, prec or live; def. auto)
`--useid1`	Look for sequences in archival database if not found (def. true)
`--compbasedadj`	Use composition-corrected scoring (def. 1)
`--filter`	Filter out compositionally biased regions from queries (def. true)
`--evalue`	Maximum E-value (def. 3)
`--maxhit`	Maximum number of hits to return (def. 500)
`--dmode`	Level of hits to report (full, rep or std; def. full)

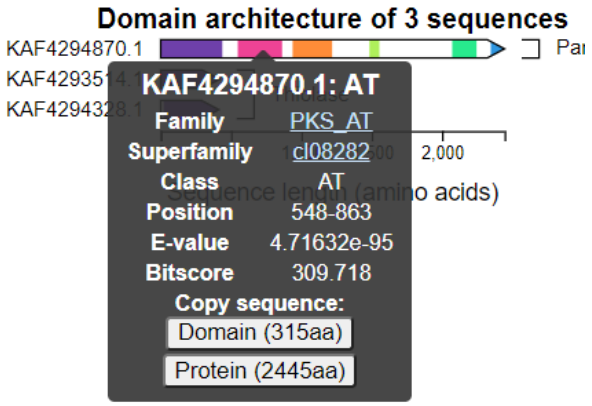
For a fuller explanation of these arguments, see the NCBI documentation here.

The synthaser plot

The synthaser plot is fairly intuitive, but deserves a little explanation of its own. Here is what the search above looks like plotted:

Query sequences are drawn to scale, with coloured boxes representing the domains that were found. Sequences are grouped by their classifications, and annotation bars for classifications are drawn in the appropriate locations automatically. This looks a not nicer when you have more sequences covering more classifications!

Hovering over a domain box in a sequence will show a tooltip menu that looks like:



This shows you a summary of the best domain family hit, with linkouts to their corresponding entries in the NCBI databases. You can also copy amino acid sequences of either the entire query or just the specific domain by clicking the buttons at the bottom.

The legend contains a list of the domains found in all of your sequences next to a box with the corresponding colour. This colour can be changed simply by clicking the box and selecting another colour.

Sequences can be hidden by clicking on their names. If a sequence containing the last occurrence of a specific domain in the plot is hidden, that domain will automatically be removed from the legend as well.

Sizing and positioning of plot elements can be controlled by the settings in the dropdown menu on the right hand side of the plot. In the above image, the only change from default was the maximum sequence length (in pixels); by default, this is set to 600 px.

You can generate FASTA files containing extracted sequences of specific domain types from your sequences using the *Download domain sequences* section at the bottom of the dropdown menu. Simply select a domain type and click the *Download!* button.

Once you are happy with your figure, you can download a SVG image file by clicking the *Save SVG* button at the top of the menu.

2.1.3 Creating custom rule sets

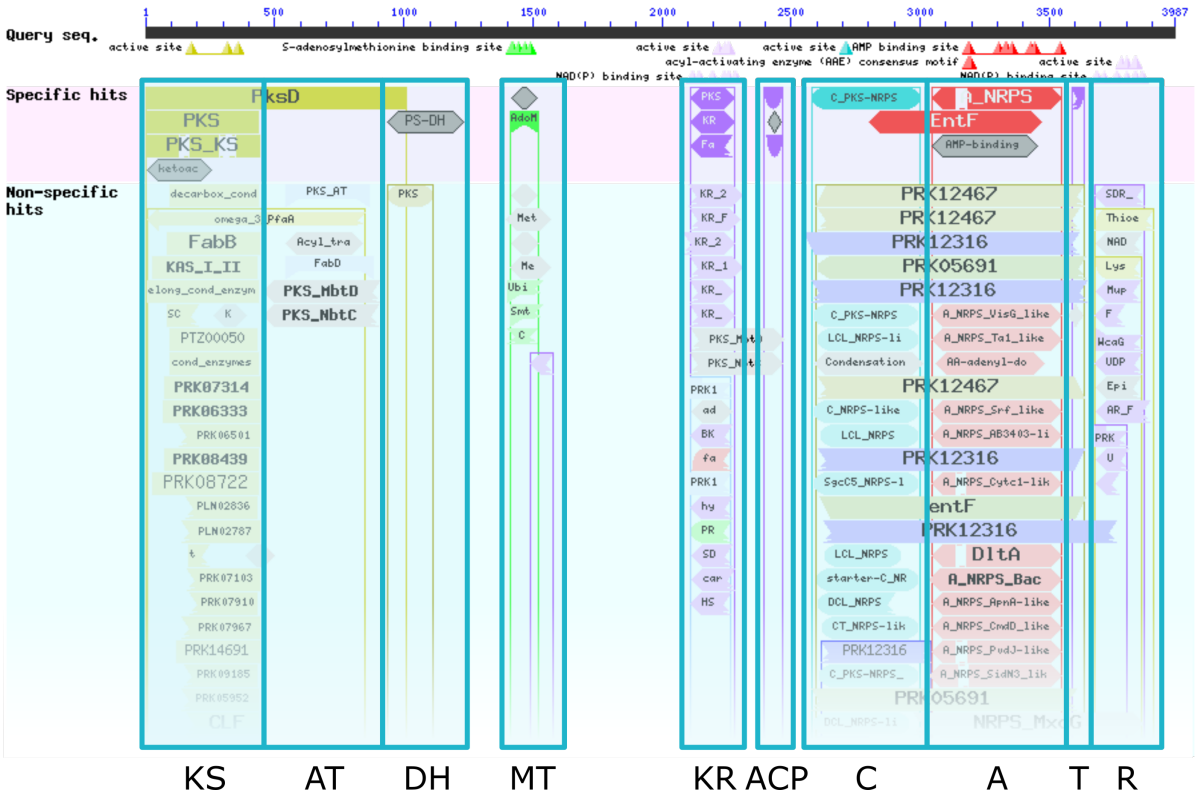
synthaser uses a central rule file which controls which domains it will save in domain architecture predictions, as well as how sequences get classified based on those domain architectures.

The default rule file distributed in the package is meant for the analysis of fungal megasynthases (polyketide synthases, fatty acid synthases and nonribosomal peptide synthetases). However, this can be repurposed for the analysis of any other multidomain protein very easily using our web application.

What follows is a brief description of how synthaser uses the rule file at the various stages of its workflow.

Identifying domain 'islands'

The problem synthaser aims to solve is best demonstrated in the following image:



This is the result of a CD-Search run using a hybrid polyketide synthase-nonribosomal peptide synthetase sequence as a query. Within this visualisation, you can see all of the domain hits found in the sequence during the search, as well as their positions within the sequence.

These domain hits very clearly fall into distinct 'islands' (indicated by blue boxes) with other, related domain hits. In this image, the domain islands correspond to the domain architecture of the synthase, `KS-AT-DH-MT-KR-ACP-C-A-T-R`.

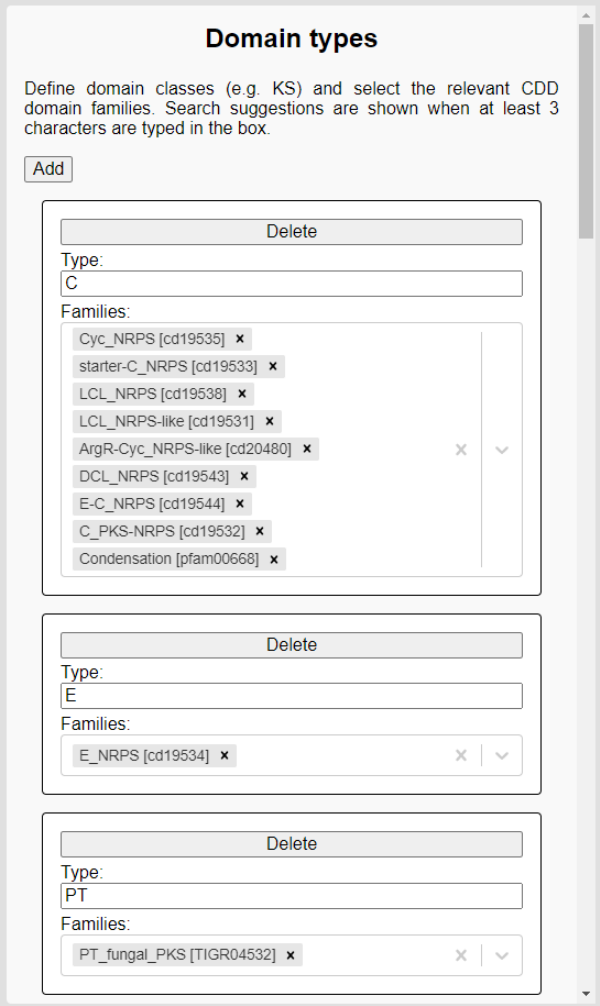
While this is easy to figure out manually by looking at the visualisation, we often want to analyse larger numbers of these sequences (e.g. when looking at full fungal genomes), which quickly makes this manual approach prohibitive. This is where synthaser comes in.

Creating domain types

The first element of the synthaser rule file is the list of domain 'types' that we want synthaser to find in the CD-Search results. Each domain type is given a name as well as a list of specific domain families which correspond to the type.

Thinking back to the domain islands example above, we may wish to create a domain type called *KS*, which consists of the domain families *PKS* and *PKS_KS* (the top-scoring domain families in the search). This would indicate to the synthaser that whenever it finds hits for either *PKS* or *PKS_KS* domains, they should be saved and categorised as *KS* domains.

These can be edited in the leftmost pane in the web application:

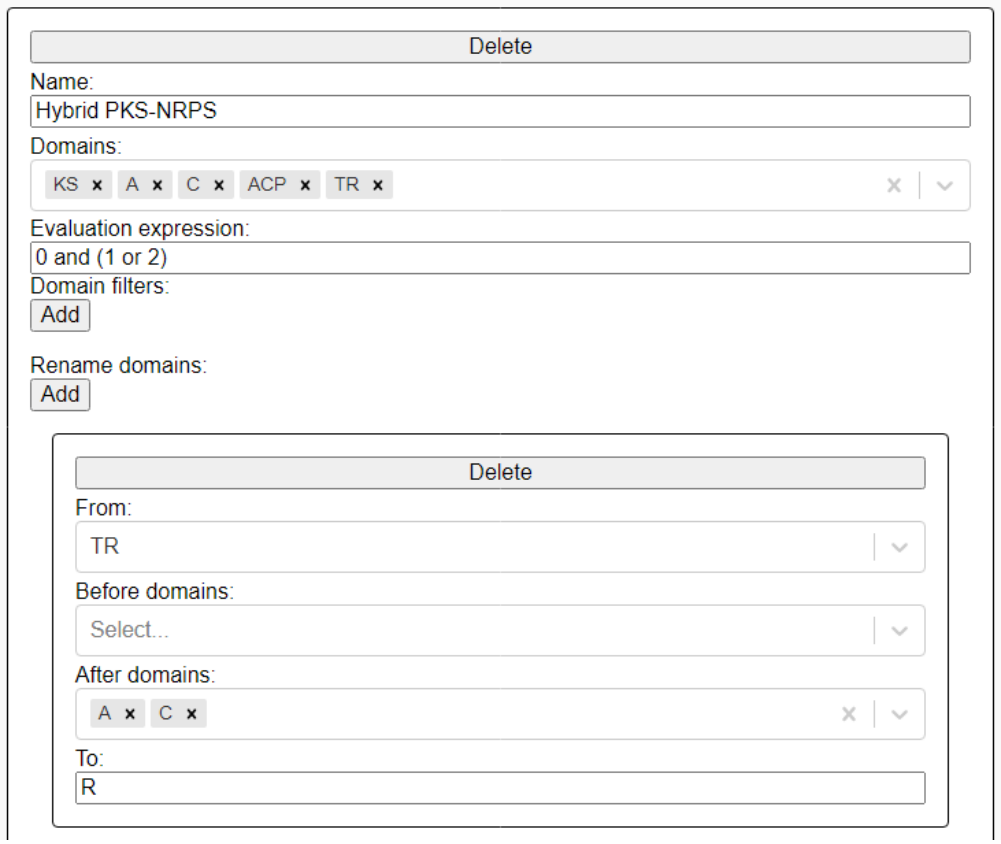


As mentioned, we first give the domain type a name (e.g. *C*), then we choose a list of specific domain families (e.g. *C_PKS-NRPS*, *Condensation*). You can directly search for families in this box by accession or name.

Creating classification rules

Once synthaser has identified the domains in a query sequence and predicted its domain architecture, the next task is classification. As with the domain identification step, while we can easily determine domain architecture directly from the visual output and tell what type of synthase we have, the difficulty comes when we want to analyse more than a single sequence at a time.

These can be edited in the *Classification rules* pane of the web application:



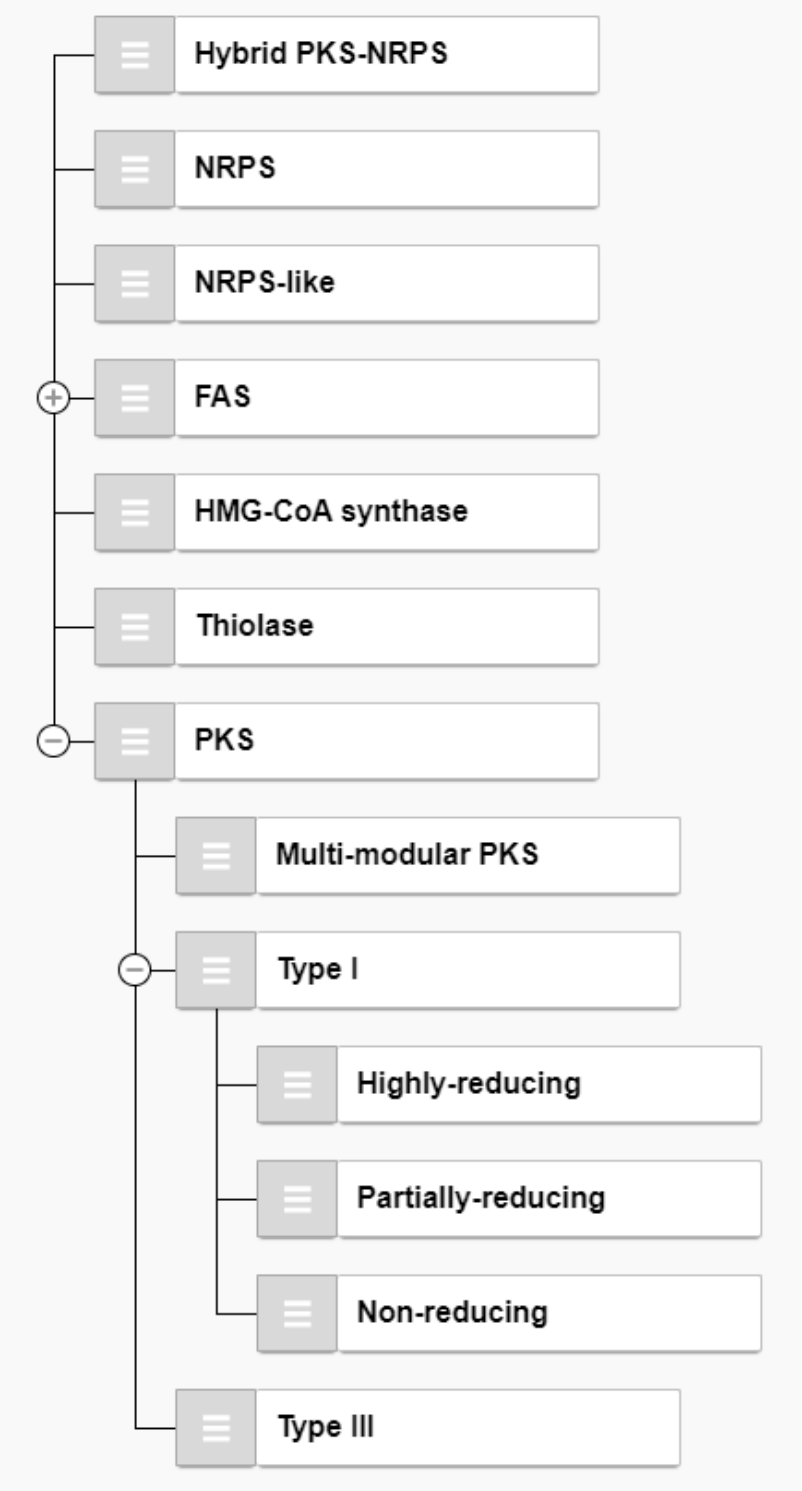
This picture shows the rule for classifying a PKS-NRPS sequence. It consists of:

1. A name, given to a sequence when the rule is evaluated succesfully (*Hyrid PKS-NRPS*).
2. A list of domain types chosen from a multi-selection box populated by domain types created in the *Domain types* pane (*KS*, *A*, *C*, *ACP*, *TR*).
3. An evaluation expression which checks for presence of key domains for this rule (0 for *KS*, 1 for *A* and 2 for *C*). This rule will be satisfied if a sequence has a *KS* domain **AND** either an *A* **OR** *C* domain.
4. A renaming rule which specifies that a domain type should be renamed in a specific circumstance due to convention (a *TR* domain, if found after a *C* or *A* domain, is renamed to *R*).

The only thing this rule does not have is domain filters, which would control the specific domain families that should be saved for a domain type. In the fungal ruleset, this is key to being able to distinguish between *KS* domains from fatty acid synthases and polyketide synthases, for example.

Classification rule evaluation hierarchy

Once you have established all of your classification rules, you need to define the order in which they should be evaluated. This can be done in the *Rule hierarchy* pane:



This list is automatically populated by the rules that you created in the *Classification rules* pane; by default, they will all be on the same evaluation level.

Rules are evaluated in top-to-bottom order according to this hierarchy. They can be rearranged by simply clicking and dragging the grey handle to the left of the rule name and dropping them where you want them. This could simply mean reordering rules on the same level (e.g. moving *NRPS* before *Hybrid PKS-NRPS*), or making certain rules children of others (e.g. if the *PKS* rule is successfully evaluated, only then does synthaser try to evaluate the *Type I* rule).

synthaser supports arbitrary levels of nesting.

Using the rule file

Once you're happy with your rule set, click the *Save rules* button in the top-right of the web application. This wll download a JSON format file containing the domain types, classification rules and hierarchy, that you can pass directly to synthaser for it to use instead of the fungal ruleset.

For example, given a custom rule file, `my_ruleset.json`:

```
synthaser search -qf sequences.fa --rule_file my_ruleset.json
```

If you ever want to modify your rule set, this can also be done in the web application by loading your rule file with the `Load rules` button, making your tweaks, then clicking `Save rules` to generate a new file.

2.1.4 Miscellaneous modules

synthaser also provides a few other modules to help you generate certain files.

getdb

The `getdb` module can be used to download pre-formatted RPS-BLAST databases for local searches. This module will connect to the NCBI FTP and download whichever database/s you specify. For example, to download the CDD to some folder `databases`:

```
synthaser getdb Cdd databases/
```

```
usage: synthaser getdb [-h] {cdd_families,Cdd,Cdd_NCBI,Cog,Kog,Pfam,Prk,Smart,Tigr}␣
↪folder

Download a pre-formatted rpsblast database.

For full description of the available databases, see:
 https://www.ncbi.nlm.nih.gov/Structure/cdd/cdd_help.shtml#CDSource

Note that 'cdd_families' will download a file containing a summary of
all families in the CDD for rule building - not a searchable database.

positional arguments:
  {cdd_families,Cdd,Cdd_NCBI,Cog,Kog,Pfam,Prk,Smart,Tigr}
                        Database to be downloaded
  folder                Folder where database is to be saved. Will save a .tar.gz␣
↪file, and extract its
                        contents to a folder of the same name.

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

getseq

The `getseq` module can be used to download sequences, in FASTA format, from the NCBI. You can provide either a text file containing newline separated NCBI identifiers, or directly in the command line separated by spaces. For example:

```
synthaser getseq KAF4294870.1 KAF4294328.1 KAF4293514.1 -o sequences.fasta
```

```
usage: synthaser getseq [-h] [-o [OUTPUT]] sequence_ids [sequence_ids ...]

Download sequences from NCBI in FASTA format. This utility will accept either a file␣
↪containing newline
separated sequence identifiers, or directly on the command line separated by spaces.

positional arguments:
  sequence_ids          Collection of NCBI sequence identifiers to retrieve

optional arguments:
  -h, --help            show this help message and exit
  -o [OUTPUT], --output [OUTPUT]
                        Where to print output (def. stdout)
```

extract

The `extract` module can be used to extract domain/query sequences from synthaser search results. It takes a JSON file (generated by `-json/--json_file`) and a prefix string which is used for the generated output files, as well as several filters.

For example, to extract KS, A and TE domain sequences:

```
$ synthaser extract session.json out_ --types KS A TE -m domain
Output: out_KS.faa out_A.faa out_TE.faa
```

To extract full NRPS and non-reducing PKS sequences:

```
$ synthaser extract session.json out_ \
    --mode synthase \
    --classes Non-reducing NRPS
Output: out_Non-reducing.faa out_NRPS.faa
```

Or to extract PKS_KS domains (CDD) only from highly-reducing PKSs:

```
$ synthaser extract session.json out_ \
    --families PKS_KS \
    --classes Highly-reducing
Output: out_PKS_KS.faa
```

```
usage: synthaser extract [-h] [-m {domain,synthase}] [--types TYPES [TYPES ...]]
                         [--classes CLASSES [CLASSES ...]] [--families FAMILIES␣
↪[FAMILIES ...]]
                         session prefix

Extract domain/synthase sequences from search results.

positional arguments:
  session               Synthaser session file
  prefix                Output file prefix

optional arguments:
  -h, --help            show this help message and exit
  -m {domain,synthase}, --mode {domain,synthase}
                        Extract domain sequences or whole synthases from a session␣
↪file
  --types TYPES [TYPES ...]
```

(continues on next page)

(continued from previous page)

```
                        Domain types
  --classes CLASSES [CLASSES ...]
                        Sequence classifications
  --families FAMILIES [FAMILIES ...]
                        CDD families
```

genbank

The `genbank` module allows you extract protein sequences from a given GenBank format file. For example:

```
synthaser genbank myfile.gbk
```

will extract all identified protein sequences and print them to the terminal.

As a convenience for fungal megasynthase analysis, we provide the `--antismash` flag, which allows you to extract PKS/NRPS sequences directly from a GenBank file generated by antiSMASH.

```
usage: synthaser genbank [-h] [--antismash] genbank

Extract protein sequences from GenBank files. To extract PKS or NRPS sequences from␣
↪antiSMASH GenBank
files, use the --antismash option.

positional arguments:
  genbank      GenBank file

optional arguments:
  -h, --help   show this help message and exit
  --antismash  Extract PKS/NRPS sequences from an antiSMASH file
```

CHAPTER 3

API Documentation

The following pages detail all synthaser modules.

3.1 API Documentation

Reference for every public API exposed by *synthaser*:

3.1.1 `synthaser.classify`

This module contains the logic for classifying synthase objects based on user-defined rules.

To classify a collection of sequences, use the classify function:

```
>>> from synthaser.classify import classify
>>> classify(my_sequences)
```

A custom classification rule file can be provided to this function like so:

```
>>> classify(my_sequences, rule_file="my_rules.json")
```

Briefly, rule files should contain:

1) Rule entries, specifying the domain combinations required to satisfy them
2) Rule graph, encoding the hierarchy and order in which rules are evaluated

Alternatively, you could build a *RuleGraph* object in Python, e.g.:

```
>>> from synthaser.classify import Rule, RuleGraph
>>> one = Rule(name="Rule 1", domains=["D1", "D2"], evaluator="0 and 1")
>>> two = Rule(name="Rule 2", domains=["D3", "D4", "D5"], evaluator="(0 and 1) or 2")
>>> three = Rule(name="Rule 3", domains=["D6", "D7"], evaluator="0 or 1")
>>> graph = [
```

(continues on next page)

(continued from previous page)

```
...     "Rule 1",
...     {
...         "Rule 2": ["Rule 3"]
...     }
... ]
>>> rg = RuleGraph(rules=[one, two, three], graph=graph)
```

And then save it to a file:

```
>>> with open("my_rules.json", "w") as fp:
...     rg.to_json(fp)
```

This *RuleGraph* object can directly classify *Synthase* objects:

```
>>> rg.classify(my_sequences)
```

For further explanation of rule files, refer to the documentation.

class synthaser.classify.**Rule**(*name=None*, *order=None*, *renames=None*, *domains=None*, *filters=None*, *evaluator=None*, ***kwargs*)

A classification rule.

name
Name given to proteins satisfying this rule.

Type str

domains
Domain types required to satisfy rule.

Type list

filters
Specific CDD families for each domain type.

Type dict

evaluator
Evaluatable rule satisfaction statement.

Type str

evaluate(*conditions*)
Evaluates the rules evaluator string given evaluated conditions.

Iterates backwards to avoid bad substitutions in larger (>=10) indices. e.g. "0 and 1 and ... and 13" -> "False and True and ... and True3"

Parameters **conditions** (*list*) – Boolean values corresponding to domains in this rule.

Returns True if rule is satisfied, otherwise False.

rename_domains(*domains*)
Renames domain types if substitutions are specified in the rule.

The rename dictionary maps domain types to other domain types. For example, an ACP domain in a PKS matches the same PP-binding domain as a T domain in an NRPS, so to follow the naming convention the NRPS rule renames ACPs to Ts.

Additionally, rename rules can be nested dicts to allow extra rules. For example, in a PKS-NRPS, the PP-binding domain in the NRPS module should be named T, not ACP. So, its rule is {'after': ['A', 'C'], 'to': 'T'}; any ACP domains after the first A or C will be renamed T.

satisfied_by(*domains*)
Evaluates this rule against a collection of domains.

Checks that: 1) required domain types are represented in the supplied domains, and 2) domains are of the desired CDD families, if any are specified.

Placeholders in the evaluator string are then replaced by their respective booleans, and evaluated.

Once a domain in the supplied domains has matched one in the rule, it cannot be matched to another in the rule. This enables rules based on counts of domains (e.g. multi-modular PKS w/ 2 KS domains).

valid_family(*domain*)
Checks a given domain matches a specified CDD family in the rule.

If no families have been specified for the given domain type, this function will return True (i.e. any family of the type is accepted).

This behaviour is controlled by the filters property of a synthaser rule. For example, to restrict a KS domain to certain CDD families:

```
"filters": [
    "type": "KS",
    "domains": ["one", "two"]
]
```

valid_order(*domains: List[synthaser.models.Domain]*) → bool
Checks given domains match specified order, if any.

Used for rules where domain order matters, e.g. a hybrid NRPS-PKS vs PKS-NRPS, where NRPS module comes before PKS and vice versa.

Iterates domain order list, finding earliest matching index in domain list. If the domain is not found, or the current index is lower than the previous (current domain occurs earlier than previous domain), order is invalid and False is returned.

class synthaser.classify.**RuleGraph**(*rules=None*, *graph=None*)
A hierarchy of classification rules.

The RuleGraph is used to classify synthases based on their domains. It stores Rule objects, as well as a directed graph controlling the order and hierarchy of classification.

An example synthaser rule graph looks like this:

```
[
    "Hybrid",
    {"PKS": ["HR-PKS", "PR-PKS", "NR-PKS"]},
    "NRPS"
]
```

In this example, the "Hybrid" rule is evaluated first. If unsuccessful, the "PKS" rule is evaluated. If this is successful, synthaser recurses into child rules, in which case the "HR-PKS", "PR-PKS" and "NR-PKS" rules can be evaluated, and so on. Each rule name must have a corresponding entry in the rules attribute.

Note that terminal leaves in the graph are placed in lists, whereas hierarchies are written as dictionaries of lists. This preserves rule order in Python, as well as preventing empty, unnecessary dictionaries at every level.

rules
Collection of synthaser rules.

Type dict

graph
Hierarchy of synthaser rules for classification.

Type dict

synthaser.classify.**classify**(*synthases*, *rule_file=None*)
Classifies synthases based on defined rules.

If no rule_file is provided, the packaged rules.json will be loaded by default.

Parameters

- **synthases** (*list*) – Synthase objects to classify.
- **rule_file** (*str*) – Path to custom classification rule file.

synthaser.classify.**get_domain_index**(*query: str, domains: List[synthaser.models.Domain]*) → Optional[int]
Finds the earliest index of a domain in a list of domains, if present.

synthaser.classify.**traverse_graph**(*graph*, *rules*, *domains*, *classifiers=None*)
Traverses a rule graph and classifies a collection of domains.

Each node is a dictionary with the schema:

{ "title": "Rule name", "children": [

{ "title": Rule name", "children": [...],

]

}

Rules are evaluated in order. If a rule is successfully evaluated, this function will recurse into any child rules, if any exist.

Finally a classification list, containing the path of rules satisfied by the given domains, is returned.

Parameters

- **graph** (*list, dict*) – Rule graph to traverse.
- **rules** (*dict*) – Rule objects to evaluate on domains.
- **domains** (*list*) – Domain objects to classify.
- **classifiers** (*list*) – Current classifiers for a Domain collection.

Returns classifiers

3.1.2 synthaser.fasta

This module contains some helper functions for handling FASTA format files/strings.

synthaser.fasta.**count**(*fasta*)
Counts sequences in an open FASTA file handle.

Iterates file and counts header lines. Then, seeks to start of the file and returns the count.

Parameters **fasta** (*file pointer*) – An open file handle corresponding to a FASTA file.

Returns Total number of sequences in the file.

Return type count (int)

synthaser.fasta.**create**(*header*, *sequence*, *limit=80*)
Creates a FASTA format string from a header and sequence.

For example:

```
>>> fasta = create_fasta('header', 'AAAAABBBBBCCCCC', wrap=5)
>>> print(fasta)
>header
AAAAA
BBBBB
CCCCC
```

Parameters

- **header** (*str*) – Name to use in FASTA definition line (i.e. >header).
- **sequence** (*str*) – The sequence corresponding to the *header*.
- **limit** (*int*) – Total characters per line before sequence is wrapped.

Returns FASTA format string.

Return type (str)

synthaser.fasta.**wrap**(*sequence*, *limit=80*)

Wraps sequences to *limit* characters per line.

Parameters

- **sequence** (*str*) – Sequence to be wrapped.
- **limit** (*int*) – Total characters per line.

Returns Sequence wrapped to maximum *limit* characters per line.

Return type (str)

3.1.3 synthaser.grouping

This module contains some functions used for grouping *Synthase* objects by their classifications.

This is used primarily when grouping sequences for the purpose of annotation in the plot (i.e. grouping Synthases of like classification, at each level in the classification hierarchy). Since annotations need to be drawn from more specific to less specific, this module generates groups in reverse.

Given a collection of classified *Synthase* objects, a basic workflow using this module might be:

1. Build a dictionary of synthase headers grouped by classification:

```
>>> levels = group_synthases(synthases)
>>> levels
defaultdict(<class 'list'>, {'PKS': ['seq1', 'seq2', ...], 'HR-PKS': ['seq1', ...]})
```

2. Determine the hierarchy of synthase classifications in your synthases.

```
>>> hierarchy = get_classification_paths(synthases)
>>> hierarchy
{'PKS': {'Type I': {'Non-reducing': {}, 'Highly-reducing': {}, 'Partially-reducing':
↪{}}}, 'Hybrid': {}}
```

Note, this is agnostic to our rule files - the rule hierarchy here is built solely from what is stored in each *Synthase* object. This also means there should be no redundant classifications.

3. Build an array of annotation groups, each in drawing (i.e. reverse) order.

```
>>> groups = get_annotation_groups(hierarchy)
>>> groups
[
  [
    {'classification': 'Partially-reducing', 'depth': 2},
    {'classification': 'Highly-reducing', 'depth': 2},
    {'classification': 'Non-reducing', 'depth': 2},
    {'classification': 'Type I', 'depth': 1},
    {'classification': 'PKS', 'depth': 0}
  ],
  [{'classification': 'Hybrid', 'depth': 0}],
]
```

Since annotations are drawn from more to less specific, and each classification is drawn at some offset to the previous one, we need some way of differentiating their level - hence the *depth* property.

synthaser.grouping.**build_dict**(*path*, *d=None*)
Recursively generates a dictionary of dictionaries from a list.

synthaser.grouping.**get_classification_paths**(*synthases*)
Determines the hierarchy of synthase classifications.

This hierarchy is used when annotating the plot with classification bars. It should be used in conjunction with the per-classification synthase dictionary generated using group_synthases().

synthaser.grouping.**group_synthases**(*synthases*)
Group synthases by their classifications.

synthaser.grouping.**iter_annotation_groups**(*hierarchy*)
Traverses hierarchy and iterates classification groups.

Groups are reverse sorted by depth, such that annotations are drawn from more specific to less specific.

synthaser.grouping.**iter_nested_keys**(*d*, *depth=0*)
Iterates over all keys in a nested dictionary, reporting their depth.

The depth indicates how deeply nested the yielded key is in the dictionary. It is used when annotating the plot to determine the position of the classification bars.

synthaser.grouping.**merge_dicts**(*a*, *b*)
Recursively merges two dictionaries, allowing overlapping keys.

3.1.4 synthaser.models

This module stores the classes used throughout synthaser.

The Domain class represents a conserved domain hit. It stores the broader domain type, the specific conserved domain profile name (from CDD), as well as its position in its parent synthase sequence and score from the search. It also provides methods for slicing the corresponding sequence and serialisation. We can instantiate a *Domain* object like so:

```
>>> from synthaser.models import Domain
>>> domain = Domain(
...     type='KS',
...     domain='PKS_KS',
...     start=756,
...     end=1178,
...     evalue=0.0,
```

(continues on next page)

(continued from previous page)

```
...     bitscore=300
... )
```

and get its sequence given the parent *Synthase* object sequence:

```
>>> domain.slice(synthase.sequence)
'MPIAVGM..'
```

Likewise, the *Synthase* class stores information about a synthase, including its name, amino acid sequence, *Domain* instances and its classification. It also contains methods for generating the domain architecture, extraction of domain sequences and more. For example, we can instantiate a new *Synthase* object like so:

```
>>> from synthaser.models import Synthase
>>> synthase = Synthase(
...     header='SEQ001.1',
...     sequence='MASGTC...',
...     domains=[
...         Domain(type='KS'),
...         Domain(type='AT'),
...         Domain(type='DH'),
...         Domain(type='ER'),
...         Domain(type='KR'),
...         Domain(type='ACP'),
...     ],
... )
```

Then, we can generate the domain architecture:

```
>>> synthase.architecture
'KS-AT-DH-ER-KR-ACP'
```

Or extract all of the domain sequences:

```
>>> synthase.extract_domains()
{
    "KS_0": "MPIAVGM...",
    "AT_0": "VFTGQGA...",
    "DH_0": "DLLGVPV...",
    "ER_0": "DVEIQVS...",
    "KR_0": "IAENMCS...",
    "ACP_0": "ASTTVAQ..."
}
```

The object can also be serialised to JSON (note the Domain object works the same way):

```
>>> js = synthase.to_json()
>>> with open('synthase.json', 'w') as handle:
...     handle.write(js)
```

and subsequently loaded from JSON:

```
>>> with open('synthase.json') as handle:
...     synthase = Synthase.from_json(handle)
```

This will internally convert the Synthase object, as well as any Domain objects it contains, to dictionaries, before converting to JSON using the builtin json library and writing to file. When loading up from JSON, this process is reversed, and the entries in the file are converted back to Python objects.

class synthaser.models.**Domain**(*pssm=None*, *type=None*, *domain=None*, *start=None*, *end=None*, *evalue=None*, *bitscore=None*, *accession=None*, *superfamily=None*)

A conserved domain hit.

type
Broader domain type (e.g. KS)

Type str

domain
Specific CDD family (e.g. PKS_KS)

Type str

start
Start of domain hit in parent sequence

Type int

end
End of domain hit in parent sequence

Type int

evalue
Domain hit E-value

Type float

bitscore
Domain hit bitscore

Type float

accession
CDD accession of domain family

Type str

superfamily
CDD accession of domain superfamily

Type str

slice(*sequence*)
Slices segment of sequence using the position of this Domain.

Given a Domain:

```
>>> domain = Domain(type='KS', subtype='PKS_KS', start=10, end=20)
```

And its corresponding Synthase sequence:

```
>>> synthase.sequence
'ACGTACGTACACGTACGTACACGTACGTAC'
```

We can extract the Domain:

```
>>> domain.slice(synthase.sequence)
'CGTACGTACA'
```

class synthaser.models.**Synthase**(*header=None*, *sequence=None*, *domains=None*, *classification=None*)

The Synthase class stores a query protein sequence, its hit domains, and the methods for filtering and classifying.

header

Synthase name.

Type str

sequence

Amino acid sequence of this Synthase.

Type str

domains

Conserved domain hits in this Synthase.

Type list

classification

All classification rules satisfied.

Type list

contains(*classes=None*, *types=None*, *families=None*)

Checks if Synthase contains given classifications, domain families or types.

extract_all_domains()

Extracts all domain sequences from this synthase.

For example, given a Synthase:

```
>>> synthase = Synthase(
...     header='synthase',
...     sequence='ACGT...',  # length 100
...     domains=[
...         Domain(type='KS', domain='PKS_KS', start=1, end=20),
...         Domain(type='AT', domain='PKS_AT', start=50, end=70)
...     ]
... )
```

Then, we can call this function to extract the domain sequences:

```
>>> synthase.extract_all_domains()
{'KS':['ACGT...'], 'AT':['ACGT...']}
```

Returns Sequences for each domain in this synthase keyed on domain type.

Return type dict

Raises

- `ValueError` – If the Synthase has no Domain objects.
- `ValueError` – If the sequence attribute is empty.

extract_domains(*types=None*, *families=None*)

Extract specific domain type/family sequences from this Synthase.

class synthaser.models.**SynthaseContainer**(*synthases*)

Simple container class for Synthase objects.

The purpose of this class is to facilitate batch actions on Synthase objects, i.e. serialisation, extraction of domain sequences, iteration over type/subtype, and printing summaries.

add_sequences(*sequences*)
Add amino acid sequence to Synthase objects in this container.

append(*synthase*)
S.append(value) – append value to the end of the sequence

extend(*synthases*)
S.extend(iterable) – extend sequence by appending elements from the iterable

extract_domains(*classes=None*, *types=None*, *families=None*, *by='sequence'*)
Extract domain sequences from Synthase objects in this container.

For example, given a *SynthaseContainer* containing *Synthase* objects:

```
>>> synthases = [Synthase(header='one', ...), Synthase(header='two', ...)]
>>> container = SynthaseContainer(synthases)
```

Then, the output of this function may resemble:

```
>>> container.extract_domains()
{'KS': [('one_KS_1', 'IAIA...'), ('two_KS_1', 'IAIE...')], 'AT': [...]}
```

extract_synthases(*classes=None*, *types=None*, *families=None*)
Bin entire synthase sequences.

classmethod from_sequences(*sequences*)
Build a SynthaseContainer from a dictionary of query sequences.

to_long(*delimiter=','*, *headers=True*)
Generate summary of the container in long data format.

For example:

```
Synthase  Length (aa)  Architecture        Classification
SEQ001.1  1000         KS-AT-DH-ER-KR-ACP  PKS, Type I, Highly-reducing
```

NOTE: actual output is character delimited, not human readable.

3.1.5 synthaser.ncbi

This module handles all interaction with NCBI.

Given a collection of *Synthase* objects, a workflow might look like:

1. Launch new CD-Search run

```
>>> cdsid = ncbi.launch(synthases)
>>> cdsid
QM3-qcdsearch-B4BAD4B59BC5B80-3E7CFCD3F93E21D0
```

The query sequences are sent to the batch CD-Search API, where a new run is started and assigned a unique CD-Search identifier (CDSID) which can be used to check on search progress.

Note that search parameters for this search are specified by the values in *SEARCH_PARAMS*:

```
>>> ncbi.SEARCH_PARAMS
{
  'db': 'cdd',
  'smode': 'auto',
  'useid1': 'true',
  'compbasedadj': '1',
  'filter': 'true',
  'evalue': '3.0',
  'maxhit': '500',
  'dmode': 'full',
  'tdata': 'hits'
}
```

which can be freely edited, either directly or by using the *set_search_params* function.

2. Poll CD-Search API for results using the CDSID

```
>>> response = ncbi.retrieve(cdsid)
```

This function repeatedly polls the API at regular intervals until either results or an error has occurred. Internally, this function calls *check()*, which takes a CDSID and sends a single request to the API. It returns a *Response* object (from the *requests* library), which will have any search content saved in its *text* or *content* properties.

```
>>> print(response.text)
#Batch CD-search tool   NIH/NLM/NCBI
#cdsid  QM3-qcdsearch-B4BAD4B59BC5B80-3E7CFCD3F93E21D0
#datatype       hitsFull Results
#status 0
#Start time     2019-09-03T04:21:23     Run time        0:00:04:23
#status success
```

3. Parse results and create *Synthase* objects

```
>>> from synthaser import results
>>> handle = results.text.split("\n")
>>> synthases = results.parse(handle)
```

Additionally, this module provides *efetch_sequences*, a function for fetching sequences from NCBI from a collection of accessions. For example:

```
>>> ncbi.efetch_sequences(['CBF71467.1', 'XP_681681.1'])
{'CBF71467.1': 'MQSAGMHRATA...', 'XP_681681.1': 'MQDLIAIVGSA...'}
```

The accessions are sent to the NCBI's Entrez API, which returns the sequences in FASTA format. They are parsed using *fasta.parse*, and the resulting dictionary is returned.

`synthaser.ncbi.`**`check`**(*cdsid*)

Checks the status of a running CD-search job.

CD-Search runs are assigned a unique search ID, which typically take the form:

```
QM3-qcdsearch-xxxxxxxxxxx-yyyyyyyyyyy
```

This function queries NCBI for the status of a running CD-Search job corresponding to the search ID specified by `cdsid`.

```
>>> response = check('QM3-qcdsearch-B4BAD4B59BC5B80-3E7CFCD3F93E21D0')
```

If the job has finished, this function will return the `requests.Response` object which contains the run results. If the job is still running, this function will return None. If an error is encountered, a ValueError will be thrown with the corresponding error code and message.

Parameters **cdsid** (*str*) – CD-search identifier (CDSID).

Returns If the job has completed and is ready for download False: If the job is still running

Return type True

Raises

- `ValueError` – If the returned results file has a successful status code but is actually empty (i.e. contains no results), perhaps due to an invalid query.
- `ValueError` – When a status code of 1, 2, 4 or 5 is returned from the request.

`synthaser.ncbi.`**efetch_sequences**(*headers*)

Retrieve protein sequences from NCBI for supplied accessions.

This function uses EFetch from the NCBI E-utilities to retrieve the sequences for all synthases specified in headers. It then calls fasta.parse to parse the returned response; note that extra processing has to occur because the returned FASTA will contain a full sequence description in the header line after the accession.

Parameters **headers** (*list*) – A collection of NCBI sequence identifiers (accession, GI, etc)

Returns Sequences downloaded from NCBI

Return type sequences (dict)

`synthaser.ncbi.`**get_results**(*cdsid*)

Downloads results corresponding to a CDSID.

Parameters **cdsid** (*str*) – CD-Search identifier

Returns Response object containing search results

Return type requests.Response

Raises `ValueError` – If response has bad status code

`synthaser.ncbi.`**launch**(*query*)

Launches a new CDSearch run.

Parameters **query** (`Synthase, SynthaseContainer`) – Synthase objects to be searched. This could either be a single Synthase object or a SynthaseContainer; other objects could be used as long as they implement a to_fasta method.

Returns CDSearch ID (CDSID) corresponding to the new run. This takes the form: QM3-qcdsearch-XXXXXXXXXXXXXXX-YYYYYYYYYYYYYYY.

Return type cdsid (str)

Raises

- `AttributeError` – query has no to_fasta method
- `AttributeError` – No CDSID was returned from NCBI

`synthaser.ncbi.`**retrieve**(*cdsid*, *max_retries=-1*, *delay=20*)

Poll CDSearch for results.

This method queries the NCBI for results from a CDSearch job corresponding to the supplied cdsid. If max_retries is -1, this function will check for results every delay interval until something is returned.

If you wish to save the results of a CD-Search run to file, you can supply an open file handle via the output parameter:

```
>>> with open('results.tsv', 'w') as results:
...     retrieve(
...         'QM3-qcdsearch-B4BAD4B59BC5B80-3E7CFCD3F93E21D0',
...         output=results
...     )
```

This function returns the Response object returned by check():

```
>>> response = retrieve('QM3-qcdsearch-B4BAD4B59BC5B80-3E7CFCD3F93E21D0')
>>> print(response.text)
#Batch CD-search tool       NIH/NLM/NCBI
#cdsid      QM3-qcdsearch-B4BAD4B59BC5B80-3E7CFCD3F93E21D0
#datatype   hitsFull Results
#status     0
...
```

Parameters

- **cdsid** (*str*) – CD-search job ID. Looks like `QM3-qcdsearch-xxxxxxxxxxx-yyyyyyyyyyy`.
- **output** (*file pointer*) – Save results to a given open file handle instead of a local file. This facilitates usage of e.g. tempfile objects.
- **max_retries** (*int*) – Maximum number of retries for checking job completion. If -1 is given, this function will keep paging for results until something is returned.
- **delay** (*int*) – Number of seconds to wait between each request to the NCBI. The wait time is re-calculated to this value each time, based on the time taken by the previous request. By default, this is set to 20; giving a value less than 10 will result in a ValueError being thrown.

Returns Response returned by the check()

Return type (requests.models.Response)

Raises

- `ValueError` – If delay is less than 10.
- `ValueError` – If no Response is returned by check()

`synthaser.ncbi.`**set_search_params**(*database=None*, *smode=None*, *useid1=None*, *compbasedadj=None*, *filter=None*, *evalue=None*, *maxhit=None*, *dmode=None*)

Set CD-Search search parameters.

All search parameters are stored in SEARCH_PARAMS; this can either be edited directly, or through this function, prior to a search.

Parameters

- **database** (*str*) – Name of search database. Available options are 'cdd' (default), 'pfam', 'smart', 'tigrfam', 'cog' and 'kog'. Only applies when smode is live.
- **smode** (*str*) – Search mode; 'auto' (automatic), 'prec' (precalculated only) or 'live' (live searches).
- **useid1** (*str*) – Search archived sequences ('true' or 'false')
- **compbasedadj** (*str*) – Composition-corrected scoring ('0' or '1')
- **filter** (*str*) – Filter out compositionally biased regions ('true' or 'false')

- **evalue** (*float*) – E-value cutoff
- **maxhit** (*int*) – Maximum number of hits per query
- **dmode** (*str*) – Data mode of output ('rep', 'std', or 'full')

For a full description of parameters, refer to the NCBI's documentation.

3.1.6 synthaser.plot

This modules handles the construction of *synthaser* visualisations.

synthaser plots are implemented as HTML documents with embedded visualisations created using the D3 JavaScript library. Files used in this process are all stored separately inside the *plot* folder in the source code (i.e. separate CSS, JavaScript, HTML).

This module provides two ways to construct a *synthaser* plot from a collection of *Synthase* objects: hosting using Python's built in *socketserver* module, or generating a completely static HTML file containing all code elements necessary for the plot.

1. Get necessary data

```
>>> data = plot.get_data(synthases)
```

This generates a dictionary which can be converted easily to JSON and given to the JavaScript visualisation. It contains a dictionary of the *Synthase* objects, the order in which they should be drawn, a dictionary of *Synthase* objects grouped by their classifications, and the annotation group hierarchy generated using *grouping.get_annotation_groups()*.

2. a) Dynamically serve plots using *socketserver*

```
>>> plot.serve_html(data)
```

This will host the visualisation on some randomly chosen open port on localhost. It requires a keyboard interrupt to stop serving the plot.

2. b) Generate static HTML file

```
>>> plot.save_html(data, "myoutput.html")
```

This will generate a completely static HTML file containing all code elements required to render the plot (JavaScript and CSS embedded directly in the HTML file). This can then be moved/copied anywhere you would like.

This workflow is mirrored by the *plot_synthases* function:

```
>>> plot.plot_synthases(synthases, output="myoutput.html")
```

class synthaser.plot.**CustomHandler**(*data*, **args*, ***kwargs*)
Handler for serving cblaster plots.

do_GET()
Serves each component of the cblaster plot.

log_message(*format*, **args*)
Suppresses logging messages on every request.

synthaser.plot.**plot_synthases**(*container*, *output=None*)
Generates synthaser plot from a collection of Synthase objects.

synthaser.plot.**save_html**(*data*, *output*)
Generates a static HTML file with all visualisation code.

synthaser.plot.**serve_html**(*data*)
Serve a synthaser plot using the socketserver module.

3.1.7 synthaser.results

This module stores functions for parsing CD-Search output.

All functionality is provided by *parse*, which takes an open file handle corresponding to a CD-Search hit table and returns a list of fully characterized Synthase objects, i.e.:

```
>>> from synthaser import results
>>> with open('results.txt') as handle:
...     synthases = results.parse(handle)
>>> synthases
[AN6791.2 KS-AT-DH-MT-ER-KR-ACP, ... ]
```

synthaser uses the *results.DOMAINS* dictionary to control which domain families, and the quality thresholds (length, bitscore) they must meet, to save in any given search. This can be edited directly using *update_domains*, or loaded from a JSON file using *load_domain_json*. An entry in this dictionary may look like:

For further details on how to obtain these values and use a custom domain file, please refer to the user guide.

synthaser.results.**choose_representative_domain**(*group*, *by='evalue'*)
Select the best domain from a collection of overlapping domains.

This function tests rules stored in *special_rules*, which are lambdas that take two variables. It sorts the group by e-value, then tests each rule using the container (first, best scoring group) against all other Domains in the group.

If any test is True, the container type is set to the rule key and returned. Otherwise, this function will return the container Domain with no modification.

Parameters

- **group** (*list*) – Overlapping Domain objects
- **by** (*str*) – Measure to use when determining the best domain of the group. Choices: 'bitscore': return domain with highest bitscore (relative to threshold) 'evalue': return domain with lowest E-value 'length': return longest domain hit

Returns Highest scoring Domain in the group. If any special rules have been satisfied, the type of this Domain will be set to that rule (e.g. Condensation -> Epimerization).

Return type *Domain*

synthaser.results.**domain_from_row**(*row*)
Parse a domain hit from a row in a CD-search results file.

For example, a typical row might looks like:

```
>>> print(row)
Q#1 - >AN6791.2     specific        225858  9       1134    0       696.51 ␣
↪COG3321 PksD    -       cl09938
```

Using this function will generate:

```
>>> domain_from_row(row)
PksD [KS] 9-1134
```

Parameters **row** (*str*) – Tab-separated row from a CDSearch results file

Returns Instance of the Domain class containing information about this hit

Return type *Domain*

Raises `ValueError` – If the domain in this row is not in the DOMAINS dictionary.

`synthaser.results.`**`filter_domains`**(*domains*, *by='evalue'*, *coverage_pct=0.5*, *tolerance_pct=0.1*)

Filter overlapping Domain objects and test adjcency rules.

Adjacency rules are tested again here, in case they are missed within overlap groups. For example, the NRPS-para261 domain is not always entirely contained by a condensation domain, so should be caught by this pass.

Parameters

- **`domains`** (*list*) – Domain instances to be filtered
- **`by`** (*str*) – Metric used to choose representative domain hit (def. 'evalue')
- **`coverage_pct`** (*float*) – Conserved domain coverage percentage threshold
- **`tolerance_pct`** (*float*) – CD length tolerance percentage threshold

Returns Domain objects remaining after filtering

Return type list

`synthaser.results.`**`filter_results`**(*results*, ***kwargs*)

Build Synthase objects from a parsed results dictionary.

Any additional kwargs are passed to _filter_domains.

Parameters **`results`** (*dict*) – Grouped Domains; output from _parse_cdsearch_table.

Returns Synthase objects containing all Domain objects found in the CD-Search.

Return type synthases (list)

`synthaser.results.`**`group_overlapping_hits`**(*domains*)

Iterator that groups Domain objects based on overlapping locations.

Parameters **`domains`** (*list*) – Collection of Domain objects belonging to a Synthase

Yields *group (list)* – Group of overlapping Domain objects

`synthaser.results.`**`is_fragmented_domain`**(*one*, *two*, *coverage_pct=0.5*, *tolerance_pct=0.1*)

Detect if two adjacent domains are likely a single domain.

This is useful in cases where a domain is detected with multiple small hits. For example, an NRPS may have two adjacent condensation (C) domain hits that are both individually too small and low-scoring, but should likely just be merged.

If two hits are close enough together, such that the distance between the start of the first and end of the second is within some tolerance (default +-10%) of the total length of a domains PSSM, this function will return True.

Parameters

- **`one`** (`Domain`) – Domain instance
- **`two`** (`Domain`) – Domain instance
- **`coverage_pct`** (*float*) – Conserved domain hit percentage coverage threshold. A hit is considered truncated if its total length is less than coverage_pct * CD length.
- **`tolerance_pct`** (*float*) – Percentage of CD length to use when calculating acceptable lower/upper bounds for combined domains.

Returns Domain instances are likely fragmented and should be combined. False: Domain instances should be separate.

Return type True

synthaser.results.**load_domains**(*rule_file*)
Loads domains from a synthaser rule file.

Rule file domain schema: {

'name': KS, 'domains': [

{ 'accession': 'smart00825', 'name': 'PKS_KS' ...

}

This function flattens the domain type array to create a dictionary of domain families, so these can be easily looked up directly from CD-Search rows.

synthaser.results.**parse**(*handle*, *mode='remote'*, ***kwargs*)
Parse CD-Search results.

Any additional kwargs are passed to *synthases_from_results*.

Parameters

- **handle** (*file*) – An open CD-Search results file handle. If you used the website to analyse your sequences, the file you should download is Domain hits, Data mode: Full, ASN text. When using a *CDSearch* object, this format is automatically selected.
- **mode** (*str*) – Search mode ('local' or 'remote')

Returns A list of Synthase objects parsed from the results file.

Return type list

Raises ValueError – Search mode not 'local' or 'remote'

synthaser.results.**parse_cdsearch**(*handle*)
Parse a CD-Search results table and instantiate Domain objects for each hit.

Parameters **handle** (*file*) – Open file handle corresponding to a CD-Search results file.

Returns Lists of Domain objects keyed on the query they were found in.

Return type results (dict)

synthaser.results.**parse_rpsbproc**(*handle*)
Parse a results file generated by rpsblast->rpsbproc.

This function takes a handle corresponding to a rpsbproc output file. local.rpsbproc returns a subprocess.CompletedProcess object, which contains the results as byte string in it's stdout attribute.

3.1.8 synthaser.rpsblast

This module provides functionality for setting up and performing local *synthaser* searches using *RPS-BLAST* and *rpsbproc*. *RPS-BLAST* (Reverse PSI-BLAST) searches query sequences against databases of domain family profiles, and *rpsbproc* is used to post-process the raw results into something resembling results from an online CD-Search run. If *synthaser* cannot find either program on the system *$PATH*, it will raise an exception. For details on installing *RPS-BLAST* and *rpsbproc*, please refer to the user guide.

A basic search can be performed using the *search* function:

```
>>> rpsblast.search("sequences.fasta", "Cdd_LE", cpu=4)
```

This will automatically search the sequences in `sequences.fasta` against the `Cdd_LE` using *RPS-BLAST* and process the raw results using *rpsbproc*, resulting in *Response* object which be readily parsed like in a remote CD-Search.

A profile database can be downloaded using the *download_database* function, e.g.:

```
>>> path = rpsblast.download_database("my_folder", flavour="Cdd")
```

This will connect to the NCBI's FTP and download the "Cdd" database (the complete database). The downloaded file will be a .tar archive, which can be extracted using *untar*:

```
>>> untarred_path = rpsblast.untar(path)
```

Alternatively, just use *getdb* to do both steps at once:

```
>>> rpsblast.getdb("Cdd", "myfolder")
```

`synthaser.rpsblast.`**`get_program_path`**(*program*)
Get full path to a program on system PATH.

`synthaser.rpsblast.`**`rpsblast`**(*query*, *database*, *cpu=2*)
Run rpsblast on a query file against a database.

`synthaser.rpsblast.`**`rpsbproc`**(*results*)
Convert raw rpsblast results into CD-Search results using rpsbproc.

Note that since rpsbproc is reliant upon data files that generally are installed in the same directory as the executable (and synthaser makes no provisions for them being stored elsewhere), we must make sure we have the full path to the original executable. If it is called via e.g. symlink, rpsbproc will not find the data files it requires and throw an error.

The CompletedProcess returned by this function contains a standard CD-Search results file, able to be parsed directly by the results module.

`synthaser.rpsblast.`**`search`**(*query*, *database*, *cpu=2*)
Convenience function for running rpsblast and rpsbproc.

3.1.9 synthaser.search

This module serves as the starting point for *synthaser*, preparing input and dispatching it to either local or remote searches.

In any given search, input can either be a FASTA file or a collection of NCBI sequence identifiers. The *prepare_input* function is used to generate a *SynthaseContainer* object from either source which can then be used as a query. For example:

```
>>> sc1 = search.prepare_input(query_ids=["SEQ001.1", "SEQ002.1"])
>>> sc2 = search.prepare_input(query_file="my_sequences.fasta")
```

If *query_ids* are used, the sequences are first retrieved using NCBI Entrez using *ncbi.efetch_sequences()*.

A full *synthaser* search can be performed using the *search* function. This prepares the input (ids or FASTA) as above, then launches local and remote searches using the *ncbi* and *rpsblast* modules, respectively. Results are then parsed using the *results* module, and classified using the *classify* module. Lastly, the *SynthaseContainer* object which was created inside this function is returned.

```
>>> sc = search.search(query_file="my_sequences.fasta")
```

To use custom domain and classification rules, simply provide the paths to each file to the *search* function:

```
>>> sc = search.search(
...     query_file="my_sequences.fasta",
...     domain_file="my_domains.json",
...     classify_file="my_rules.json",
... )
```

Previous searches are stored in the *SEARCH_HISTORY* variable, and can be summarised using the *history* function:

```
>>> ncbi.history()
1.      Run ID: QM3-qcdsearch-B4BAD4B59BC5B80-3E7CFCD3F93E21D0
    Parameters:
                    db: cdd
                 smode: auto
                useid1: true
          compbasedadj: 1
                filter: true
                evalue: 3.0
                maxhit: 500
                 dmode: full
                 tdata: hits
```

This module contains routines for performing local/remote searches.

synthaser.search.**history**()
Print out summary of previously saved CD-Search runs. :raises: `ValueError` – If SEARCH_HISTORY is empty (i.e. no searches have been run)

synthaser.search.**prepare_input**(*query_ids=None*, *query_file=None*)
Generate a SynthaseContainer from either query IDs or a query file.

Returns Synthase objects for query sequences

Return type *SynthaseContainer*

Raises `ValueError` – Neither query_ids nor query_file provided

synthaser.search.**search**(*mode='remote'*, *query_ids=None*, *query_file=None*, *rule_file=None*, *classify_file=None*, *results_file=None*, *cdsid=None*, *delay=20*, *max_retries=-1*, *database=None*, *cpu=2*, ***kwargs*)
Run a synthaser search.

CD-Search parameters can be given as kwargs which are passed on to _remote.

Parameters

- **mode** (*str*) – synthaser search mode ('local' or 'remote')
- **query_ids** (*str, file*) – NCBI sequence identifiers to analyse
- **query_file** (*file*) – Open FASTA file handle
- **rule_file** (*file*) – Custom rule JSON file to use when parsing results
- **results_file** (*file*) – Results file from a previous CDSearch/RPSBLAST search
- **cdsid** (*str*) – CDSearch ID from a previous search
- **delay** (*int*) – Time delay (s) between polling NCBI for results (def. 20)

- **max_retries** (*int*) – Maximum number of polling attempts before exiting (def. -1)
- **database** (*str*) – rpsblast database to use in local searches
- **cpu** (*int*) – Number of threads to use in rpsblast

Returns Synthase objects representing query sequences

Return type *SynthaseContainer*

Raises `ValueError` – Too many sequences provided (NCBI limit = 4000)

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

I

L

M

N

P

R

S

T

V

W