Data Function PyLib Documentation

Release 0.1

Glysade, LLC

May 10, 2023

CONTENTS

CHAPTER

ONE

DATA FUNCTION BACKGROUND

This document contains information on building Glysade Python data functions for Spotfire.

To quickly understand how to construct a data function script that can be included in the data function definition YAML, see the examples in *Getting Started With Python Data Functions*.

A core API and utility functions for handling chemical and biological data is described in *Data Function Core API*. This core API should be sufficient to build most data functions.

The library contains a number of builtin data functions described in *Builtin Data Function Modules*. These data functions are executed by dynamically importing modules and executing a class method, rather than running a script.

The remainder of the documentation concerns the *ruse* packages. These modules are required by the builtin data functions. If possible, avoid using these modules as they may change in the future.

CHAPTER

TWO

GETTING STARTED WITH PYTHON DATA FUNCTIONS

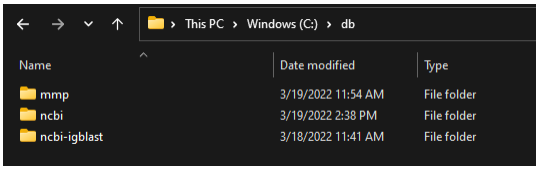
Data functions are defined in data function definition YAML files. The YAML file contains a description of the UI components and input fields to the data function.

Glysade provides a pre-built CPython instance containing RDKit and Biopython modules. Additionally, an API library is available. This library contains methods to read data function requests, construct data function responses, convert columns to and from sequences or molecule, together with modules for built in data functions.

The data function YAML file describes how Python data functions are executed. If the **executorId** is set to *Glysade.CPythonDataFxn* and the **serviceUri** is *glysade.python* then a script is executed if **serviceName** is *Script*, otherwise the **serviceName** setting is use to specify a Python module and class to run the data function. Any script to be run is specified in the **script** tag. The Python script should define a method called *execute* that takes a `DataFunctionRequest` as an argument and returns a `DataFunctionResponse` object.

2.1 Data Function Databases

Many of the data functions defined in the definition YAML require local resources such as Blast, IGBlast and MMP databases. These example databases may be downloaded as a ZIP archive from Dropbox. The ZIP archive should be expanded in the root of the C:\ drive.



2.2 Exact Mass Example

This data function returns a new column containing the exact masses of the molecules in the input column.

```
from rdkit.Chem.Descriptors import ExactMolWt
from df.chem_helper import column_to_molecules
from df.data_transfer import DataFunctionRequest, DataFunctionResponse, DataType,␣
↪ColumnData, \
    string_input_field


def execute(request: DataFunctionRequest) -> DataFunctionResponse:
    column_id = string_input_field(request, 'structureColumn')
```

(continues on next page)

(continued from previous page)

```
    input_column = request.inputColumns[column_id]
    mols = column_to_molecules(input_column)
    weights = [None if m is None else ExactMolWt(m) for m in mols]
    output_column = ColumnData(name=f'{input_column.name} Exact Mass', dataType=DataType.
↪DOUBLE, values=weights)
    response = DataFunctionResponse(outputColumns=[output_column])
    return response
```

- Lines 1-4. Required imports.
- Line 7. The data function script must implement an execute method with this signature
- Line 8. The value of the *structureColumn* input field is extracted from the request using the helper method *string_input_field()*. This is the Spotfire columnId of the structure column
- Line 9. The *ColumnData* object containing the Spotfire input columns is in the Dict *inputFields* keyed by columnId
- Line 10. Convert column values to RDKit molecules using the helper function *column_to_molecules()*
- line 11. Map the list of RDKit molecules to a list of exact masses, accounting for the fact that some molecules may be none
- line 12. Construct an output column of type *ColumnData* containing the exact masses
- lines 13-14. Construct and return the response of type *DataFunctionResponse*

2.3 Deprotect Example

RDkit contains a method to perform common deprotection reactions. This data function takes an input column of molecules and returns an output column of deprotected molecules together with a second boolean column with rows set to True when a deprotection reaction has occurred.

```
from typing import Optional

from df.chem_helper import column_to_molecules, molecules_to_column
from df.data_transfer import DataFunctionRequest, DataFunctionResponse, DataType,␣
↪ColumnData, \
    string_input_field
from rdkit import Chem
from rdkit.Chem.rdDeprotect import Deprotect
from rdkit.Chem.rdchem import Mol


def compare_molecules(mol1: Optional[Mol], mol2: Optional[Mol]) -> bool:
    if not mol1 or not mol2:
        return False
    return Chem.MolToSmiles(mol1, True) != Chem.MolToSmiles(mol2, True)


def execute(request: DataFunctionRequest) -> DataFunctionResponse:
    column_id = string_input_field(request, 'structureColumn')
    input_column = request.inputColumns[column_id]
    input_molecules = column_to_molecules(input_column)
```

(continues on next page)

(continued from previous page)

```
21      deprotected_molecules = [None if m is None else Deprotect(m) for m in input_
↪molecules]
22      changed = [compare_molecules(mol1, mol2) for mol1, mol2 in zip(input_molecules,␣
↪deprotected_molecules)]
23      output_molecules_column = molecules_to_column(deprotected_molecules, f'Deprotected
↪{input_column.name}', DataType.STRING)
24      changed_column = ColumnData(name='Changed', dataType=DataType.BOOLEAN,␣
↪values=changed)
25      response = DataFunctionResponse(outputColumns=[output_molecules_column, changed_
↪column])
26      return response
```

- Lines 11-14. This function returns True if two molecules are the same using a canonical smiles string comparison
- Line 21. Deprotects the the input molecules (accounting for null/None values)
- Line 22. Create values for a boolean column indicating if a molecule has changed (been deproctected)
- Line 23. Create an output column of structures for the deprotected molecules using the helper function *molecules_to_column()*

2.4 DNA Translation Example

In this example of a biological transform data function, DNA sequences are translated into protein sequences using Biopython.

```
1  from df.bio_helper import column_to_sequences, sequences_to_column
2  from df.data_transfer import DataFunctionRequest, DataFunctionResponse, string_input_
↪field
3
4
5  def execute(request: DataFunctionRequest) -> DataFunctionResponse:
6      column_id = string_input_field(request, 'sequenceColumn')
7      input_column = request.inputColumns[column_id]
8      input_sequences = column_to_sequences(input_column)
9      codon_table_name = string_input_field(request, 'codonTableName', 'Standard')
10     output_sequences = [None if s is None else s.translate(codon_table_name) for s in␣
↪input_sequences]
11     output_column = sequences_to_column(output_sequences, f'Translated {input_column.
↪name}', genbank_output=False)
12     response = DataFunctionResponse(outputColumns=[output_column])
13     return response
```

- Lines 6-7. Extract the sequence column from the request in the same fashion as for molecular columns
- Line 8. Convert the sequence column to a list of sequence records using the helper function *column_to_sequences()*
- Line 9. A second input field with id *codonTableName* may be used to specify an alternative codon table
- Line 10. Use Biopython to translate the input DNA or RNA sequences to protein sequences accounting for null/None values.
- Line 11. Create an output column for the protein sequences using the helper function *sequences_to_column()*

2.5 Debugging and Developing Python Data Functions

If a data function Python script has been created using the *Developer* tab of the *Charts Data Function* visual, it is relatively simple to debug and code that data function script using *PyCharm <https://www.jetbrains.com/pycharm/>*._

In the final step of the data function development wizard you are prompted to export the data function definition to a folder. After doing this, navigate to the folder and run the *pycharm.bat* file. Assuming you have installed PyCharm, and either *pycharm.cmd* or *pycharm64.exe* are on your path, PyCharm will open and create a new project in the folder.

Unfortunately, it is not possible to automate adding the Glysade CPython distribution to Pycharm. If you have already defined a python interpreter in PyCharm, the first time PyCharm is used to open a data function the correct path to the Glysade Python interpreter must be added as described here. The interpreter path is listed in the text of the *pycharm.bat* file.

The following files are available in the project:

1. *script.py*: the script block from the data function definition
2. *in.json*: example data function JSON input data
3. *main.py*: a script that reads *in.json* calls the data function in *script.py* and writes the result to *out.json*
4. *test.py*: a unit test template
5. *command.bat*: a batch file that runs *main.py* with the correct environment and Python interpreter. Once in production it is this script that runs the data function.

Once the data function is working satisfactorily, to create the required data function YAML, the contents of *script.py* can be pasted into the script dialog in the *Developer* tab of the *Charts Data Function* visual.

2.6 Fixing Broken Data Functions

When a Python data function fails to run, an error message is posted as a Spotfire notification. The error message will contain the path of the data function job directory.

The job directory contains the files listed in *Debugging and Developing Python Data Functions*. *Command.bat* may be run to reproduce the data function error and *script.py* can be edited to resolve the error. Once the script is fixed the data function definition YAML can be updated.

CHAPTER

THREE

DATA FUNCTION CORE API

3.1 data_transfer.py

Data transfer model objects and utility functions for Glysade Python data functions

We use classes derived from Pydantic Models. This allows request and response validation and serialization to and from JSON.

class df.data_transfer.**ColumnData**(***data*)

A data model class containing a Spotfire column. Used when the data function contains column inputs and should be instantiated when the data function is returning column outputs

Create a new model by parsing and validating input data from keyword arguments.

Raises ValidationError if the input data cannot be parsed to form a valid model.

contentType: Optional[str]

Optional content type for the column (e.g. chemical/s-smiles)

dataType: *DataType*

The data type stored in the column

insert_nulls(*positions=None*, *offset=0*)

Inserts nulls (or None) into the values array at the specified positions. The attribute *missing_null_positions* will be used if the position argument is not supplied.

For example, you may remove null values from an input column, create an output column then use the *missing_null_positions* attribute from the input column as an argument to correctly insert nulls into the output column.

Parameters

- **positions** (Optional[List[int]]) – optional list of null positions
- **offset** (int) – optional offset for null positions

Return type
None

missing_null_positions: List[int]

An array to track null values if they have been removed

name: str

The column name

properties: Dict[str, str]

Spotfire column properties

remove_nulls()

Remove null data values and store their position in the *missing_null_positions* attribute

Return type
None

values: List[Optional[Any]]

A list of row values for the column. The type used is based on the dataType (*DataType*) attribute or None if the cell is empty

class df.data_transfer.**DataFunction**

Base class to be used when defining builtin data function classes that are run from their module files

abstract execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.data_transfer.**DataFunctionRequest**(**data*)

Model class for data function requests

Create a new model by parsing and validating input data from keyword arguments.

Raises ValidationError if the input data cannot be parsed to form a valid model.

id: str

Unique UUID4 identifier for the request

inputColumns: Dict[str, *ColumnData*]

Input columns in the request. This dictionary is keyed by column id.

inputFields: Dict[str, *InputField*]

Input fields keyed by input field *InputField.id*. If an input field value has an associated input column then *InputField.selectorType* will be COLUMN and *InputField.id* will be the column id (that can be used to retrieve column data from *inputColumns*).

script: Optional[str]

An optional Python script that contains code to execute the data function.

serviceName: str

Data function service name. If we are running a Python script this will be Script. Otherwise it is the name of the module and class that will run the data function.

class df.data_transfer.**DataFunctionResponse**(**data*)

Model class for data function responses

Create a new model by parsing and validating input data from keyword arguments.

Raises ValidationError if the input data cannot be parsed to form a valid model.

outputColumns: List[*ColumnData*]

An array of output columns (to be added to the data function's input table)

outputTables: List[*TableData*]

An array of output tables

enum df.data_transfer.**DataType**(*value*)

A string enum to specify Spotfire column data types and input field data types. The Spotfire column type and equivalent Python type are listed below.

Members:

- BOOLEAN: Spotfire boolean data type, associated value should be Python bool
- STRING: Spotfire string data type, Python str
- BINARY: Spotfire binary data type, Python str value is the result of gzipping then Base64 encoding the original raw value.
- INTEGER: integer data type, equivalent to Spotfire 32 bit integer column type (Integer). Python int. Since Python ints may be 64 bit numbers use this as a return type with care.
- LONG: long data type, equivalent to Spotfire 64 bit integer column type (LongInteger). Python int. Since Python will automatically use 64 bit integers when required, use this type for return columns to avoid data truncation
- FLOAT: 64 bit double data type. Python float
- DOUBLE: 64 bit double data type. Python float. There is no difference between the double and float data type
- STRING_LIST: a JSON list of strings. Used in input fields only
- INTEGER_LIST: a JSON list of integers. Used in input fields only
- DOUBLE_lIST: a JSON list of double. Used in input fields only

Valid values are as follows:

BOOLEAN = <DataType.BOOLEAN: 'boolean'>

STRING = <DataType.STRING: 'string'>

BINARY = <DataType.BINARY: 'binary'>

INTEGER = <DataType.INTEGER: 'integer'>

LONG = <DataType.LONG: 'long'>

FLOAT = <DataType.FLOAT: 'float'>

DOUBLE = <DataType.DOUBLE: 'double'>

STRING_LIST = <DataType.STRING_LIST: 'list(string)'>

INTEGER_LIST = <DataType.INTEGER_LIST: 'list(integer)'>

DOUBLE_LIST = <DataType.DOUBLE_LIST: 'list(double)'>

class df.data_transfer.**InputField**(***data*)

A class to represent a data function input field. This is not normally constructed by the user but is created by parsing the data function input JSON.

Create a new model by parsing and validating input data from keyword arguments.

Raises ValidationError if the input data cannot be parsed to form a valid model.

contentType: Optional[str]

Optional content type for the input field (e.g. chemical/x-smiles)

data: Any

The value for the input field. The type used is based on the dataType (*DataType*) attribute

dataType: *DataType*

The data type of the input field

id: str

The id name for the input field (as defined in the data function definition)`

selectorType: Optional[*InputFieldSelectorType*]

Optional selector type. For example, the input field may be used to select a column, in which case the data value is the column id.

enum df.data_transfer.**InputFieldSelectorType**(*value*)

A string enum to specify if an input field is used to select input data from Spotfire

Members:

- COLUMN: the input field is used to select data in a column
- TABLE: the input fields is used to select a table (not currently implemented)

Valid values are as follows:

COLUMN = <InputFieldSelectorType.COLUMN: 'column'>

TABLE = <InputFieldSelectorType.TABLE: 'table'>

class df.data_transfer.**TableData**(***data*)

Model class for Spotfire data table. Instantiate this to return a new table from a data function

Create a new model by parsing and validating input data from keyword arguments.

Raises ValidationError if the input data cannot be parsed to form a valid model.

columns: List[*ColumnData*]

A list of columns in the data table

tableName: str

A suggested name for the data table

df.data_transfer.**binary_input_field**(*request*, *field*, *default_value=None*)

A helper method for retrieving a binary input field from a request.

Parameters

- **request** (*DataFunctionRequest*) – the request
- **field** (str) – the input field
- **default_value** (Optional[str]) – a default value to return if the input field has no data

Return type
Optional[str]

Returns

df.data_transfer.**boolean_input_field**(*request*, *field*, *default_value=None*)

A helper method for retrieving a boolean input field from a request.

Parameters

- **request** (*DataFunctionRequest*) – the request
- **field** (str) – the input field
- **default_value** (Optional[bool]) – a default value to return if the input field has no data

Return type
Optional[bool]

Returns

df.data_transfer.**double_input_field**(*request*, *field*, *default_value=None*)

A helper method for retrieving a double input field from a request.

Parameters

- **request** (*DataFunctionRequest*) – the request
- **field** (str) – the input field
- **default_value** (Optional[float]) – a default value to return if the input field has no data

Return type
Optional[float]

Returns

df.data_transfer.**integer_input_field**(*request*, *field*, *default_value=None*)

A helper method for retrieving an integer input field from a request.

Parameters

- **request** (*DataFunctionRequest*) – the request
- **field** (str) – the input field
- **default_value** (Optional[int]) – a default value to return if the input field has no data

Return type
Optional[int]

Returns

df.data_transfer.**string_input_field**(*request*, *field*, *default_value=None*)

A helper method for retrieving a string input field from a request.

Parameters

- **request** (*DataFunctionRequest*) – the request
- **field** (str) – the input field
- **default_value** (Optional[str]) – a default value to return if the input field has no data

Return type
Optional[str]

Returns

df.data_transfer.**string_list_input_field**(*request*, *field*, *default_value=None*)

A helper method for retrieving a string list input field from a request.

Parameters

- **request** (*DataFunctionRequest*) – the request
- **field** (str) – the input field
- **default_value** (Optional[List[str]]) – a default value to return if the input field has no data

Return type
Optional[List[str]]

Returns

3.2 chem_helper.py

Helper functions for data functions that manipulate chemical structures.

The RDKit is used to handle chemistry

For those data functions that encode or decode molecules using a *DataType* argument with an optional content type argument, if the content_type is not set, chemical/x-smiles will be used if the data_type is *STRING*, otherwise chemical/x-mdl-molfile if the data_type is *BINARY*.

df.chem_helper.**column_to_molecules**(*column*, *do_sanitize_mols=True*)

Converts a structure column to a list of molecules

Parameters

- **column** (*ColumnData*) – the input column
- **do_sanitize_mols** (Optional[bool]) – whether to run sanitize_mol on the final mols

Return type
List[Optional[Mol]]

Returns
a list of molecules

df.chem_helper.**input_field_to_molecule**(*request*, *query_data_field='queryData'*)

Converts an input field request to a molecule.

Parameters

- **request** (*DataFunctionRequest*) – the request
- **query_data_field** (str) – the input field name

Return type
Optional[Mol]

Returns
a molecule

df.chem_helper.**molecule_to_value**(*mol*, *data_type*, *content_type*)

Converts a molecule to an encoded string

Parameters

- **mol** (Mol) – the molecule

- **data_type** (*DataType*) – the data type for the output string should be *STRING* or *BINARY*
- **content_type** (Optional[str]) – an optional content type for encoding the molecule

Return type
str

Returns
encoded molecule

df.chem_helper.**molecules_to_column**(*mols*, *column_name*, *data_type*, *content_type=None*)

Converts a list of molecules to a Spotfire column.

Parameters

- **mols** (List[Optional[Mol]]) – the molecules
- **column_name** (str) – the name for the column
- **data_type** (*DataType*) – the data type for the column should be *STRING* or *BINARY*
- **content_type** (Optional[str]) – an optional content type for encoding the molecules

Return type
ColumnData

Returns
a new column

df.chem_helper.**value_to_molecule**(*v*, *data_type*, *content_type=None*, *do_sanitize_mols=True*)

Converts a string value to a molecule.

Parameters

- **v** (str) – the string value
- **data_type** (*DataType*) – the data type for the string should be *STRING* or *BINARY*
- **content_type** (Optional[str]) – molecular content type
- **do_sanitize_mols** (Optional[bool]) – whether to run sanitize_mol on the final mols

Return type
Optional[Mol]

Returns
a molecule

3.3 bio_helper.py

Helper functions for data functions that manipulate biological sequences.

Biopython SeqRecord objects are used to represent sequences

df.bio_helper.**column_to_sequences**(*column*, *id_column=None*)

Converts a Spotfire column into a list of sequence records

Parameters

- **column** (*ColumnData*) – the Spotfire column
- **id_column** (Optional[*ColumnData*]) – if set, row values from this column are used to set the sequence identifier

Return type
List[Optional[SeqRecord]]

Returns
sequence records

df.bio_helper.**query_from_request**(*request*, *input_field_name='query'*)

Converts a string input field into sequence request. The function will attempt to parse Genbank, then fasta formats. If neither works the input field data will be used a raw sequence.

Parameters

- **request** (*DataFunctionRequest*) – the request
- **input_field_name** (str) – the input field name

Return type
SeqRecord

Returns
the extracted sequence

df.bio_helper.**sequences_to_column**(*sequences*, *column_name*, *genbank_output=True*)

Converts a list of sequence records to a Spotfire column object

Parameters

- **sequences** (List[Optional[SeqRecord]]) – The list of sequences
- **column_name** (str) – The name of the new column
- **genbank_output** – if set true (the default) the returned column will contain binary encoded Genbank records (otherwise string sequences will be used)

Return type
ColumnData

Returns
Spotfire column

CHAPTER

FOUR

BUILTIN DATA FUNCTION MODULES

class df.AbComponentAnalysis.**AbComponentAnalysis**

A data function that creates an analysis of property changes in one part of an antibody while other parts are held constant

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.AntibodyNumbering.**AntibodyNumbering**

Identifies antibody chain numbering and aligns chains

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.BlastLocalColumnSearch.**BlastLocalColumnSearch**

Performs a BLAST search of user selected databases using queries from a sequence column

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.BlastLocalTextSearch.**BlastLocalTextSearch**

Performs a BLAST search of user selected databases using a query entered as text

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.BlastTableSearch.**BlastTableSearch**

Performs a blast search of the sequences in the provided table using the provided query sequence

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.BlastWebSearch.**BlastWebSearch**

Perform a remote Blast sequence search against an Entrez database

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.EnzymeRestriction.**EnzymeRestriction**

Performs restriction analysis using codes from REBASE

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.ExactMass.**ExactMass**

Calculates the exact mass for a column of structures

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.ExtractGenbankRegions.**ExtractGenbankRegions**

Extracts all regions matching a feature key and qualifier from a column of genbank sequences into new columns

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.IgBlastnLocalTextSearch.**IgBlastnLocalTextSearch**

Analyze immunoglobulin (Ig) sequences using Igblastn

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.IgBlastpLocalTextSearch.**IgBlastpLocalTextSearch**

Analyze immunoglobulin (Ig) sequences using Igblastp

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.MultipleSequenceAlignment.**MultipleSequenceAlignment**

Performs multiple sequence alignment using a variety of algorithms on an input column

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.PairwiseSequenceAlignment.**PairwiseSequenceAlignment**

Pairwise sequence alignment service, aligning sequences in an input column to a query sequence

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.ReactionTableSearch.**ReactionTableSearch**

Finds reactant and product pairs in the input columns using a reaction. The reaction must include atom maps.

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.TranslateOpenReadingFrames.**TranslateOpenReadingFrames**

Translates DNA from an input column using all open reading frames, and outputs the protein found into a new table

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

class df.TranslateSequences.**TranslateSequences**

Translates DNA in the input column to an output protein column. If the input column is a protein sequence will create a naive DNA sequence in the output

execute(*request*)

Abstract method that runs a data function.

Parameters
request (*DataFunctionRequest*) –

Return type
DataFunctionResponse

CHAPTER

FIVE

RUSE.BIO MODULES

5.1 bio_data_table_helper.py

Functions for adding sequence data to and extracting sequence data from *ruse.util.data_table.DataTable* objects

- Encoding and decoding data table cells containing sequence data
- Creating data tables and data table columns from Blast search and sequnce alignment results

ruse.bio.bio_data_table_helper.**add_blast_results_to_data_table**(*data_table*, *blast_results*, *id_to_index_map*)

Adds columns from a *ruse.bio.blast_parse.BlastResult* object into a *ruse.util.data_table.DataTable*

This is used when a sequence column from a data table is used to create a blast database. From a search against that database hit information is appended as columns to the data table. Each hit is appended to the row that contains the target sequence. The columns include the aligned sequence pair, blast expect value, blast score and number of bits.

Parameters

- **data_table** (*DataTable*) – The data table
- **blast_results** (*BlastResult*) – The blast results

Return type
None

ruse.bio.bio_data_table_helper.**add_sequences_to_data_table**(*data_table*, *sequences*, *missing_indices*, *column_name='Aligned Sequence'*, *add_dendrogram_names=False*)

Add matching/aligned sequences to a new column as sequence strings. Sequences are assigned to rows based on their id which should map to an existing row name. The sequence is added as a text field

Parameters

- **data_table** (*DataTable*) – The data table
- **sequences** (List[SeqRecord]) – List of new SeqRecords to add in a new column.
- **missing_indices** (List[int]) – Ordered list of null input sequence indices
- **column_name** (str) – New column name

- **add_dendrogram_names** (bool) – Add a second column containing sequence names that is used for construction of a dendrogram

Return type
None

ruse.bio.bio_data_table_helper.**add_sequences_to_data_table_as_genbank_column**(*data_table*, *sequences*, *missing_indices*, *column_name='Aligned Sequence'*, *add_dendrogram_names=False*)

Add matching/aligned sequences to a new column as sequence strings. Sequences are assigned to rows based on their id which should map to an existing row name. The sequence is added as an encoded Genbank record

Parameters

- **data_table** (*DataTable*) – The data table
- **sequences** (List[SeqRecord]) – List of new SeqRecords to add in a new column.
- **missing_indices** (List[int]) – Ordered list of missing null sequences
- **column_name** (str) – New column name
- **add_dendrogram_names** (bool) – Add a second column containing sequence names that is used for construction of a dendrogram

Return type
None

ruse.bio.bio_data_table_helper.**create_data_table_from_blast_results**(*query*, *blast_results*, *database_name*, *query_column_type=None*)

Creates a new *ruse.util.data_table.DataTable* from *ruse.bio.blast_parse.BlastResult*

The new table contains one row for each target sequence that matches the query:

- the aligned sequences as a text pair (query and target matches separated by a '|')
- the id of the target
- the definition of the target
- blast expect value
- blast score
- blast number of matching bits
- the target sequence record as a gzipped Base64 encoded genbank record

Parameters

- **query** (SeqRecord) – Bio.SeqRecord.SeqRecordSeqRecord object containing query sequence
- **blast_results** (*BlastResults*) – Blast results from a web search

Return type
DataTable

Returns
The *ruse.util.data_table.DataTable* data table

ruse.bio.bio_data_table_helper.**create_data_table_from_genbank_sequences**(*sequences*)

Create a *ruse.util.data_table.DataTable* from a list of Biopython `Bio.SeqReord.SeqRecord`. A single column is created containing the sequences as gzipped Base64 encoded Genbank records.

Parameters
sequences (`List`[`SeqRecord`]) – Input sequences as a list of `Bio.SeqRecord.SeqRecord`

Return type
DataTable

Returns
The *ruse.util.data_table.DataTable* data table

ruse.bio.bio_data_table_helper.**create_data_table_from_multiple_blast_searches**(*query_sequences*, *results*, *sequence_column_type=None*)

Creates a new `DataTable` from `MultipleBlastResults` obtained by performing blast alignment searched using the query sequences

The new table contains one row for each target sequence that matches the query:

- the aligned sequences as a text pair (query and target matches separated by a '|')
- the id of the query
- the definition of the query
- the id of the target
- the definition of the target
- blast expect value
- blast score
- blast number of matching bits
- the query sequence record either as a gzipped Base64 encoded genbank record or sequence string
- the target sequence record either as a gzipped Base64 encoded genbank record or sequence string

Note that target results from NT databases may include full length chromosomes and Genbank entries for those are large, in which case the string encoding should be preferred

Parameters

- **query_sequences** (`List`[`SeqRecord`]) – query sequences
- **results** (*MultipleBlastResults*) – multiple blast search results
- **sequence_column_type** (`Optional`[`str`]) – should be either 'genbank' or 'string', sets how query and target sequences are encoded in the table

Return type
DataTable

Returns
The *ruse.util.data_table.DataTable* data table

ruse.bio.bio_data_table_helper.**create_data_table_from_sequences**(*sequences*)

Create a *ruse.util.data_table.DataTable* from a list of Biopython `Bio.SeqRecord.SeqRecord`. The sequences are added as strings. Two columns are created: an id column and a sequence column

Parameters
sequences (`List`[`SeqRecord`]) – Input sequences as a list of `Bio.SeqRecord.SeqRecord`

Return type
DataTable

Returns
The data table

ruse.bio.bio_data_table_helper.**data_table_cell_to_sequence**(*data_table*, *row_no*, *column_no*)

Given a data table and cell decodes the cell to to a Biopython SeqRecord.

The content-type definition for the column is used to determine the cell encoding

Parameters

- **data_table** (*DataTable*) – The data table
- **row_no** (`int`) – Row index
- **column_no** (`int`) – Column index

Return type
`SeqRecord`

Returns
Sequence as `Bio.SeqRecord.SeqRecord`

ruse.bio.bio_data_table_helper.**data_table_column_to_sequence**(*data_table*, *column_no*, *missing_indices=[]*)

Decodes a data table sequence column to a list of Biopython SeqRecords. The content-type definition for the column is used to determine the column encoding

Parameters

- **data_table** (*DataTable*) – The data table
- **column_no** (`int`) – column index to decode

Return type
[<class 'Bio.SeqRecord.SeqRecord'>]

Returns
Tuple containing list of SeqRecords contained in column and list of missing ids

ruse.bio.bio_data_table_helper.**genbank_base64_str_to_sequence**(*data*, *row_index*)

Decodes a string to a Biopython SeqRecord

The string must have been created using a GenBank formatted sequence record that has been gzipped then encoded in Base64(utf8)

Parameters
data (`str`) – the string

Return type
`SeqRecord`

Returns
Sequence as `Bio.SeqRecord.SeqRecord`

ruse.bio.bio_data_table_helper.**genbank_cell_to_sequence**(*data_table*, *column_no*, *row_index*)

Given a data table and cell decodes the cell to to a Biopython SeqRecord.

The cell must have been created using a GenBank formatted sequence record that has been gzipped then encoded as a Base64(utf8) string.

Parameters

- **data_table** – The data table
- **column_no** (int) – column index of cell
- **row_index** (int) – row index of cell

Return type
SeqRecord

Returns
Sequence as Bio.SeqRecord.SeqRecord

ruse.bio.bio_data_table_helper.**sequence_to_genbank_base64_str**(*record*)

Converts a Biopython sequence record to an encoded string. The sequenced is converted to a Genbank format string, then gzipped and encoded in Base64(utf8)

Parameters
record (Optional[SeqRecord]) – Sequence as Bio.SeqRecord.SeqRecord

Return type
str

Returns
Encoded string

5.2 blast_parse.py

Copyright (C) 2017-2022 Glysade, LLC

Classes for parsing the results from Blast searches

class ruse.bio.blast_parse.**BlastResult**(*target_id*, *target_def*, *target_length*, *align_length*, *query*, *target*, *evalue*, *score*, *bits*, *accession*, *query_start*, *query_end*, *search_type*)

Class to hold an NCBI Blast result, comprising a single alignment between query and target sub sequences.

Attributes:

- All constructor parameters
- target_record (SeqRecord) the target sequence downloaded from Entrez

Constructs the class from NCBI blast search output fields

Parameters

- **target_id** (str) – the id or the target/database sequence
- **target_def** (str) – the definition for the target sequence
- **target_length** (int) – the full length of the target seqence
- **align_length** (int) – the length of the matching alignment
- **query** (str) – The matching query sequence

- **target** (str) – The matching target sequence
- **evalue** (float) – The Blast expect value for the alignment
- **score** (int) – The Blast score
- **bits** (float) – The Blast bit count
- **accession** (str) – Target accession

complement_target_sequence()

Reverse complements the target record sequence if the match is on the other strand (blastn only)

Determination of reverse complement match handles ambiguous DNA in the query, but not on the target. SequenceMatcher has an exhaustive matcher, but this is too slow for use on larger target sequences

Return type
None

data_table_name()

Used when the blast hit is the result of searching a blast database created from sequences extracted from a data table column

Return type
str

Returns
Sequence name

classmethod fasta_header_identifier(*header*)

Attempt to extract an identifier that can be used as an Entrez query from a header string. The header string is a list of identifiers separated by the '|' character- sit the first line of FASTA files and can be extracted from definitions or ids in Blast results

Parameters
header (str) – string to extract identifier from

Return type
Optional[str]

identifier()

Parse target ids and target definition for an identifier that can be used for Entrez search

Return type
str

Returns
an identifier for Entrez search

class ruse.bio.blast_parse.**BlastResults**

A class to hold results for a query. A container for a list of *BlastResult*

Attributes:

- query_id: query identifier
- query_def: query definition
- database: the name of the database searched
- search_type: a *ruse.bio.blast_search.BlastSearchType* describing the blast search parameters
- hits: List of associated *BlastResult* results

No argument constructor

complement_target_sequence()

For BLASTN searches replace the target sequence by the complement if it looks like the match is on the other strand :rtype: None :return:

parse(*file*)

Parses query search results from a blast result file. There should only be one result in the file

Parameters
file (str) – file-like handle to blast search results in XML format

Return type
None

read_record(*blast_record*)

Processes a blast search record from Biopython extracting target hits into a list of class:*BlastResult* results

Parameters
blast_record – record returned from Biopython `Bio.Blast.NCBIXML.read()` or `Bio.Blast.NCBIXML.parse()`

Return type
None

retrieve_local_targets()

Retrieve from a local database full length sequences for all targets without full length records

Return type
None

retrieve_targets()

Retrieve from NCBI entrez genbank records for all targets

Return type
None

to_data()

Maps target ids to dictionaries of results

Return type
Dict[str, List[Dict[str, Union[str, int, float]]]]

Returns
a map of target identifiers to Blast class:*BlastResult* results

to_mapping()

Maps target_ids to results

Return type
Dict[str, *BlastResult*]

Returns
a map of target identifiers to Blast class:*BlastResult* results

class ruse.bio.blast_parse.**MultipleBlastResults**

A class to store the results of multiple query searches against a single Blast database

Attributes:

- query_hits: a list of *BlastResults*, with one result class for each query
- database: the name of the database searched

- search_type: a `BlastSearchType` describing the blast search parameters

Empty constructor

complement_target_sequence()

For blastn searches, if it looks like the match is on the other strand, reverse complements the matching target record, see *BlastResult.complement_target_sequence()*

Return type
None

parse(*file*)

Uses Biopython's `Bio.Blast.NCBIXML` parser to extract relevant information from a Blast search's XML output. Each query's search is added as a *BlastResults*.

Parameters
file (str) – inout handle to blast results in XML format

Return type
None

retrieve_local_targets()

Retrieves all full-length target sequences from the local blast or emboss database (assumes search is local) :return:

retrieve_targets(*retrieve_missing_locally=False*)

Retrieves target records from NCBI Entrez non-redundant databases

Parameters
retrieve_missing_locally (bool) – if True extract missing full length target sequences from the local database (assumes search is local)

Return type
None

truncate_target_sequences(*maximum_length=10000*)

Truncate full length target sequences

Parameters
maximum_length (int) – Maxumum target length

Return type
None

5.3 blast_search.py

Copyright (C) 2017-2022 Glysade, LLC

Classes and functions for performing Blast searches against local and NCBI-hosted databases

class ruse.bio.blast_search.**BlastCreateAndSearch**

A class that creates a blast database on the fly from supplied sequences and performs a single search against that database

Attributes:

- query: A `Bio.SeqRecord.SeqRecord` query sequence
- search_name: the name for the search

- search_type: Search type of *BlastSearchType* value
- options: Blast search options as detailed in *BlastSearch.multiple_query_search_blast_database()*
- search: A *BlastSearch* object that runs the local search
- database: The name for the database
- target_sequences: A list of `Bio.SeqRecord.SeqRecord` target sequences
- query_type: The query sequence type. Either 'prot' or 'nucl'
- target_type: The target sequence type. Either 'prot' or 'nucl'

Empty constructor

clean_search()

Remove all search and database files

Return type
None

search_blast_sequences(*query*, *sequences*, *search_type=None*, *search_name=None*, *query_type=None*, *options={}*)

Creates a blast database from the target sequences, then performs blast search using the query sequence

Parameters

- **query** (SeqRecord) – The `Bio.SeqRecord.SeqRecord` query sequence
- **sequences** (List[SeqRecord]) – A list of `Bio.SeqRecord.SeqRecord` target sequences
- **search_type** (Union[*BlastSearchType*, str, None]) – Optional search type of *BlastSearchType* value. Will guess from sequences if not present
- **search_name** (Optional[str]) – Optional search name. Assigned using UUID if not present
- **options** (Dict[str, Union[str, int, float]]) – Blast search options as detailed in *BlastSearch.multiple_query_search_blast_database()*

Return type
str

Returns

class ruse.bio.blast_search.**BlastDatabase**(*name=None*)

A class to handle creation of blast databases

Attributes:

- name: the name of the database
- sequence_count: the number of sequences in the database

Constructor

Parameters
name (Optional[str]) – the name of the database

build_database(*sequences*, *database_name=None*, *type=None*)

Constructs a local database from the input sequences

Parameters

- **sequences** (List[SeqRecord]) – a list of Bio.SeqRecord.SeqRecord sequence records
- **database_name** (Optional[str]) – the name of the database. If None it will be created from a UUID
- **type** (Optional[str]) – The database type (prot or ncul). If None it will be set from the sequences

Return type
str

Returns
the name of the database

clean_database()

Removes all database files

Return type
None

class ruse.bio.blast_search.**BlastSearch**

A class to manage a blast search of multiple queries against a local database

Attributes:

- queries: A list of Bio.SeqRecord.SeqRecord query sequences
- database_name: The name of the target database
- search_type: Search type of *BlastSearchType* value
- options: Dict of allowable options (see *BlastSearch.multiple_query_search_blast_database()*)

Empty constructor

clean_search()

Remove all search results

Return type
None

multiple_query_search_blast_database(*queries*, *database_name*, *search_type*, *search_name=None*, *options={}*)

Searches multiple queries against a local Blast database, uses :func"*Bio.Blast.Applications.NcbiblastxCommandline*

Options can be any option available to the particular blast program. Default options are:
num_threads: number of processors max_target_seqs: 10 max_hsps: 1 (multiple hsps are not processed by the parse classes)

Parameters

- **queries** (List[SeqRecord]) – A list of Bio.SeqRecord.SeqRecord query sequences
- **database_name** (str) – The name of the database to search
- **search_type** (Union[*BlastSearchType*, str]) – Search type of *BlastSearchType* value
- **search_name** (Optional[str]) – The name of the search (if none a UUID will be used to assign ont)
- **options** (Dict[str, Union[str, int, float]]) – Optional dict of allowable options as option name, value pairs

Returns
The search name

`output_file()`

Return type
str

Returns
the name of the xml output file

`search_blast_database(query, database_name, search_type, search_name=None, options={})`

Search a single query against a blast database. Delegates to *multiple_query_search_blast_database()*

Parameters

- **query** (SeqRecord) – A Bio.SeqRecord.SeqRecord query sequence
- **database_name** (str) – The name of the database to search
- **search_type** (Union[*BlastSearchType*, str]) – Search type of *BlastSearchType* value
- **search_name** (Optional[str]) – The name of the search (if none a UUID will be used to assign ont)
- **options** (Dict[str, Union[str, int, float]]) – Optional dict of allowable options (see *BlastSearch.multiple_query_search_blast_database()*).

Return type
str

Returns
The search name

enum ruse.bio.blast_search.**BlastSearchType**(*value*)

A Enum class for the different types of Blast searches

Members:

- BLASTN: compare nucl query with nucl
- TBLASTX: compare nucl query with nucl (using translations/both strands to protein sequences)
- TBLASTN: compare protein query (using translations/both strands) with nucl
- BLASTP: compare protein with protein
- BLASTX: compare nucl query (using translations/both strands) with protein

Valid values are as follows:

BLASTN = <BlastSearchType.BLASTN: 1>

TBLASTX = <BlastSearchType.TBLASTX: 2>

TBLASTN = <BlastSearchType.TBLASTN: 3>

BLASTP = <BlastSearchType.BLASTP: 4>

BLASTX = <BlastSearchType.BLASTX: 5>

The Enum and its members also have the following methods:

query_type()

The expected query type for this search

Return type
str

Returns
"nucl" or "prot"

database_type()

The expected database type for this search

Return type
str

Returns
"nucl" or "prot"

exe()

Determine full patch to command line search program

Return type
str

Returns
full path to executable

classmethod from_string(*value*)

Generates a search type enum from the string search name

Parameters
value – The string name fo a search type

Return type
BlastSearchType

Returns
The equivalent Enum class

classmethod default_search_type(*query_type*, *target_type*)

Returns default search type from query and target sequence type

Parameters

- **query_type** (str) – one of 'nucl' or 'prot'
- **target_type** (str) – one of 'nucl' or 'prot'

Return type
BlastSearchType

Returns
Blast search type

class ruse.bio.blast_search.**BlastWebDatabase**(*name*, *protein*)

Create new instance of BlastWebDatabase(name, protein)

name: str

Alias for field number 0

protein: bool

Alias for field number 1

class ruse.bio.blast_search.**BlastWebSearch**

A class to do a BLAST search using the QBLAST server at NCBI or a cloud service provider, using `Bio.Blast.NCBIWWW()`

Attributes:

- query: The `Bio.SeqRecord.SeqRecord` query sequence
- database_name: The remote database to search
- search_name: Search name. Assigned using UUID if not present
- search_type: Search type of *BlastSearchType* value.
- options: Blast search options

Empty constructor

clean_search()

Cleans search results from hard drive

Return type
None

output_file()

Return type
str

Returns
the xml result file

search_blast_database(*query*, *database_name=None*, *search_type=None*, *search_name=None*, *hitlist_size=100*, *query_type=None*, *expect=None*, *word_size=None*, *options={}*)

Performs a remote blast search against the NCBI QBLAST server

Parameters

- **query** (SeqRecord) – The `Bio.SeqRecord.SeqRecord` query sequence
- **database_name** (Optional[str]) – The remote database to search. If not present use 'nt' for nucleotide query sequence and 'nr' for a protein query
- **search_type** (Union[*BlastSearchType*, str, None]) – Optional search type of *BlastSearchType* value. Will guess from sequences and database if not present
- **query_type** (Optional[str]) – Optional query type- should be one of 'nucl' or 'prot', if not present guess from sequence
- **search_name** (Optional[str]) – Optional search name. Assigned using UUID if not present
- **options** (Dict[str, Union[str, int, float]]) – Dictionary of Qblast search options, passed to `Bio.Blast.NCBIWWW()`

Return type
str

Returns
search name

ruse.bio.blast_search.**full_path_to_blast_exe**(*exe*)

Finds the full path to an NCBI executable

Parameters
exe (str) – the name of the program

Return type
str

Returns
full path to the program

5.4 bio_util.py

A number of utility functions to augment Biopython functionality

enum ruse.bio.bio_util.**SequenceType**(*value*)

An enumeration.

Valid values are as follows:

PROTEIN = <SequenceType.PROTEIN: 1>

DNA = <SequenceType.DNA: 2>

ruse.bio.bio_util.**entrez_email**()

Returns the email to be passed to NCBI when retrieving information from NCBI using the Entrez interface. This is set using the ENTREZ_EMAIL environment variable

Return type
str

Returns
Email to pass to NCBI

ruse.bio.bio_util.**extract_feature**(*record*, *feature_id=None*, *qualifier_name=None*, *qualifier_value=None*, *type=None*)

Extracts a feature from a sequence record Can search by type, feature_id or both qualifier_name and qualifier_value Matching is case insensitive

Parameters

- **record** (SeqRecord) – :class'*SeqRecord* object to extract feature from
- **feature_id** (Optional[str]) – feature id string criteria
- **qualifier_name** (Optional[str]) – qualifier name criteria
- **qualifier_value** (Optional[str]) – qualifier value criteria
- **type** (Optional[str]) – feature type restriction

Return type
SeqFeature

Returns
matching SeqFeature

ruse.bio.bio_util.**is_dna**(*seq*)

Return true if seq could be a dna sequence. Note that there are many protein sequences that could be dna sequences

Parameters
seq (str) – the sequence to test

Return type
bool

Returns
True if the sequence contains all DNA alphabet

ruse.bio.bio_util.**is_protein**(*seq*)

Return true if seq could be a protein sequence, note any dna sequence could also be a protein sequence, so not is_dna may be a better test

Parameters
seq (str) – The sequence to test

Return type
bool

Returns
True if the sequence contains all Protein alphabet

ruse.bio.bio_util.**sequences_to_file**(*file*, *sequences*, *format='fasta'*)

Writes sequences to a file

Parameters

- **file** (str) –
- **sequences** (List[SeqRecord]) –
- **format** (str) –

Return type
None

ruse.bio.bio_util.**sub_sequence**(*record*, *start*, *end*)

Extract a sequence range from a SeqRecord while copying annotations

Parameters

- **record** (SeqRecord) – sequence to extract range from
- **start** (int) – start of sequence range
- **end** (int) – end of sequence range

Return type
Optional[SeqRecord]

Returns
sequence range

5.5 blast-utils.py

A set of utility functions for handing NCBI Blast search and Entrez queries

class ruse.bio.blast_utils.**BlastRecord**(*id*, *record*)

Stores a local blast database sequence record and the id used to retrieve it

Attributes:

- id: identifier that was used to extract the local record
- record: sequence as `Bio.SeqRecord.SeqRecord`

Create new instance of BlastRecord(id, record)

class ruse.bio.blast_utils.**BlastRecords**

Class to facilitate retrieval of sequence records from a local database Uses the external program *blastdbcmd* to export records from the local database

Attributes:

- ids: list of ids retrieved from the local database
- records: retrieved records as list of `Bio.SeqRecord.SeqRecord`
- record_map: Dictionary mapping ids to records
- database: The local database
- error: Any blastdbcmd error

Empty constructor

retrieve_from_local_blast_database(*database*, *ids*)

Retrieves a list of target sequences from a local blast database

Parameters

- **database** (str) – The name of the local database
- **ids** (List[str]) – The ids to retrieve

Return type
List[*BlastRecord*]

class ruse.bio.blast_utils.**SequenceMatcher**

A class to facilitate matching ambiguous dna sequences

classmethod match_sequences(*seq*, *sub*)

A class method to determine the the first position that an ambiguous dna subsequence can match a dna sequence

Parameters

- **seq** (str) – The main sequence
- **sub** (str) – Subsequence

Return type
int

Returns
The position of start of the first match of the subsequence to the sequence, or -1 if no such match exists

ruse.bio.blast_utils.**retrieve_entrez_records**(*db*, *target_ids*)

Uses `Bio.Entrez.efetch()` to retrieve reference sequences from NCBI

Parameters

- **db** (`str`) – Etnrez database ("nt" or "nr")
- **target_ids** (`List[str]`) – List of Entrez/NCBI identifiers

Return type
`List[SeqRecord]`

Returns
a List of Genbank entries as `Bio.SeqRecord.SeqRecord` objects

5.6 sequence_align.py

Copyright (C) 2017-2022 Glysade, LLC

A classs to handle multiple sequence alignment (MSA)

class ruse.bio.sequence_align.**MultipleSequenceAlignment**

A class to perform MSA using ClustalO on input sequences

Attributes:

- options: a key-value dictionary of options to be passed to ClustalO
- input_sequences: a List of `Bio.SeqRecord.SeqRecord` sequence objects to be aligned
- aligned_sequences: the input sequences aligned as a List of `Bio.SeqRecord.SeqRecord`
- alignment: Biopython parsing of alignment as `Bio.Align.Bio.MultipleSeqAlignment`
- basename: UUID string for common resource naming
- tree: guide tree for alignment as `Bio.Nexus.Trees.Tree`
- distances: distance of each sequence from root in guide tree

Empty constructor

add_distances()

Add an evolutionary distance for each sequence using the guide tree or fasttree

Return type
`None`

align_sequences(*options*, *sequences*, *alignment_method=SequenceAlignmentMethod.CLUSTALO*)

Performs MSA using ClustalO on the input sequences and creates alignment attributes. A guide tree with also be created if there are 3 or more input sequences

Parameters

- **options** (`Dict[str, Union[str, int, float]]`) – a key-value dictionary of options to be passed to ClustalO
- **sequences** (`List[SeqRecord]`) – a List of `Bio.SeqRecord.SeqRecord` sequence objects to be aligned

Return type
None

`clean_files`()
Delete MSA ClustalO files

Return type
None

`copy_features_to_aligned_sequences`()
Copies sequence features from input sequences to aligned sequences

Return type
None

enum ruse.bio.sequence_align.**SequenceAlignmentMethod**(*value*)
An enumeration.

Valid values are as follows:

CLUSTALO = <SequenceAlignmentMethod.CLUSTALO: 1>

MUSCLE = <SequenceAlignmentMethod.MUSCLE: 2>

CLUSTALW = <SequenceAlignmentMethod.CLUSTALW: 3>

ruse.bio.sequence_align.**copy_features_to_gapped_sequence**(*in_seq*, *out_seq*, *local=False*)
Copies sequence features from an input sequence to another sequence, adjusting for inserted or deleted gaps

Parameters
- **in_seq** (SeqRecord) – The input sequence with no gaps as a Bio.SeqRecord.SeqRecord object
- **out_seq** (SeqRecord) – The gapped aligned output sequence as a Bio.SeqRecord.SeqRecord object
- **local** (bool) – Set true if the alignment is local (partial continuous match)

Return type
None

5.7 phylo_tree.py

Copyright (C) 2017-2022 Glysade, LLC

Classes to handle the generation and processing of phylogenetic trees from multiple sequence alignments.

class ruse.bio.phylo_tree.**PhyloTree**(*tree*)
A class to facilitate operations on pylogenetic trees. Handles conversion of Biopython trees to JSON and calculation of leaf distances

Attributes:
- tree: Bioython pyhlogenetic tree (class Bio.Phylo.BaseTree.Tree)
- data_tree: Python dictionary representation of tree, that is JSON compatible

Constructor. Converts Biopython tree object of Bio.Phylo.BaseTree.Tree to a JSON compatible dictionary.

Parameters
tree (Tree) – Bioython pyhlogenetic tree

leaf_distances(*last_distance_only=False*)

Non-recursive function to find the distance to the leaves

Parameters
last_distance_only (bool) – set true to return the distance from leaf to parent rather than cumulative distance to the root. This is appropriate for UPGMA trees (such as those created by ClustalO) which have constant root to leaf distances.

Return type
Dict[str, float]

Returns
A dictionary mapping leaf names to distances

tree_to_data_recursive()

Converts the guide tree to a nested dictionary data structure. Uses a recursive method used to validate `_tree_to_data()`

Return type
Dict

Returns
tree as python data structure

class ruse.bio.phylo_tree.**PhyloTreeBuilder**

A class to build a phylogenetic tree from a multiple sequence alignment using an external application

Attributes:

- input_aln_file: Clustal format multiple sequence alignment file used to construct tree
- basename: UUID string used to construct output files
- alignment: a `Bio.AlignIO` object of input alignment

Empty constructor

build_fasttree_tree(*input_aln_file*)

Creates a Biopython Tree object (of class `Bio.Phylo.BaseTree.Tree`) by running FastTree (see *http://www.microbesonline.org/fasttree/*).

Assumes that FastTree binaries are available in this distribution. For nucleotide sequences the -gtr and -nt arguments to FastTree are set. No arguments are set for protein alignment.

Parameters
input_aln_file (str) – path of clustal alignment file

Return type
Tree

Returns
tree created from running FastTree

build_raxml_tree(*input_aln_file*)

Creates a Biopython Tree object (of class `Bio.Phylo.BaseTree.Tree`) by running raxml (see *https://sco.h-its.org/exelixis/web/software/raxml/index.html*). The raxml executable must be on the PATH.

This is not a recommended method to create trees. It is very time consuming and deprecated in favor of ExaML.

Parameters
input_aln_file (str) – path of clustal alignment file

Return type
Tree

Returns
tree created from running RaxML

`cleanup()`

Removes all output and intermediate files :return:

`classmethod find_fasttree_exe()`

Returns the full path to an FastTree (*http://www.microbesonline.org/fasttree/*) executable. Windows and Linux FastTree binaries are included in this distribution.

Return type
Optional[str]

Returns
full path to the program, of None if no executable can be found

CHAPTER

SIX

RUSE.CHEM MODULES

6.1 chem_data_table_helper.py

Functions for adding molecules to and extracting molecules from *ruse.util.data_table.DataTable* objects

- Encoding and decoding data table cells containing molecules
- Creating data tables and data table columns from lists of molecules

Uses RDKit for chemical structure handling

ruse.chem.chem_data_table_helper.**add_mols_as_data_table_column**(*data_table*, *mols*, *content_type='chemical/x-mdl-molfile'*, *title='Molecular conformation'*, *three_dimensional=None*)

Add the molecules as a new column to a data table. They should have an SD tag containing the row index

Parameters

- **data_table** (*DataTable*) – The data table
- **mols** (Iterable[Mol]) – A iterator of rdkit.Chem.rdchemRdKit.Mol molecules
- **content_type** (str) – The content type to store the molecules as
- **title** (str) – Optional column title
- **three_dimensional** (Optional[bool]) – set true if the molecules are 3D. If None ruse.rdkit.rdkit_util.is_three_dimensional() is used to determine 3D nature

Return type
None

ruse.chem.chem_data_table_helper.**create_data_table_from_mols**(*mols*, *content_type='chemical/x-smiles'*, *title='Molecule'*)

Create a new table with a single column containing the input molecules

Parameters

- **mols** (Iterable[Mol]) – A iterator of rdkit.Chem.rdchemRdKit.Mol molecules
- **content_type** (str) – The chemical mime type specifying the format to be used to store the molecules
- **title** (str) – title to use for new column

Return type
DataTable

Returns
The new data table as *ruse.util.data_table.DataTable*

ruse.chem.chem_data_table_helper.**create_data_table_from_rgroup_decomposition**(*decomp*, *core_format='chemical/x-mdl-molfile'*)

Creates a data table from the results of an R Group decomposition. Multiple structure columns are created in the output table: one for the structures, one for matching cores and a column for each rgroup

Parameters

- **decomp** (*RgroupDecomposer*) – results of core decomposition of a set of molecules as *ruse.rdkit.rgroup.RgroupDecomposer*
- **core_format** (str) – the chemical mime type to encode the cores in, should be chemical/x-mdl-molfile or chemical/x-smarts. Defaults to chemical/x-mdl-molfile

Return type
DataTable

Returns
the data table

ruse.chem.chem_data_table_helper.**create_data_table_from_rocs_search**(*rocs*)

Create a new *ruse.util.data_table.DataTable* from the results of an OpenEye ROCS search. The new table has columns for the query and target structures, rank, combo score, shape and color tanimoto.

Parameters
rocs – ruse.services.rocs_service.Rocs object containing ROCS search result

Return type
DataTable

Returns
the new data table

ruse.chem.chem_data_table_helper.**data_table_column_to_local_files**(*data_table*, *column_index*, *prefix=None*, *num_mols_per_file=1*)

Writes all structures in a column in batches to local MOL files. The files will be named <prefix>_<starting_row_index>.mol

Parameters

- **data_table** (*DataTable*) – The data table to extract the molecules from
- **column_index** (int) – Column number of molecular column
- **prefix** (Optional[str]) – File name prefix (UUID will be used if this is not set)
- **num_mols_per_file** (int) – Batch size, defaults to 1

Return type
List[str]

Returns
A list of the file names created

ruse.chem.chem_data_table_helper.**data_table_column_to_mols**(*data_table*, *column_index*, *include_empty=False*)

Convert a column in a data table containing chemical structures to a generator of RDKit molecules

Parameters

- **data_table** (*DataTable*) – the data table as *ruse.util.data_table.DataTable*
- **column_index** (int) – index of the column containing chemical structures
- **include_empty** (bool) – set True to include empty cells as None in the iterator

Return type
Iterable[Mol]

Returns
A generator of rdkit.Chem.rdchemRdKit.Mol molecules

ruse.chem.chem_data_table_helper.**merge_data_table_from_rdkit_rgroup_decomposition**(*table*, *decomp*, *core_format='chemical/x-mdl-molfile'*)

Merges the results of an R Group decomposition with an existing data table. Multiple structure columns are created in the output table: one for matching cores and a column for each rgroup

Parameters

- **table** (*DataTable*) – original input data table
- **decomp** (*Rgroup*) – results of core decomposition of a set of molecules as *ruse.rdkit.rgroup.RgroupDecomposer*
- **core_format** (str) – the chemical mime type to encode the cores in, should be chemical/x-mdl-molfile or chemical/x-smarts. Defaults to chemical/x-mdl-molfile

Return type
DataTable

Returns
the data table

ruse.chem.chem_data_table_helper.**merge_data_table_from_rgroup_decomposition**(*table*, *decomp*, *core_format='chemical/x-mdl-molfile'*)

Merges the results of an R Group decomposition with an existing data table. Multiple structure columns are created in the output table: one for matching cores and a column for each rgroup

Parameters

- **table** (*DataTable*) – original input data table
- **decomp** (*RgroupDecomposer*) – results of core decomposition of a set of molecules as *ruse.rdkit.rgroup.RgroupDecomposer*
- **core_format** (str) – the chemical mime type to encode the cores in, should be chemical/x-mdl-molfile or chemical/x-smarts. Defaults to chemical/x-mdl-molfile

Return type
DataTable

Returns
the data table

CHAPTER

SEVEN

RUSE.RDKIT MODULES

7.1 rdkit_utils.py

Copyright (C) 2017-2022 Glysade, LLC

Utility functions for handling chemical structures using RDKit (www.rdkit.org)

enum ruse.rdkit.rdkit_utils.**RDKitFormat**(*value*)

Enumerated class for structure formats handled by RDkit interface

Members:

- sdf
- pdb
- smi
- sma
- rxn

Valid values are as follows:

sdf = <RDKitFormat.sdf: 1>

pdb = <RDKitFormat.pdb: 2>

smi = <RDKitFormat.smi: 3>

sma = <RDKitFormat.sma: 4>

rxn = <RDKitFormat.rxn: 5>

ruse.rdkit.rdkit_utils.**encode_mol**(*type*, *mol*)

Converts an RDKit molecule to a molecular string which is then gzipped and Base64 encoded.

Parameters

- **type** (*RDKitFormat*) – The structure format for the string as *RDKitFormat*
- **mol** (Mol) – RDkit molecule, class rdkit.Chem.rdchem.Mol

Return type

str

Returns

The molecular string, gzipped then Base64 encoded

ruse.rdkit.rdkit_utils.**file_to_format**(*file*)

Elucidates the RDKit structure format from a filename

Parameters
file (str) – the name of a file

Return type
RDKitFormat

Returns
The structure format for the compounds in the file as *RDKitFormat*

ruse.rdkit.rdkit_utils.**file_to_mols**(*file*, *type=None*)

Reads a list of RDKit molecules from a file

Parameters

- **file** (str) – The name of the file
- **type** (Optional[*RDKitFormat*]) – The structure format for the compounds in the file as *RDKitFormat*. If None (the default) the type will be inferred from the file name

Return type
List[Mol]

Returns
a List of rdkit.Chem.rdchem.Mol RDKit molecules

ruse.rdkit.rdkit_utils.**is_three_dimensional**(*mol*)

Return true is this mol was created from a three dimensional SD file

Parameters
mol (Mol) –

Return type
bool

Returns
True if the SD file header contains 3D

ruse.rdkit.rdkit_utils.**mol_supplier**(*file*, *type=None*)

Wrapper round the RDKit molecular suppliers. Opens the file using the correct supplier and and yields RdKit molecules.

Parameters

- **file** (str) – The name of the file
- **type** (Optional[*RDKitFormat*]) – The structure format for the compounds in the file as *RDKitFormat*. If None (the default) the type will be inferred from the file name

Return type
Iterable[Mol]

Returns
yields RDkit molecules of class rdkit.Chem.rdchem.Mol

ruse.rdkit.rdkit_utils.**mol_to_string**(*type*, *mol*, *isomeric_smiles=True*)

Converts an RDKit molecule to a molecular string

Parameters

- **type** (*RDKitFormat*) – The structure format for the string as *RDKitFormat*

- **mol** (Mol) – RDkit molecule, class rdkit.Chem.rdchem.Mol
- **isomeric_smiles** – If True set smiles to be isomeric (default)

Return type
str

Returns
The molecular string

ruse.rdkit.rdkit_utils.**mol_writer**(*file*, *type=None*)

Wrapper round the RDKit writers. Returns the correct writer to a file for the molecular format

Parameters

- **file** (str) – The name of the file
- **type** (Optional[*RDKitFormat*]) – The structure format for the compounds in the file as *RDKitFormat*. If None (the default) the type will be inferred from the file name

Return type
Union[SDWriter, SmilesWriter]

Returns
yields the appropriate writer for the molecular type and file

ruse.rdkit.rdkit_utils.**mols_to_file**(*file*, *mols*, *type=None*)

Writes a list of RDKit molecules to a file

Parameters

- **file** (str) – The name of the file
- **mols** (Iterable[Mol]) – an Iterator of rdkit.Chem.rdchem.Mol RDKit molecules
- **type** (Optional[*RDKitFormat*]) – The structure format for the compounds in the file as *RDKitFormat*. If None (the default) the type will be inferred from the file name

Return type
None

ruse.rdkit.rdkit_utils.**print_mol_information**(*mol*)

Prints information about a molecule. For debugging purposes

Parameters
mol (Mol) –

Return type
None

Returns

ruse.rdkit.rdkit_utils.**remove_atom_mappings**(*mol*, *remove_h=True*)

Removes all atom mappings from a molecule

Parameters
mol (Mol) – the molecule

Return type
Mol

ruse.rdkit.rdkit_utils.**remove_explicit_hydrogens**(*mol*)

Removes all explicit hydrogens and hydrogen atoms from a molecule

Parameters
mol (Mol) –

Return type
Mol

Returns

ruse.rdkit.rdkit_utils.**string_to_mol**(*type*, *mol_string*, *do_sanitize_mol=True*)

Converts a string to an RDKit molecule

Parameters

- **type** (*RDKitFormat*) – The structure format for the string as *RDKitFormat*
- **mol_string** (str) – Molecular string
- **do_sanitize_mol** (Optional[bool]) – whether to run sanitize_mol on the final mol

Return type
Union[Mol, ChemicalReaction, None]

Returns
RDkit molecule, class rdkit.Chem.rdchem.Mol

ruse.rdkit.rdkit_utils.**string_to_mols**(*type*, *mol_string*)

Converts a string to an RDKit molecule

Parameters

- **type** (*RDKitFormat*) – The structure format for the string as *RDKitFormat*
- **mol_string** (str) – Molecular string

Return type
List[Mol]

Returns
RDkit molecule, class rdkit.Chem.rdchem.Mol

ruse.rdkit.rdkit_utils.**type_to_format**(*type*)

Given a chemical mime type returns the equivalent *RDKitFormat* Enumerated type

Parameters
type (str) – Chemical mime type

Return type
RDKitFormat

Returns
RDKit format

7.2 rgroup.py

Contains classes for performing r group decomposition

class ruse.rdkit.rgroup.**Attachment**(*attachment_point: AttachmentPoint*, *mol: Mol*)

A named tuple that describes an R-group and associated attachment point

Attributes:

- attachment_point: the location of this R group on the core as an *AttachmentPoint* object

- mol: the R-group as a `rdkit.Chem.Mol` object

Create new instance of Attachment(attachment_point, mol)

attachment_point: *AttachmentPoint*

Alias for field number 0

heavy_sidechain()

Return type
None

Returns
True if this sidechain is not a hydrogen

mol: Mol

Alias for field number 1

relabel_sidechain()

Relabel attachment atom with R group number- should only be done once all matching is completed

Return type
None

class ruse.rdkit.rgroup.**AttachmentPoint**(*core_index: int*, *group_num: int*, *required: bool*)

A named tuple that stores information about core R group attachment points

Attributes:

- core_index: the atom index of the attachment point on the core
- group_num: the associated R group number

Create new instance of AttachmentPoint(core_index, group_num, required)

core_index: int

Alias for field number 0

group_num: int

Alias for field number 1

required: bool

Alias for field number 2

class ruse.rdkit.rgroup.**Core**(*index*, *mol*, *allow_rgroups_at_any_free_valance=False*)

A class to represent a core query for decomposition

If the core contains explicit R groups they are used to define attachment points- otherwise any atom in the core may be used as an attachment point

Attributes:

- index: the index of the core in a multi-core decomposition
- mol: a class:*rdkit.Chem.Mol* molecule that contains the original core definition
- attachment_points: a list of R-group attachment points or *AttachmentPoint* objects
- matching_mol: a class:*rdkit.Chem.Mol* molecule that contains the core matcher. This differs from the mol attribute as it will have attachment points removed

Constructor. Finds attachment points and constructs core matching molecule

Parameters

- **index** (int) – core index in multi-core decomposition
- **mol** (Mol) – a class:*rdkit.Chem.Mol* molecule that defined the core

fix_sidechain(*attachment_point*, *sidechain*)

Restores cyclicity in sidechains which have multiple mappings to the same attachment point. After Chem.ReplaceCore cycles at the R-group positions will be broken and replaced with multiple wildcard atoms

Parameters

- **attachment_point** (*AttachmentPoint*) – attachment point as *AttachmentPoint* object
- **sidechain** (Mol) – sidechain fragment as rdkit.Chem.Mol molecule

Return type
Mol

Returns
fixed structure

match_attachment_points_to_sidechain(*sidechain*)

Returns a list of attachment points if this side chain matches the core. If a match is found labels the wildcard atom in the sidechain with the R group number of the attachment point.

Typically there will only be one unique attachment point- though a cyclic sidechain may match 2 or more core attachment points and an attachment point may appear more than once

Parameters
sidechain (Mol) – sidechain fragment as class:*rdkit.Chem.Mol* molecule

Return type
List[*AttachmentPoint*]

Returns
Matching attachment points or None

match_sidechains(*mol*, *match*, *sidechains*)

Match all side chains in a decomposition to the r group positions in this core.

The list returned is the same length as self.attachment_points (the list of available R-group positions). If there is an side chain attached at that position the output list will contain an *Attachment* object, otherwise it will be None

Parameters
sidechains (List[Mol]) – a list of side chains as rdkit.Chem.Mol molecules

Return type
List[Optional[*Attachment*]]

Returns
A list of optional attachments

class ruse.rdkit.rgroup.**DecompositionSummary**(*molecule: Mol*, *core:* Core, *sidechains: List[*Attachment | *None]*, *mapping: Tuple[int, ...]*)

A class that is use to store molecular decompositions.

The list of side chains is the same length as core.attachment_points (the list of available R-group positions). If there is an side chain attached at that position the output list will contain an *Attachment* object, otherwise it will be None

Attributes:

- molecule: the molecule that is decomposed (as `rdkit.Chem.Mol`)
- core: the decomposition core (class *Core*)
- sidechains: A list of optional attachments
- mapping: mapping of core to molecule

Create new instance of DecompositionSummary(molecule, core, sidechains, mapping)

contains(*other*)

Parameters
other (*DecompositionSummary*) – another decomposition

Return type
bool

Returns
True if this decomposition is a superset of the other decomposition

core: *Core*

Alias for field number 1

mapping: Tuple[int, ...]

Alias for field number 3

molecule: Mol

Alias for field number 0

relabel_attachments()

Relabel attachment atoms with R group number- should only be done once all matching is completed

Return type
None

sidechains: List[Optional[*Attachment*]]

Alias for field number 2

to_molecule_list(*generate_coordinates=False*)

Converts this decomposition to a list of rdkit molecules. The list contains molecule, core and R groups. If an R group is not present in the molecule it will be None in the list

Parameters
generate_coordinates (bool) – set True to create 2D coordinates for depiction

Return type
List[Optional[Mol]]

Returns
the list of optional molecules

class ruse.rdkit.rgroup.**Rgroup**

A class to perform r group decomposition using new Rdkit methods.

to_molecule_grid(*generate_rgroup_coords=False*, *column_major=False*, *include_missing=True*)

Converts the decompositions to a grid of molecules

Each row of the grid contains molecule, core and R groups. If an R group is not present in the molecule it will be None in the list

Parameters

- **generate_rgroup_coords** – set True to generate 2D molecular co-ordinates for depiction
- **column_major** – set True to return the transpose of the grid

Return type
List[List[Optional[Mol]]]

Returns
the grid of molecules

class ruse.rdkit.rgroup.**RgroupDecomposer**

A class to perform multi-core decomposition of a set of molecules

Attributes:

- cores: a list of cores to decompose molecules
- molecules: the input molecules
- decomposition: the results of decomposition. This a list with one entry for each input molecule. If there are any decompositions for a molecule its entry will be a list of those decompositions (each of class *DecompositionSummary*), if there are no decompositions for a molecule the entry will be None
- match_first_core_only: set true if only the results for the first matching core should be included. By default, matches for all cores are included. If this is set and the first matching core has multiple matches the one will the largest number of R groups will be retained

Empty constructor

decompose(*core_smarts*, *structure_smiles*)

Performs decomposition

Parameters

- **core_smarts** (List[str]) – A list of core smarts patterns
- **structure_smiles** (List[str]) – A list of molecule smiles

Return type
None

decompose_molecules(*core_molecules*, *molecules*, *process_symmetric_groups=False*)

Performs decomposition

Parameters

- **core_molecules** (List[Mol]) – a list of query core molecules (each of rdkit.Chem.Mol)
- **molecules** (List[Mol]) – a list of target molecules to decompose (each of rdkit.Chem.Mol)

Return type
None

Returns

number_r_groups()

Returns the number of available R-groups in the decomposition

Returns
number of R-groups

to_molecule_grid(*generate_rgroup_coords=False*, *column_major=False*, *include_missing=False*)

Converts the decompositions to a grid of molecules

Each row of the grid contains molecule, core and R groups. If an R group is not present in the molecule it will be None in the list

Parameters

- **generate_rgroup_coords** – set True to generate 2D molecular co-ordinates for depiction
- **column_major** – set True to return the transpose of the grid

Return type
List[List[Optional[Mol]]]

Returns
the grid of molecules

ruse.rdkit.rgroup.**is_r_group_atom**(*atom*)

Returns true if this is an R group atom.

Parameters
atom (Atom) – An rdkit atom

Return type
Optional[int]

Returns
The rgroup number, or none if the atom does not define an rgroup

ruse.rdkit.rgroup.**mol_to_cores**(*mol_block*, *print_information=False*)

Converts an MDL mol block containing multiple query cores into a list of Rdkit query molecules

Parameters

- **mol_block** (str) –
- **print_information** (bool) – set True to print out summary information

Return type
List[Mol]

Returns
a list of core molecules of rdkit.Chem.Mol

CHAPTER

EIGHT

RUSE.UTIL MODULES

8.1 data_table.py

Copyright (C) 2017-2022 Glysade, LLC

The data table class

class ruse.util.data_table.**DataTable**(*columns=None*, *data=None*, *output_columns=None*)

A class for representing tabular data

Attributes:

- data: A 2D array of all data in the table. Each cell can have an float, int or string value- with columns having the same type. Array is row major- i.e. implemented as a list of rows
- columns: Column definition structure. The columns attribute is a list of dictionaries, with the following entries:
  - name: column name
  - dataType: data type of the column (e.g. float, int or string)
  - properties: Dictionary of properties, including ContentType which is the mime type of the column (e.g. chemical/x-sequence-type)
  - id: column id. Normally a generated UUID
- output_columns: a list of output columns for use in task chaining

Constructor. Call with no arguments to create an empty data table

Parameters

- **columns** (Optional[List[Dict[str, Union[str, Dict[str, Union[str, Dict[str, str]]]]]]]) – defines any columns in the data table
- **data** (Optional[List[List[Union[str, int, float]]]]) – a 2D array to set cell data
- **output_columns** – used to specify any output columns

__iadd__(*other*)

Adds/appends another data table to this one, using row axis. Columns are mapped from the other data table to this one using names and rows appended

Parameters
other – the data table to append to this one

Returns
this data table

__init__(*columns=None*, *data=None*, *output_columns=None*)

Constructor. Call with no arguments to create an empty data table

Parameters

- **columns** (Optional[List[Dict[str, Union[str, Dict[str, Union[str, Dict[str, str]]]]]]]) – defines any columns in the data table
- **data** (Optional[List[List[Union[str, int, float]]]]) – a 2D array to set cell data
- **output_columns** – used to specify any output columns

add_column(*base_name*, *data_type*, *props=None*, *content_type=None*, *fill_rows=True*, *data=None*)

Adds a column to the data table. Does not change the class data attribute unless fill_rows is True

Parameters

- **base_name** (str) – a base name for the column
- **data_type** (str) – the data type (float, integer or string) for the column
- **props** (Optional[Dict[str, str]]) – dictionary of column properties
- **content_type** (Optional[str]) – the content type or mime type for the column
- **fill_rows** (bool) – set True (the default) to resize data to include empty cells (value None) for the new column
- **data** (Optional[List[Union[str, int, float]]]) – optional data for the new column

Return type
int

Returns
the index of the new column

append_columns(*other*)

Appends the columns in another table to this one. Assumes each table has the same number of rows

Parameters
other – the other data table

Return type
None

classmethod column_definition(*name*, *data_type*, *content_type=None*, *properties=None*)

Creates a column definition for a column. This should only be used when creating a new table, otherwise column names may not be unique

Parameters

- **name** (str) – the column name
- **data_type** (str) – the column data type: one of float, int, binary or string
- **content_type** (Optional[str]) – the mime type or content type for the column
- **properties** (Optional[Dict[str, str]]) – Optional additional properties

Return type
Union[str, Dict[str, Union[str, Dict[str, str]]]]

Returns
column definition as dictionary

column_values(*column_no*)

Return all values for a column

Parameters
column_no (int) – the column index

Return type
List[Union[str, int, float]]

Returns

content_type(*column_idx*)

Returns the content type for the column at an index

Parameters
column_idx (int) – the column index

Returns
the content type for the column

get_column_index(*name*)

Return the index of the column with the given name, or -1 if no column is found

Parameters
name (str) – column name

Returns
index of column in data table

set_value(*value*, *row_idx*, *column_idx*)

Set a cell value

Parameters

- **value** (Union[str, int, float]) – cell value
- **column_idx** (int) – column index
- **row_idx** (int) – row index

Return type
None

to_data()

Convert to a raw data structure, suitable for encoding into JSON

Return type
Dict[str, Union[List[List[Union[str, int, float]]], List[Dict[str, Union[str, Dict[str, Union[str, Dict[str, str]]]]]], Dict[str, Dict[str, Dict[str, Union[str, int, float]]]]]]

Returns
Raw data representation

to_json()

Converts the datatype to a simple python data structure that can be converted to JSON

Return type
Dict[str, Union[List[List[Union[str, int, float]]], List[Dict[str, Union[str, Dict[str, Union[str, Dict[str, str]]]]]], Dict[str, Dict[str, Dict[str, Union[str, int, float]]]]]]

Returns
Json compatible data

unique_column_name(*name*)

Creates a unique name for a column, assuming that it is the next column added. If there is no column name in the data table that has the base column name, then the base name will be the column name. Otherwise, the column number will be appended to the base column name. If that name is in use then a integer will be found to append that creates a unique column name

Parameters
name (str) – base name for the column

Return type
str

Returns
unique name for the column

8.2 frozen.py

Prevent objects setting undeclared attributes

See http://code.activestate.com/recipes/252158-how-to-freeze-python-classes/

class ruse.util.frozen.**Frozen**

Subclasses of Frozen are frozen, i.e. it is impossible to add new attributes to them and their instances.

ruse.util.frozen.**frozen**(*set*)

Raise an error when trying to set an undeclared name, or when calling

Parameters
set – default attribute setter

Returns
new attribute setter

8.3 log.py

Copyright (C) 2017-2022 Glysade, LLC

Ruse logging module

class ruse.util.log.**Level**

Global logging levels

__weakref__

list of weak references to the object (if defined)

ruse.util.log.**_to_log**(*level*, *args*)

Writes log message

Parameters

- **level** – log level as str
- **args** – messages

ruse.util.log._to_string(*args*)

Convert a list of messages to a single string

Parameters
args – messages

Returns
joined string

ruse.util.log.**critical**(**args*)

Log an critical message

Parameters
args – messages

ruse.util.log.**debug**(**args*)

Log a debug message

Parameters
args – messages

ruse.util.log.**error**(**args*)

Log an error message

Parameters
args – messages

ruse.util.log.**info**(**args*)

Log an informational message

Parameters
args – messages

ruse.util.log.**set_level**(*level*)

Set the global log level

Parameters
level –

ruse.util.log.**warning**(**args*)

Log an warning message

Parameters
args – messages

8.4 util.py

Copyright (C) 2017-2022 Glysade, LLC

Helper functions for Ruse

exception ruse.util.util.**RuseServerException**

An exception class for ruse service errors

ruse.util.util.**dict_to_list**(*in_dict*)

Converts a dictionary to a list

Parameters
in_dict (Dict[TypeVar(T), TypeVar(T)]) – input dictionary

Return type
List[TypeVar(T)]

Returns
output list

ruse.util.util.**find_on_path**(*alternatives*)

Return the full path of the first of the input executables found on the PATH. Throws ValueError if none of the alternatives can be found

Parameters
alternatives (List[str]) –

Return type
str

Returns

ruse.util.util.**get_task_id**()

Gets the id of the current task from the current working directory :rtype: int :return: Current task id

ruse.util.util.**get_task_result**(*task_id*)

Retrieves result.json from the ruse server for a given task id

Parameters
task_id (int) – task id

Returns
output json for task

ruse.util.util.**getter**(*json_obj*, *key*)

Given a key extract value from a input Json python dictionary

Parameters

- **json_obj** – dictionary
- **key** (str) – the key to extract

Returns
value of key, or None if key is not present

ruse.util.util.**has_input_field**(*json_in*, *field*)

Parameters

- **json_in** – python data structure from submitted Json
- **field** (str) – input field name

Return type
bool

Returns
True if the input field is present in the Json

ruse.util.util.**input_field_type**(*json_in*, *field*)

Retrieve the contentType attribute for an input field from submitted json

Parameters

- **json_in** – python data structure from submitted Json
- **field** (str) – input field name

Return type
str

Returns
input field content type

ruse.util.util.**input_field_value**(*json_in*, *field*, *optional=False*)

Retrieve the data attribute for an input field from submitted json

Parameters

- **json_in** – python data structure from submitted Json
- **field** (str) – input field name
- **optional** (bool) – If true return None if the input field is not present

Returns
input field data

ruse.util.util.**is_exe**(*fpath*)

Parameters
fpath –

Returns
true if thid file is executable

ruse.util.util.**num_processors**()

Find the number of processors on the local machine

Return type
int

Returns
number of processors

ruse.util.util.**which**(*program*)

Finds the full path of an executable on the PATH environment (checking that it is, in fact, executable)

Parameters
program (str) – the executable to find

Return type
str

Returns
full path, or None if the executable is not on the PATH

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

r

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

S

T

U

V

W