Frootlab Shared Library

Release 0.9.1

Patrick Michl

Mar 26, 2019

Contents

CHAPTER 1

Introduction

The *Frootlab Shared Library* (flib) is a multi-purpose Python library, which primarily aims to support projects at Frootlab by a common base library. The majority of the comprised modules, however, is kept generic and well documented, to facilitate their application in other open source projects as well.

The Frootlab Shared Library was originally created in 2019 to allow a separation of the collaborative data science platform *Nemoa* into smaller and individually maintainable subprojects Pandora, Motley and Nemoa. For more information about these projects see their respective project pages.

CHAPTER 2

Installation

The *Frootlab Shared Library* (flib) requires Python 3.7 or later. If you do not already have a Python environment configured on your computer, please see the instructions for installing the full scientific Python stack.

Note: If you are using the Windows platform and want to install optional packages (e.g., *scipy*), then it may be useful to install a Python distribution such as: Anaconda, Enthought Canopy, Python(x,y), WinPython, or Pyzo. If you already use one of these Python distributions, please refer to their online documentation.

Below it is assumed, that you have the default Python environment configured on your computer and you intend to install the frootlab shared library inside of it. If you want to create and work with Python virtual environments, please follow instructions on venv and virtual environments.

2.1 Install the latest distributed package

You can install the latest distributed package of flib by using `pip`:

```
$ pip install flib
```

2.2 Install the development branch

The installation requires that you have Git installed on your system. Under this prerequisite the first step is to clone the github repository of flib:

```
$ git clone https://github.com/frootlab/flib.git
```

Thereupon the development branch can locally be installed by using *pip*:

```
$ cd flib
$ pip install -e .
```

The `pip install` command allows you to follow the development branch as it changes by creating links in the right places and installing the command line scripts to the appropriate locations.

2.3 Update the development branch

Once you have cloned the GitHub repository onto a local directory, you can update it anytime by running a `git pull` in this directory:

```
$ git pull
```

2.4 Testing the development branch

The Frootlab Shared Library uses the Python builtin module **:module:‘unittest‘** for testing. Since the tests are not included in the distributed package at you are required to install the development branch, as described above. Thereupon you have to switch to the repository directory and run:

```
$ python3 tests
```

CHAPTER 3

flib

3.1 flib package

3.1.1 Subpackages

flib.base package

Submodules

flib.base.abc module

Metaclasses and Abstract Base Classes for frequently used design patterns.

class Isolated
Bases: object

Abstract Base Class for per-instance isolated classes.

The Isolated base class is a helper class, which is included to allow instance checking against the IsolatedMeta metaclass.

class IsolatedMeta
Bases: abc.ABCMeta

Metaclass for isolated classes.

Isolated classes create a new subclass for any instance to allow the modification of class methods without side effects. Common use cases for isolated classes include Singletons, Multitons and built classes, that avoid generic programming in favor of higher efficiency or lower memory usage.

class Multiton
Bases: object

Abstract Base Class for Multiton Classes.

The Multiton base class is a helper class, which is included to allow instance checking against the MultitonMeta metaclass.

class MultitonMeta
Bases: *flib.base.abc.IsolatedMeta*

Metaclass for Multitons.

Multiton Classes only create a single instance per given arguments by using a new subclass for class isolation. This allows a controlled creation of multiple distinct objects, that are globally accessible and unique. Multiton classes may be regarded as a generalization of Singletons in the sense of 'Collections of Singletons'. Common use cases comprise application global configurations, caching and collections of constants (given as immutable objects).

class Proxy
Bases: abc.ABC

Abstract Base Class for Connect Proxies.

connect(**args*, ***kwds*) → None
Establish connection to source.

disconnect() → None
Close connection to source.

pull() → None
Pull state changes from source.

push() → None
Push state changes to source.

class Singleton
Bases: object

Abstract Base Class for Singletons.

The Singleton base class is a helper class, which is included to allow instance checking against the SingletonMeta metaclass.

class SingletonMeta
Bases: *flib.base.abc.IsolatedMeta*

Metaclass for Singletons.

Singleton classes only create a single instance per application and therefore by definition are special case of isolated classes. This creation pattern ensures the application global uniqueness and accessibility of instances, comparably to global variables. Common use cases comprise logging, sentinel objects and application global constants (given as immutable objects).

sentinel(*cls: flib.base.abc.SingletonMeta*) → object
Class decorator that creates a Sentinel from a Singleton class.

Parameters **cls** – Subclass of the class *Singleton*

Returns Instance of the given Singleton class, which adopts a class like behaviour, including the ability of instantion, hashing and its representation.

flib.base.attrib module

Attributes and Attribute Groups.

class Attribute(*fget: Union[Callable, str, None] = None, fset: Union[Callable, str, None] = None, fdel: Union[Callable, str, None] = None, doc: Optional[str] = None, dtype: Union[Type[Any], Tuple[Type[Any], ...], None] = None, readonly: bool = False, default: Any = None, factory: Union[Callable, str, None] = None, binddict: Optional[str] = None, bindkey: Optional[str] = None, remote: bool = False, inherit: bool = False, category: Optional[str] = None*)

Bases: `property`, *`flib.base.abc.Isolated`*

Extended data descriptor for Attributes.

Data descriptors are classes, used for binding attributes to fields and thereby provide an abstraction layer that facilitates encapsulation and modularity. When any class contains a data descriptor as a class attribute (i.e. a method), then the data descriptor class defines the accessor, mutator and manager methods of the respective attribute.

A succinct way of building data descriptors is given by the `property class`, which automatically creates an accessor, a mutator and a destructor method from it's passed arguments. The *Attribute* class extends this automation by additional options for managing and controlling the behaviour of the attribute:

- Declaration of accessor, mutator and destructor methods by *forward references*
- Automatic *type checking* against given data type(s)
- Restriction to *read-only* attributes
- Setting of *default values*, ether as a fixed value, a factory function or by inheritance from a parental object
- Binding to *arbitrary mappings*, to allow a namespace aggregation of the attributes data e.g. by their logical attribute type.
- Binding to *arbitrary keys*, e.g. to provide different accessors to identical attribute data
- Handling of *remote attributes* of a parent object, to allow sharing of attributes between different objects
- Locical aggregation of attributes by *categories*

Parameters

- **fget** – Accessor method of the attribute. If provided, it must be a callable or a string, that references a valid method of the owner instance.
- **fset** – Mutator method of the attribute. If provided, it must be a callable or a string, that references a valid method of the owner instance.
- **fdel** – Destructor method of the attribute. If provided, it must be a callable or a string, that references a valid method of the owner instance.
- **doc** – Docstring of the attribute, which is retained throughout the runtime of the application. For more information and doctring convention see **PEP 257**.
- **dtype** – Data type definition of the attribute given as a type or a tuple of types, which is used for type checking assignments to the attribute. If the value passed to the mutator method is not an instance of any of the given types, a *`InvalidTypeError`* is raised. By default type checking is disabled.
- **default** – Default value, which is returned by calling the getter method, if the following conditions are met: (1) The attribute is not a remote attribute of the parent, (2) the attribute is not inherited from the parent, (3) the attribute has not yet been set, (4) the attribute has no default factory.
- **factory** – Default method, which is called if a default value is required. If provided, it must be a callable or a string, that references a valid method of the owner instance. This method is called, if the following conditions are met: (1) The attribute is not a remote

attribute of the parent, (2) the attribute is not inherited from the parent, (3) the attribute has not yet been set.

- **binddict** – Name of the dictionary (or arbitrary mapping), which comprises the key, which is used to store the attribute data. If provided, it must be a string, that references an attribute of the owner instance, with type mapping. By default, the special attribute `__dict__` is used.
- **bindkey** – Name of key within the bound dictionary, which is used to store the attribute data. By default the name of the attribute is used.
- **remote** – Boolean value which determines, if the accessor, mutator and destructor methods are bypassed to the currently referenced parent attribute group. If no attribute group is referenced or the referenced attribute group, does not contain an attribute of the same name, a ReferenceError is raised on any request to the Attribute.
- **inherit** – Boolean value which determines if the default value is inherited from the (current) value of a referenced parent object. If no parent object is referenced or the referenced object, does not contain an attribute of the same name, then the default value is retrieved from the default factory, or if not given, from the default value. By default the default value is not inherited from the parent.
- **readonly** – Boolean value which determines, if the attribute is a read-only attribute. For read-only attributes the mutator method raises an AttributeError on requests to the mutator method. By default the attribute is read-writable.
- **category** – Optional name of category, which allows a logical aggregation of attributes. If given, that category has to be a string. By default not category is set.

class Content(**args*, ***kwds*)

Bases: *flib.base.attrib.Attribute*

Attributes for persistent content storage objects.

class Group(*parent: Optional[Group] = None*, *readonly: Optional[bool] = None*, *remote: Optional[bool] = None*, *inherit: Optional[bool] = None*, *content: Optional[Dict[str, Any]] = None*, *metadata: Optional[Dict[str, Any]] = None*)

Bases: *flib.base.abc.Isolated*

Class for Attribute Groups.

Attribute Groups are used to bind attributes (and other attribute groups) into tree structured objects with a hierarchical control interface. This includes a common interface to access and mutate the values of it's contained (sub)attributes as well as a common interface to superseed the settings of it's contained (sub)groups to control the group behaviour in different applications.

Parameters

- **parent** – Reference to logical parent :class:'attribute group <.attrib.Group>', which is used for inheritance and shared attributes. By default no parent is referenced. Note: The logical parent does not denote the attribute group, that contains this group, but an equally structured attribute, which is used to infere dynamical values for the attributes.
- **readonly** – Boolean value, which superseeds the contained attributes read-only behaviour. For the values True or False, all contained attributes respectively are read-only or read-writable. By the default value None, the attributes' settings are not superseeded.
- **remote** – Boolean value, which superseeds the contained attributes remote behaviour. For the value True, all contained attributes are shared attributes of the parent attribute group, which must be referenced and contain the respetive attributes, or an error is raised. For the

value False, all contained atributes are handled locally, regardless of a parent group. By the default value None, the attributes' settings are not superseeded.

- **inherit** – Boolean value, which superseeds the contained attributes inheritance behaviour. For the value True, all contained attributes inherit their default values from the attribute values of the parent attribute group, which must be referenced and contain the respetive attributes, or an error is raised. For the value False, the default values of the contained atributes are handled locally, regardless of a parent group. By the default value None, the attributes settings are not superseeded.
- **content** –
- **metadata** –

class MetaData(**args*, ***kwds*)
Bases: *flib.base.attrib.Attribute*

Attributes for persistent metadata storage objects.

class Temporary(**args*, ***kwds*)
Bases: *flib.base.attrib.Attribute*

Attributes for non persistent storage objects.

class Virtual(**args*, ***kwds*)
Bases: *flib.base.attrib.Attribute*

Attributes for non persistent virtual objects.

flib.base.binary module

Binary object functions.

as_bytes(*data: Union[bytes, bytearray, memoryview, str], encoding: Optional[str] = None*) → bytes
Convert bytes-like object or str to bytes.

Parameters **data** – Binary data given as bytes-like object or string

compress(*data: Union[bytes, bytearray, memoryview, str], level: int = -1*) → bytes
Compress binary data using the gzip standard.

Parameters

- **data** – Binary data given as bytes-like object or string.
- **level** – Compression level ranging from *-1* to *9*, where the default value of *-1* is a compromise between speed and compression. For level *0* the given binary data is deflated without attempted compression, *1* denotes the fastest compression with minimum compression capability and *9* the slowest compression with maximum compression capability.

Returns Binary data as bytes.

decode(*data: Union[bytes, bytearray, memoryview, str], encoding: Optional[str] = None, compressed: bool = False*) → bytes
Decode bytes-like object or str.

Parameters

- **data** – Binary data given as bytes-like object or string
- **encoding** – Encodings specified in **RFC 3548**. Allowed values are: *base16*, *base32*, *base64* and *base85* or None for no encoding. By default no encoding is used.

Returns Binary data as bytes.

decompress (*data: Union[bytes, bytearray, memoryview, str]*) → bytes
Decompress gzip compressed binary data.

Parameters **data** – Binary data given as bytes-like object or string.

Returns Binary data as bytes.

encode (*data: Union[bytes, bytearray, memoryview, str], encoding: Optional[str] = None*) → bytes
Encode bytes-like object or str.

Parameters

- **data** – Binary data given as bytes-like object or string
- **encoding** – Encodings specified in **RFC 3548**. Allowed values are: *base16*, *base32*, *base64* and *base85* or None for no encoding. By default no encoding is used.

Returns Binary data as bytes.

pack (*obj: object*, *encoding: Optional[str] = None*, *compression: Optional[int] = None*) → bytes
Compress and encode arbitrary object to bytes.

Parameters

- **obj** – Any object, that can be pickled
- **encoding** – Encodings specified in **RFC 3548**. Allowed values are: 'base16', 'base32', 'base64' and 'base85' or None for no encoding. By default no encoding is used.
- **compression** – Determines the compression level for `zlib.compress()`. By default no zlib compression is used. For an integer ranging from -1 to 9, a zlib compression with the respective compression level is used. Thereby *-1* is the default zlib compromise between speed and compression, *0* deflates the given binary data without attempted compression, *1* is the fastest compression with minimum compression capability and *9* is the slowest compression with maximum compression capability.

Returns Compressed and encoded byte representation of given object hierachy.

unpack (*data: Union[bytes, bytearray, memoryview, str], encoding: Optional[str] = None, compressed: bool = False*) → Any
Decompress and decode object from binary data.

Parameters

- **data** – Binary data given as bytes-like object or string.
- **encoding** – Encodings specified in **RFC 3548**. Allowed values are: 'base16', 'base32', 'base64' and 'base85' or None for no encoding. By default no encoding is used.
- **compressed** – Boolean value which determines, if the returned binary data shall be decompressed by using `zlib.decompress()`.

Returns Arbitry object, that can be pickled.

flib.base.call module

Collection of helper functions for callables.

parameters (*op: Callable*, **args*, ***kwds*) → collections.OrderedDict
Get parameters of a callable object.

Parameters

- **op** – Callable object

- ***args** – Arbitrary arguments, that are zipped into the returned parameter dictionary.
- ****kwds** – Arbitrary keyword arguments, that respectively - if declared within the callable object - are merged into the returned parameter dictionary. If the callable object allows a variable number of keyword arguments, all given keyword arguments are merged into the parameter dictionary.

Returns Ordered Dictionary containing all parameters.

Examples

```
>>> parameters(parameters)
OrderedDict()
>>> parameters(parameters, list)
OrderedDict([('operator', list)])
```

parse(*text: str*) → Tuple[str, tuple, dict]

Split a function call in the function name, it's arguments and keywords.

Parameters **text** – Function call given as valid Python code.

Returns A tuple consisting of the function name as string, the arguments as tuple and the keywords as dictionary.

safe_call(*f: Callable*, **args*, ***kwds*) → Any

Evaluate callable object for given parameters.

Evaluates a callable for the subset of given parameters, which is known to the callables signature.

Parameters

- **f** – Callable object
- ***args** – Arbitrary arguments
- ****kwds** – Arbitrary keyword arguments

Returns:

flib.base.catalog module

Organization and handling of algorithms.

class Card(*category: Type[flib.base.catalog.Category]*, *reference: Callable*, *data: Dict[str, Any]*)

Bases: object

Base Class for Catalog Cards.

class Category

Bases: object

Base class for Catalog Categories.

class Manager

Bases: *flib.base.abc.Singleton*

Catalog Manager.

add(*cat: type*, *kwds: dict*, *obj: Callable*) → None

add_category(*cat: type*) → None

get(*path: Union[str, Callable]*) → flib.base.catalog.Card

has_category(*cat: type*) → bool

search(*cat: Optional[type] = None*, *path: Optional[str] = None*, ***kwds*) → flib.base.catalog.Results

class Results

Bases: `list`

Class for search results.

get(*key: Hashable*) → list

association(*name: Optional[str] = None*, *tags: Optional[List[str]] = None*, *classes: Optional[List[str]] = None*, *plot: Optional[str] = 'Histogram'*, *directed: bool = True*, *signed: bool = True*, *normal: bool = False*, ***attr*) → Callable

Attribute decorator for association measure.

Parameters

- **name** – Name of the measure of association
- **tags** – List of strings, that describe the algorithm and allow it to be found by browsing or searching.
- **classes** – Optional list of model class names, that can be processed by the algorithm.
- **plot** – Name of plot class, which is used to interpret the results. Supported values are: None, 'Heatmap', 'Histogram', 'Scatter2D' or 'Graph'. Default: 'Heatmap'.
- **directed** – Boolean value which indicates if the measure of association is dictected. Default: True.
- **signed** – Boolean value which indicates if the measure of association is signed. Default: True.
- **normal** – Boolean value which indicates if the measure of association is normalized. Default: False.
- ****attr** – Supplementary user attributes, with the purpose to identify and characterize the algorithm by their respective values.

Returns Decorated function or method.

category(*cls: type*) → Type[flib.base.catalog.Category]

Decorate class as Catalog Category.

custom(*name: Optional[str] = None*, *category: Optional[str] = None*, *classes: Optional[List[str]] = None*, *tags: Optional[List[str]] = None*, *plot: Optional[str] = None*, ***attr*) → Callable

Attribute decorator for custom algorithms.

For the case, that an algorithm does not fit into the builtin categories ('objective', 'sampler', 'statistic', 'association') then a custom category is required, which does not prescribe the function arguments and return values.

Parameters

- **name** – Name of the algorithm
- **category** – Custom category name for the algorithm.
- **tags** – List of strings, that describe the algorithm and allow it to be found by browsing or searching
- **classes** – Optional list of model class names, that can be processed by the algorithm

- **plot** – Name of plot class, which is used to interpret the results. The default value None indicates, that that results can not be visalized. Supported values are: None, 'Heatmap', 'Histogram', 'Scatter2D' or 'Graph'
- ****attr** – Supplementary user attributes, with the purpose to identify and characterize the algorithm by their respective values.

Returns Decorated function or method.

objective(*name: Optional[str] = None*, *classes: Optional[List[str]] = None*, *tags: Optional[List[str]] = None*, *optimum: str = 'min'*, *scope: str = 'local'*, *plot: Optional[str] = None*, ***attr*) → Callable

Attribute decorator for objective functions.

Objective functions are scalar functions, thet specify the goal of an optimization problem. Thereby the objective function identifies local or global objectives by it's extremal points, which allows the application of approximations.

Parameters

- **name** – Name of the algorithm
- **tags** – List of strings, that describe the algorithm and allow it to be found by browsing or searching
- **classes** – Optional list of model class names, that can be processed by the algorithm
- **plot** – Name of plot class, which is used to interpret the results. The default value None indicates, that that results can not be visalized. Supported values are: None, 'Heatmap', 'Histogram', 'Scatter2D' or 'Graph'
- **scope** – Scope of optimizer: Ether 'local' or 'global'
- **optimum** – String describung the optimum of the objective functions. Supported values are 'min' and 'max'
- ****attr** – Supplementary user attributes, with the purpose to identify and characterize the algorithm by their respective values.

Returns Decorated function or method.

pick(*cat: Optional[type] = None*, *path: Optional[str] = None*, ***kwds*) → Callable

register(*cat: type*, ***kwds*) → Callable

Decorator to register classes and functions in the catalog.

sampler(*name: Optional[str] = None*, *classes: Optional[List[str]] = None*, *tags: Optional[List[str]] = None*, *plot: Optional[str] = 'Histogram'*, ***attr*) → Callable

Attribute decorator for statistical samplers.

Statistical samplers are random functions, that generate samples from a desired posterior distribution in Bayesian data analysis. Thereby the different approaches exploit properties of the underlying dependency structure. For more information see e.g. [1]

Parameters

- **name** – Name of the algorithm
- **tags** – List of strings, that describe the algorithm and allow it to be found by browsing or searching
- **classes** – Optional list of model class names, that can be processed by the algorithm
- **plot** – Name of plot class, which is used to interpret the results. Supported values are: None, 'Heatmap', 'Histogram', 'Scatter2D' or 'Graph'. The default value is 'Histogram'

- ****attr** – Supplementary user attributes, with the purpose to identify and characterize the algorithm by their respective values.

Returns Decorated function or method.

References

[1] https://en.wikipedia.org/wiki/Gibbs_sampling

search(*cat: Optional[type] = None*, *path: Optional[str] = None*, ***kwds*) → flib.base.catalog.Results

search_old(*module: Optional[module] = None*, ***kwds*) → dict
Search for algorithms, that pass given filters.

Parameters

- **module** – Module instance, which is used to recursively search in submodules for algorithms. Default: Use the module of the caller of this function.
- ****kwds** – Attributes, which are testet by using the filter rules

Returns Dictionary with function information.

statistic(*name: Optional[str] = None*, *classes: Optional[List[str]] = None*, *tags: Optional[List[str]] = None*, *plot: Optional[str] = 'Histogram'*, ***attr*) → Callable
Attribute decorator for sample statistics.

Sample statistics are measures of some attribute of the individual columns of a sample, e.g. the arithmetic mean values. For more information see [1]

Parameters

- **name** – Name of the algorithm
- **tags** – List of strings, that describe the algorithm and allow it to be found by browsing or searching
- **classes** – Optional list of model class names, that can be processed by the algorithm
- **plot** – Name of plot class, which is used to interpret the results. Supported values are: None, 'Heatmap', 'Histogram', 'Scatter2D' or 'Graph'. The default value is 'Histogram'
- ****attr** – Supplementary user attributes, with the purpose to identify and characterize the algorithm by their respective values.

Returns Decorated function or method.

References

[1] https://en.wikipedia.org/wiki/Statistic

flib.base.check module

Check types and values of objects.

has_attr(*obj: object*, *attr: str*) → None
Check if object has an attribute.

has_opt_type(*name: str, obj: object, hint: Generic[T]*) → None
Check type of optional object.

has_size (*name: str*, *obj: Sized*, *size: Optional[int] = None*, *min_size: Optional[int] = None*, *max_size: Optional[int] = None*) → None
Check the size of a sized object.

has_type (*name: str, obj: object, hint: Generic[T]*) → None
Check type of object.

is_callable (*name: str*, *obj: object*) → None
Check if object is callable.

is_class (*name: str*, *obj: object*) → None
Check if object is a class.

is_identifier (*name: str*, *string: str*) → None
Check if a string is a valid identifier.

is_negative (*name: str, obj: Union[int, float]*) → None
Check if number is negative.

is_not_negative (*name: str, obj: Union[int, float]*) → None
Check if number is not negative.

is_not_positive (*name: str, obj: Union[int, float]*) → None
Check if number is not positive.

is_positive (*name: str, obj: Union[int, float]*) → None
Check if number is positive.

is_subclass (*name: str, obj: object, ref: Type[Any]*) → None
Check if object is a subclass of given class.

is_subset (*a: str*, *seta: set*, *b: str*, *setb: set*) → None
Check if a set is a subset of another.

is_typehint (*name: str*, *obj: object*) → None
Check if object is a supported typeinfo object.

no_dublicates (*name: str, coll: Collection[T_co]*) → None
Check if all elements of a collection are unique.

not_empty (*name: str*, *obj: Sized*) → None
Check if a sized object is not empty.

flib.base.env module

Environmental integration.

basename (**args*, *pkgname: Optional[str] = None*) → str
Extract file basename from a path like structure.

Parameters ***args** – Path like arguments, respectively given by a tree of strings, which can be joined to a path.

Returns String containing basename of file.

Examples

```
>>> filename(('a', ('b', 'c')), 'base.ext')
'base'
```

clear_filename(*fname: str*) → str
Clear filename from invalid characters.

Parameters **fname** – Arbitrary string, which is be cleared from invalid filename characters.

Returns String containing valid path syntax.

Examples

```
>>> clear_filename('3/\nE{$5}.e')
'3E5.e'
```

copytree(*source: Union[str, os.PathLike, Sequence[Union[str, os.PathLike]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]]]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]]]]]]], target: Union[str, os.PathLike, Sequence[Union[str, os.PathLike]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]]]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]]]]]]], pkgname: Optional[str] = None*) → None
Copy directory structure from given source to target directory.

Parameters

- **source** – Path like structure, which comprises the path of a source folder
- **target** – Path like structure, which comprises the path of a destination folder

Returns True if the operation was successful.

expand(**args*, *udict: Optional[Dict[str, Any]] = None*, *pkgname: Optional[str] = None*, *envdirs: bool = True*) → pathlib.Path
Expand path variables.

Parameters

- ***args** – Path like arguments, respectively given by a tree of strings, which can be joined to a path.
- **udict** – dictionary for user variables. Thereby the keys in the dictionary are encapsulated by the symbol '*%*'. The user variables may also include references.
- **envdirs** – Boolen value which determines if environmental path variables are expanded. For a full list of valid environmental path variables see '.env.get_dirs'. Default is True

Returns String containing valid path syntax.

Examples

```
>>> expand('%var1%/c', 'd', udict = {'var1': 'a/%var2%', 'var2': 'b'})
'a\\b\\c\\d'
```

fileext(**args*, *pkgname: Optional[str] = None*) → str
Fileextension of file.

Parameters ***args** – Path like arguments, respectively given by a tree of strings, which can be joined to a path.

Returns String containing fileextension of file.

Examples

```
>>> fileext(('a', ('b', 'c')), 'base.ext')
'ext'
```

filename(**args*, *pkgname: Optional[str] = None*) → str
Extract file name from a path like structure.

Parameters ***args** – Path like arguments, respectively given by a tree of strings, which can be joined to a path.

Returns String containing normalized directory path of file.

Examples

```
>>> filename(('a', ('b', 'c')), 'base.ext')
'base.ext'
```

get_cwd() → pathlib.Path
Get path of current working directory.

Returns Path of current working directory.

get_dir(*dirname: str*, **args*, *pkgname: Optional[str] = None*, ***kwds*) → pathlib.Path
Get application specific environmental directory by name.

This function returns application specific system directories by platform independent names to allow platform independent storage for caching, logging, configuration and permanent data storage.

Parameters

- **dirname** – Environmental directory name. Allowed values are:

  user_cache_dir Cache directory of user

  user_config_dir Configuration directory of user

  user_data_dir Data directory of user

  user_log_dir Logging directory of user

  site_config_dir Site global configuration directory

  site_data_dir Site global data directory

  site_package_dir Site global package directory

  site_temp_dir Site global directory for temporary files

  package_dir Current package directory

  package_data_dir Current package data directory

- ***args** – Optional arguments that specify the application, as required by the function '.env.update_dirs'.
- ****kwds** – Optional keyword arguments that specify the application, as required by the function '.env.update_dirs'.

Returns String containing path of environmental directory or None if the pathname is not supported.

get_dirname(**args*, *pkgname: Optional[str] = None*) → str
Extract directory name from a path like structure.

Parameters ***args** – Path like arguments, respectively given by a tree of strings, which can be joined to a path.

Returns String containing normalized directory path of file.

Examples

```
>>> get_dirname(('a', ('b', 'c'), 'd'), 'base.ext')
'a\\b\\c\\d'
```

get_dirs(**args*, *pkgname: Optional[str] = None*, ***kwds*) → Dict[str, Any]
Get application specific environmental directories.

This function returns application specific system directories by platform independent names to allow platform independent storage for caching, logging, configuration and permanent data storage.

Parameters

- ***args** – Optional arguments that specify the application, as required by the function '.env.update_dirs'.
- ****kwds** – Optional keyword arguments that specify the application, as required by the function '.env.update_dirs'.

Returns Dictionary containing paths of application specific environmental directories.

get_encoding() → str
Get preferred encoding used for text data.

This is a wrapper function to the standard library function `locale.getpreferredencoding()`. This function returns the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess.

Returns String representing the preferred encoding used for text data.

get_home() → pathlib.Path
Get path of current users home directory.

Returns Path of current users home directory.

get_hostname() → str
Get hostname of the computer.

This is a wrapper function to the standard library function `platform.node()`. This function returns the computer's hostname. If the value cannot be determined, an empty string is returned.

Returns String representing the computer's hostname or None.

get_osname() → str
Get name of the Operating System.

This is a wrapper function to the standard library function `platform.system()`. This function returns the OS name, e.g. 'Linux', 'Windows', or 'Java'. If the value cannot be determined, an empty string is returned.

Returns String representing the OS name or None.

get_temp_dir() → pathlib.Path
Get path to temporary file within the package temp directory.

get_temp_file(*suffix: Optional[str] = None*) → pathlib.Path
Get path to temporary file within the package temp directory.

get_username() → str
Login name of the current user.

This is a wrapper function to the standard library function `getpass.getuser()`. This function checks the environment variables LOGNAME, USER, LNAME and USERNAME, in order, and returns the value of the first one which is set to a non-empty string. If none are set, the login name from the password database is returned on systems which support the pwd module, otherwise, an exception is raised.

Returns String representing the login name of the current user.

get_var(*varname: str*, **args*, *pkgname: Optional[str] = None*, ***kwds*) → Optional[str]
Get environment or application variable.

Environment variables comprise static and runtime properties of the operating system like 'username' or 'hostname'. Application variables in turn, are intended to describe the application distribution by authorship information, bibliographic information, status, formal conditions and notes or warnings. For mor information see **PEP 345**.

Parameters

- **varname** – Name of environment variable. Typical application variable names are: 'name': The name of the distribution 'version': A string containing the distribution's version number 'status': Development status of the distributed application.

  Typical values are 'Prototype', 'Development', or 'Production'

  'description': A longer description of the distribution that can run to several paragraphs.

  'keywords': A list of additional keywords to be used to assist searching for the distribution in a larger catalog.

  'url': A string containing the URL for the distribution's homepage.

  'license': Text indicating the license covering the distribution 'copyright': Notice of statutorily prescribed form that informs

  users of the distribution to published copyright ownership.

  'author': A string containing the author's name at a minimum; additional contact information may be provided.

  'email': A string containing the author's e-mail address. It can contain a name and e-mail address, as described in **RFC 822**.

  'maintainer': A string containing the maintainer's name at a minimum; additional contact information may be provided.

  'company': The company, which created or maintains the distribution. 'organization': The organization, twhich created or maintains the

  distribution.

  'credits': list with strings, acknowledging further contributors, Teams or supporting organizations.

- ***args** – Optional arguments that specify the application, as required by the function *update_vars()*.
- **pkgname** –

- ****kwds** – Optional keyword arguments that specify the application, as required by the function *update_vars()*.

Returns String representing the value of the application variable.

get_vars(**args*, *pkgname: Optional[str] = None*, ***kwds*) → Dict[str, Any]
Get dictionary with environment and application variables.

Environment variables comprise static and runtime properties of the operating system like 'username' or 'hostname'. Application variables in turn, are intended to describe the application distribution by authorship information, bibliographic information, status, formal conditions and notes or warnings. For mor information see **PEP 345**.

Parameters

- ***args** – Optional arguments that specify the application, as required by *update_vars()*.
- ****kwds** – Optional keyword arguments that specify the application, as required by *update_vars()*.

Returns Dictionary containing application variables.

is_dir(*path: Union[str, os.PathLike, Sequence[Union[str, os.PathLike]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]]]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]]]]]]], pkgname: Optional[str] = None*) → bool
Determine if given path points to a directory.

Extends `pathlib.Path.is_dir()` by nested paths and path variable expansion.

Parameters **path** – Path like structure, which is expandable to a valid path

Returns True if the path points to a regular file (or a symbolic link pointing to a regular file), False if it points to another kind of file.

is_file(*path: Union[str, os.PathLike, Sequence[Union[str, os.PathLike]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]]]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]]]]]]], pkgname: Optional[str] = None*) → bool
Determine if given path points to a file.

Extends `pathlib.Path.is_file()` by nested paths and path variable expansion.

Parameters **path** – Path like structure, which is expandable to a valid path.

Returns True if the path points to a directory (or a symbolic link pointing to a directory), False if it points to another kind of file.

join_path(**args*) → pathlib.Path
Join nested iterable path-like structure to single path object.

Parameters ***args** – Arguments containing nested iterable paths of strings and PathLike objects.

Returns Single Path comprising all arguments.

Examples

```
>>> join_path(('a', ('b', 'c')), 'd')
Path('a\\b\\c\\d')
```

match_paths (*paths: List[Union[str, os.PathLike]], pattern: str*) → List[Union[str, os.PathLike]]
Filter pathlist to matches with wildcard pattern.

Parameters

- **paths** – List of paths, which is filtered to matches with pattern.
- **pattern** – String pattern, containing Unix shell-style wildcards: '*': matches arbitrary strings '?': matches single characters [seq]: matches any character in seq [!seq]: matches any character not in seq

Returns Filtered list of paths.

Examples

```
>>> match_paths([Path('a.b'), Path('b.a')], '*.b')
[Path('a.b')]
```

mkdir (**args*, *pkgname: Optional[str] = None*) → bool
Create directory.

Parameters ***args** – Path like structure, which comprises the path of a new directory

Returns True if the directory already exists, or the operation was successful.

rmdir (**args*, *pkgname: Optional[str] = None*) → bool
Remove directory.

Parameters ***args** – Path like structure, which identifies the path of a directory

Returns True if the directory could be deleted

touch (*path: Union[str, os.PathLike, Sequence[Union[str, os.PathLike]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]]]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]], Sequence[Union[str, os.PathLike, Sequence[Union[str, os.PathLike]]]]]]], parents: bool = True, mode: int = 438, exist_ok: bool = True, pkgname: Optional[str] = None*) → bool
Create an empty file at the specified path.

Parameters

- **path** – Nested path-like object, which represents a valid filename in the directory structure of the operating system.
- **parents** – Boolean value, which determines if missing parents of the path are created as needed.
- **mode** – Integer value, which specifies the properties if the file. For more information see `os.chmod()`.
- **exist_ok** – Boolean value which determines, if the function returns False, if the file already exists.

Returns True if the file could be created, else False.

update_dirs (*appname: Optional[str] = None*, *appauthor: Union[str, bool, None] = None*, *version: Optional[str] = None*, *pkgname: Optional[str] = None*, ***kwds*) → None
Update application specific directories from name, author and version.

This function retrieves application specific directories from the package appdirs. Additionally the directory 'site_package_dir' is retrieved fom the standard library package distutils and 'package_dir' and 'package_data_dir' from the current top level module.

Parameters

- **appname** – is the name of application. If None, just the system directory is returned.
- **appauthor** – is the name of the appauthor or distributing body for this application. Typically it is the owning company name. You may pass False to disable it. Only applied in windows.
- **version** – is an optional version path element to append to the path. You might want to use this if you want multiple versions of your app to be able to run independently. If used, this would typically be "<major>.<minor>". Only applied when appname is present.
- ****kwds** – Optional directory name specific keyword arguments. For more information see appdirs.

update_vars (*filepath: Union[str, os.PathLike, None] = None*, *pkgname: Optional[str] = None*) → None
Update environment and application variables.

Environment variables comprise static and runtime properties of the operating system like 'username' or 'hostname'. Application variables in turn, are intended to describe the application distribution by authorship information, bibliographic information, status, formal conditions and notes or warnings. For mor information see **PEP 345**.

Parameters **filepath** – Valid filepath to python module, that contains the application variables as module attributes. By default the current top level module is used.

flib.base.literal module

Data type dependent string representation of objects.

as_datetime (*text: str*, *fmt: Optional[str] = None*) → datetime.datetime
Convert text to datetime.

Parameters

- **text** – String representation of datetime
- **fmt** – Optional string parameter, that specifies the format, which is used to decode the text to datetime. The default format is the [ISO 8601]_ format, given by the string *%Y-%m-%d %H:%M:%S.%f*.

Returns Value of the text as datetime.

as_dict (*text: str*, *delim: str = ', '*) → dict
Convert text into dictionary.

Parameters

- **text** – String representing a dictionary. Valid representations are: Python format: Allows keys and values of arbitrary types:

  Example: "{'a': 2, 1: True}"

  Delimiter separated expressions: Allow string keys and values: Example (Variant A): "<key> = <value><delim> . . . " Example (Variant B): "'<key>': <value><delim> . . . "

- **delim** – A string, which is used as delimiter for the separatation of the text. This parameter is only used in the DSV format.

Returns Value of the text as dictionary.

as_list (*text: str*, *delim: str = ', '*) → list
Convert text into list.

Parameters

- **text** – String representing a list. Valid representations are: Python format: Allows elements of arbitrary types:

  Example: "['a', 'b', 3]"

  Delimiter separated values (DSV): Allows string elements: Example: "a, b, c"

- **delim** – A string, which is used as delimiter for the separatation of the text. This parameter is only used in the DSV format.

Returns Value of the text as list.

as_path (*text: str*, *expand: bool = True*) → pathlib.Path
Convert text into list.

Parameters

- **text** – String representing a path.
- **expand** – Boolen value, whoch determines, if variables in environmental path variables are expanded.

Returns Value of the text as Path.

as_set (*text: str*, *delim: str = ', '*) → set
Convert text into set.

Parameters

- **text** – String representing a set. Valid representations are: Python format: Allows elements of arbitrary types:

  Example: "{'a', 'b', 3}"

  Delimiter separated values (DSV): Allows string elements: Example: "a, b, c"

- **delim** – A string, which is used as delimiter for the separatation of the text. This parameter is only used in the DSV format.

Returns Value of the text as set.

as_tuple (*text: str*, *delim: str = ', '*) → tuple
Convert text into tuple.

Parameters

- **text** – String representing a tuple. Valid representations are: Python format: Allows elements of arbitrary types:

  Example: "('a', 'b', 3)"

  Delimiter separated values (DSV): Allows string elements: Example: "a, b, c"

- **delim** – A string, which is used as delimiter for the separatation of the text. This parameter is only used in the DSV format.

Returns Value of the text as tuple.

decode (*text: str*, *target: Optional[type] = None*, *undef: Optional[str] = 'None'*, ***kwds*) → Any
Decode text representation of object to object.

Parameters

- **text** – String representing the value of a given type in it's respective syntax format. The standard format corresponds to the standard Python representation if available. Some types also accept further formats, which may use additional keywords.
- **target** – Target type, in which the text is to be converted.
- **undef** – Optional string, which respresents an undefined value. If undef is a string, then the any text, the matches the string is decoded as None, independent from the given target type.
- ****kwds** – Supplementary parameters, that specify the encoding format of the target type.

Returns Value of the text in given target format or None.

encode (*obj: object*, ***kwds*) → str
Encode object to literal text representation.

Parameters **obj** – Simple object

Returns Literal text representation of given object.

estimate (*text: str*) → Optional[type]
Estimate type of text by using `literal_eval()`.

Parameters **text** – String representation of python object.

Returns Type of text or None, if the type could not be determined.

from_str (*text: str*, *charset: Optional[str] = None*, *spacer: Optional[str] = None*) → str
Filter text to given character set.

Parameters

- **text** –
- **charset** – Name of used character set. Supportet options are:

  printable Printable characters

  UAX-31 ASCII identifier characters as defined in [UAX31]

Returns String, which is filtered to the chiven character set.

flib.base.mapping module

Helper functions for mappings.

crop (*d: Mapping[KT, VT_co]*, *prefix: str*, *trim: bool = True*) → dict
Crop mapping to keys, that start with an initial string.

Parameters

- **d** – Mapping that encodes sections by the prefix of string keys
- **prefix** – Key prefix as string
- **trim** – Determines if the section prefix is removed from the keys of the returned mapping. Default: True

Returns Subset of the original mapping, which only contains keys that start with the given section. Thereby the new keys are trimmed from the initial section string.

Examples

```
>>> crop({'a1': 1, 'a2': 2, 'b1': 3}, 'a')
{'1': 1, '2': 2}
```

flatten (*d: Dict[Any, Dict[Any, Dict[str, Any]]]*, *group: Optional[str] = None*) → Dict[Any, Dict[str, Any]]
Flatten grouped record dictionary by given group name.

Inverse dictionary operation to 'groupby'.

Parameters

- **d** – Nested dictionary, which entries are interpreted as attributes.
- **group** – Attribute names, which describes the groups.

Returns Dictinary which flattens the groups of the original dictinary to attributes.

Examples

```
>>> flatten({1: {'a': {}}, 2: {'b': {}}})
{'a': {}, 'b': {}}
>>> flatten({1: {'a': {}}, 2: {'b': {}}}, group='id')
{'a': {'id': 1}, 'b': {'id': 2}}
```

groupby (*d: Dict[Any, Dict[str, Any]]*, *key: str*, *rmkey: bool = False*) → Dict[Any, Dict[Any, Dict[str, Any]]]
Group record dictionary by the value of a given key.

Parameters

- **d** – Dictionary of dictionaries, which entries are interpreted as attributes.
- **key** – Name of attribute which is used to group the results by it's corresponding value.
- **rmkey** – Boolean which determines, if the group attribute is removed from the the sub dictionaries.

Returns Dictinary which groups the entries of the original dictinary in subdictionaries.

merge (**args*, *mode: int = 1*) → Mapping[KT, VT_co]
Recursively right merge mappings.

Parameters

- ***args** – Mappings with arbitrary structure
- **mode** – Creation mode for merged mapping: 0: change rightmost dictionary 1: create new dictionary by deepcopy 2: create new dictionary by chain mapping

Returns Dictionary containing right merge of dictionaries.

Examples

```
>>> merge({'a': 1}, {'a': 2, 'b': 2}, {'c': 3})
{'a': 1, 'b': 2, 'c': 3}
```

select (*d: Mapping[KT, VT_co], pattern: str*) → dict
Filter mappings to keys, that match a given pattern.

Parameters

- **d** – Mapping, which keys aregiven by strings
- **pattern** – Wildcard pattern as described in the standard library module `fnmatch`.

Returns Subset of the original mapping, which only contains keys, that match the given pattern.

Examples

```
>>> select({'a1': 1, 'a2': 2, 'b1': 3}, 'a*')
{'a1': 1, 'a2': 2}
```

strkeys (*d: dict*) → Dict[Union[str, Tuple[str, ...]], Any]
Recursively convert dictionary keys to string keys.

Parameters **d** – Hierarchivally structured dictionary with keys of arbitrary types.

Returns New dictionary with string converted keys. Thereby keys of type tuple are are not converted as a whole but with respect to the tokens in the tuple.

Examples

```
>>> strkeys({(1, 2): 3, None: {True: False}})
{('1', '2'): 3, 'None': {'True': False}}
```

sumjoin (**args*) → dict
Sum values of common keys in differnet dictionaries.

Parameters ***args** – dictionaries, that are recursively right merged

Returns New dictionary, where items with keys, that only occur in a single dictionary are adopted and items with keys, that occur in multiple dictionaries are united by a sum.

Examples

```
>>> sumjoin({'a': 1}, {'a': 2, 'b': 3})
{'a': 3, 'b': 3}
>>> sumjoin({1: 'a', 2: True}, {1: 'b', 2: True})
{1: 'ab', 2: 2}
```

flib.base.operator module

Classes and functions for functional programming.

class Getter (**args, domain: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None, target: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None*)
Bases: *flib.base.operator.Operator*

Class for Getters.

A getter essentially is the composition of a *fetch* operation, that specifies the fields of a given domain type and a subsequent *representation* of the fetched fields as an object of given target type, by using the target frame (if given) as field identifiers.

Parameters

- ***args** – Valid field identifiers within the domain type.
- **domain** – Optional domain like parameter, that specifies the type and (if required) the frame of the operator's domain. Supported domain types are `object`, subclasses of the `Mapping class` and subclasses of the `Sequence class`. If no domain type is specified (which is indicated by the default value None) the fields are identified by their argument positions of the operator.
- **target** – Optional domain like parameter, that specifies the type and (if required) the frame of the operator's target. Supported target types are `tuple`, `list` and `dict`. If no target is specified (which is indicated by the default value None) the target type depends on the arguments, that are passed to the operator. In this case for a single argument, the target type equals the type of the argument and for multiple argumnts, the target type is tuple.

class Identity(*domain: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None*)

Bases: *flib.base.operator.Operator*

Class for identity operators.

Parameters **domain** –

class Lambda(*expression: str = '', domain: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None, variables: Tuple[str, ...] = (), default: Optional[Callable[[...], Any]] = None, compile: bool = True*)

Bases: *flib.base.operator.Operator*

Class for operators, that are based on arithmetic expressions.

Parameters

- **expression** –
- **domain** – Optional domain category of the operator. If provided, the category has to be given as a `type`. Supported types are `object`, subclasses of the class:*Mapping class <collection.abs.Mapping>* and subclasses of the `Sequence class`. The default domain is object.
- **variables** – Tuple of variable names. This parameter is only required, if the domain category is a subclass of the `Sequence class`. In this case the variable names are used to map the fields (given as names) to their indices within the domain tuple.
- **default** –
- **compile** – Optional Boolean parameter, which determines if the operator is compiled after it is parsed.

variables

class Operator(**args, domain: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None, target: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None*)

Bases: `collections.abc.Callable`, *flib.base.abc.Multiton*

Abstract Base Class for operators.

Parameters

- ***args** –

- **domain** –
- **target** –

domain

target

class Vector(**args, domain: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None, target: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None, default: Optional[Callable[[...], Any]] = None*)

Bases: collections.abc.Sequence, *flib.base.operator.Operator*

Class for vectorial functions.

Parameters

- ***args** – Optional definitions of the function components. If provided, any component has to be given as a valid variable definition.
- **domain** – Optional domain like parameter, that specifies the type and (if required) the frame of the operator's domain. The accepted parameter values are documented in the class *Getter*.
- **target** – Optional domain like parameter, that specifies the type and (if required) the frame of the operator's target. The accepted parameter values are documented in the class *Getter*.
- **default** – Default operator which is used to map fields to field variables. By default the identity is used.

components

fields

class Zero(*target: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None*)

Bases: *flib.base.operator.Operator*

Class for zero operators.

A zero operator (or zero morphism) maps all given arguments to the zero object (empty object) of a given target category.

Parameters **target** – Optional target category of the operator. If provided, the target category must by given as a type, like int, float, str, set, tuple, list, dict or object. Then the returned operator maps all objects of any domain category to the zero object of the target category. By default the used zero object is None.

compose(**args, unpack: bool = False*) → Callable[[...], Any]

Compose operators.

Parameters ***args** – Operators, which shall be composed. If provided, any given operator is required to be a callable or None.

Returns Composition of all arguments, that do not evaluate to False. If all arguments evaluate to False, the identity operator is returned.

create_aggregator(**args, domain: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None, target: type = <class 'tuple'>*) → Callable[[Sequence[Any]], Any]

Creates an aggregation operator with specified variables.

Parameters

- ***args** – Optional variable definitions. If provided the operators given within the variable definitions are required to be valid aggregation functions

- **domain** – Optional domain category of the operator. If provided, the category has to be given as a `type`. Supported types are `object`, subclasses of the class:*Mapping class <collection.abs.Mapping>* and subclasses of the `Sequence class`. The default domain is object.
- **target** – Optional target category of the operator. If provided, the category has to be given as a type. Supported types are `tuple` and :dict:'dict'. If no target type is specified, the target category of the operator depends on the domain. In this case for the domain *object*, the target type is documented by the the builtin function `attrgetter()`, for other domains by the function `itemgetter()`.

Return:

create_group_aggregator(**args, key: Union[Hashable, Tuple[Hashable, ...], None] = None, domain: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None, target: type = <class 'tuple'>, presorted: bool = False*) → Callable[[Sequence[Any]], Any]

Creates a group aggregation operator.

Parameters

- ***args** – Optional variable definitions. If provided the operators given within the variable definitions are required to be valid aggregation functions. If not provided, the returned operator is the identity.
- **key** – Optional grouping key. If provided, the grouping key can be a field identifier or a composite key, given by a tuple of field identifiers. Thereby the type and the concrete meaning of the field identifiers depends on the domain of the operator.
- **domain** – Optional domain category of the operator. If provided, the category has to be given as a `type`. Supported types are `object`, subclasses of the class:*Mapping class <collection.abs.Mapping>* and subclasses of the `Sequence class`. The default domain is object.
- **target** – Optional target category of the operator. If provided, the category has to be given as a type. Supported types are `tuple` and :dict:'dict'. If no target type is specified, the target category of the operator depends on the domain. In this case for the domain *object*, the target type is documented by the the builtin function `attrgetter()`, for other domains by the function `itemgetter()`.
- **target** – Optional target type of the operator. Supported types are `tuple` and :dict:'dict'. If no target type is specified, the target of the operator depends on the domain. In this caser for the domain object, the target type is documented by the the builtin function `attrgetter()`, for other domains by the function `itemgetter()`.
- **presorted** – The operation splits in three consecutive steps, where in the first step the input sequence is sorted by the given keys. Consequently if the sequences are already sorted, the first step is not required and can be omitted to increase the performance of the operator. By default the input sequences are assumed not to be presorted.

Returns:

create_grouper(**args, domain: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None, presorted: bool = False*) → Callable[[Sequence[Any]], Any]

Create a grouping operator with fixed grouping keys.

Parameters

- ***args** – Optional *grouping keys*, which are used to group sequences of objects of given domain type. If provided, any grouping key is required to be a valid field identifier for the domain type.

- **domain** – Optional domain like parameter, that specifies the type and (if required) the frame of the operator's domain. The accepted parameter values are documented in the class *Getter*.
- **presorted** – The grouping operation splits in two consecutive steps: In the first step the input sequence is sorted by the given keys. Thereupon in the second step the sorted sequence is partitioned in blocks, which are equal with respect to the keys. Consequently if the sequences are already sorted, the first step is not required and can be omitted to increase the performance of the operator. By default the input sequences are assumed not to be presorted.

Returns List of sequences containing objects of a given domain typr, which are equal with respect to given grouping keys.

create_setter(**args, domain: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = <class 'object'>*) → Callable[[...], Any]

Create a setter operator.

Parameters

- ***args** – Optional pairs containing field values. If provided, the pairs have to be given in the format *(<field>, <value>)*, where *<field>* is a valid field identifier within the domain type and *<value>* an arbitrary object.
- **domain** – Optional domain like parameter, that specifies the type and (if required) the frame of the operator's domain. Supported domain types are `object`, subclasses of the class:*Mapping class <collection.abs.Mapping>* and subclasses of the `Sequence class`. If no domain type is specified (which is indicated by the default value None) the returned operator ir the zero operator.

create_sorter(**args, domain: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None, reverse: bool = False*) → Callable[[Sequence[Any]], Sequence[Any]]

Create a sorter with fixed sorting keys.

Sorters are operators, that act on sequences of objects of a given category and change the order of the objects within the sequence.

Parameters

- ***args** – Optional *sorting keys*, which in hierarchically descending order are used to sort sequences of objects of given domain type. If provided, any sorting key is required to be a valid field identifier for the domain type.
- **domain** – Optional domain like parameter, that specifies the type and (if required) the frame of the operator's domain. The accepted parameter values are documented in the class *Getter*.
- **reverse** – Optional boolean parameter. If set to True, then the sequence elements are sorted as if each comparison were reversed.

Returns Callable function which sorts a sequence of objects of a given domain by given sorting keys.

create_wrapper(***attrs*) → Callable[[...], Any]

Create a function wrapper that adds attributes.

Parameters ****attrs** – Arbitrary keyword arguments

Returns Function wrapper for given function, with additional specified attributes.

flib.base.otree module

Helper functions for objects and object trees.

call_attr(*obj: object*, *attr: str*, **args*, ***kwds*) → Any
Call an object attribute with given arguments.

Parameters

- **obj** – Arbitrary object
- **attr** – Name of callable object attribute
- ***args** – Arbitrary arguments, that are passed to the call
- ***kwds** – Arbitrary keyword arguments, that are passes to the call, if supported by the member attribute.

Returns Result of call.

get_lang_repr(*obj: object*, *separator: str = 'and'*) → str
Get enumerated representation of a collection's items.

Parameters **seperator** – String separator for collection items.

Returns Natural language representation of object.

get_members(*obj: object, pattern: Optional[str] = None, classinfo: Union[Type[Any], Tuple[Type[Any], ...]] = <class 'object'>, rules: Optional[Dict[str, Callable[[Any, Any], bool]]] = None, **kwds*) → list
List members of an object.

This is a wrapper function to *get_members_dict()*, but only returns the names of the members instead of the respective dictionary of attributes.

get_members_dict(*obj: object, pattern: Optional[str] = None, classinfo: Union[Type[Any], Tuple[Type[Any], ...]] = <class 'object'>, rules: Optional[Dict[str, Callable[[Any, Any], bool]]] = None, **kwds*) → dict
Get dictionary with an object's members dict attributes.

Parameters

- **obj** – Arbitrary object
- **pattern** – Only members which names satisfy the wildcard pattern given by 'pattern' are returned. The format of the wildcard pattern is described in the standard library module fnmatch. By default all names are allowed.
- **classinfo** – Classinfo given as a class, a type or a tuple containing classes, types or other tuples. Only members, which are ether an instance or a subclass of classinfo are returned. By default all types are allowed.
- **rules** – Dictionary with custom test functions, which are used for the comparison of the attribute value against the argument value. The dictionary items are of the form <*attribute*>: <*test*>, where <*attribute*> is the attribute name and <*test*> is a boolean valued lambda function, with two arguments <*arg*> and <*attr*>, which respectively give the value of the keyword argument and the member attribute. A member passes the rules, if all <*test*> functions evaluate to True against the given keyword arguments:

```
rules = {'tags': lambda arg, attr: set(arg) <= set(attr)}
```

By default any attribute, which is not in the filter rules is compared to the argument value by equality.

- ****kwds** – Keyword arguments, that define the attribute filter for the returned dictionary. For example if the argument "tags = ['test']" is given, then only members are returned, which have the attribute 'tags' and the value of the attribute equals ['test']. If, however, the filter rule of the above example is given, then any member, with attribute 'tags' and a corresponding tag list, that comprises 'test' is returned.

Returns Dictionary with fully qualified object names as keys and attribute dictinaries as values.

get_methods(*obj: object*, *pattern: Optional[str] = None*, *groupby: Optional[str] = None*, *key: Optional[str] = None*, *val: Optional[str] = None*) → Union[Dict[str, Any], Dict[Any, Dict[str, Any]], Dict[Any, Dict[Any, Dict[str, Any]]]]

Get methods from a given class instance.

Parameters

- **obj** – Class object
- **pattern** – Only methods, which names satisfy the wildcard pattern given by 'pattern' are returned. The format of the wildcard pattern is described in the standard library module `fnmatch`.
- **groupby** – Name of attribute which value is used to group the results. If groupby is None, then the results are not grouped. Default: None
- **key** – Name of the attribute which is used as the key for the returned dictionary. If key is None, then the method names are used as key. Default: None
- **val** – Name of attribute which is used as the value for the returned dictionary. If val is None, then all attributes of the respective methods are returned. Default: None

Returns Dictionary containing all methods of a given class instance, which names satisfy a given filter pattern.

get_name(*obj: object*) → str

Get name identifier for an object.

This function returns the name identifier of an object. If the object does not have a name attribute, the name is retrieved from the object class.

Parameters **obj** – Arbitrary object

Returns Name of an object.

get_summary(*obj: object*) → str

Get summary line for an object.

This function returns the summary line of the documentation string for an object as specified in **PEP 257**. If the documentation string is not provided the summary line is retrieved from the inheritance hierarchy.

Parameters **obj** – Arbitrary object

Returns Summary line for an object.

has_base(*obj: object, base: Union[type, str]*) → bool

Return true if the object has the given base class.

Parameters

- **obj** – Arbitrary object
- **base** – Class name of base class

Returns True if the given object has the named base as base

flib.base.parser module

Multi-Purpose Expression Parser.

class Expression(*tokens: List[flib.base.parser.Token], vocabulary: flib.base.parser.Vocabulary, mapping: Optional[dict] = None*)

Bases: object

as_func(*compile: bool = True*) → Callable

as_string(*translate: Optional[dict] = None*) → str

eval(**args*, ***kwds*) → Any

origin

simplify(*values: Optional[dict] = None*) → flib.base.parser.Expression

subst(*key: str, expr: Union[Expression, str]*) → flib.base.parser.Expression

Substitute variable in expression.

symbols

variables

class Parser(*vocabulary: Optional[flib.base.parser.Vocabulary] = None*)

Bases: object

eval(*expression: str*, **args*, ***kwds*) → Any

expression

parse(*expression: str*, *variables: Optional[Tuple[str, ...]] = None*) → flib.base.parser.Expression

success

class PyBuiltin

Bases: *flib.base.parser.PyOperators*

Python3 Operators and Builtins.

class PyExprEval

Bases: *flib.base.parser.Vocabulary*

Symbols used by py-expression-eval.

class PyOperators

Bases: *flib.base.parser.Vocabulary*

Python3 Operators.

This vocabulary is based on https://docs.python.org/3/reference/expressions.html

---

Hint: In difference to standard Python interpreter, some expressions are not valid: Invalid: x + -y -> Valid: x + (-y)

---

class Symbol(*type: int*, *key: str*, *value: Any*, *priority: int = 0*, *builtin: bool = False*, *factory: bool = False*)

Bases: object

Data Class for Parser Symbols.

builtin = False

factory = False

priority = 0

class Token(*type: int*, *id: Union[str*, *int] = 0*, *priority: int = 0*, *value: Any = 0*, *key: str = ''*)
Bases: object

id = 0

key = ''

priority = 0

value = 0

class Vocabulary
Bases: set

Base Class for Parser Vocabularies.

get(*type: int*, *key: str*) → flib.base.parser.Symbol
Get symbol from vocabulary.

search(*type: Optional[int] = None*, *builtin: Optional[bool] = None*) → Dict[str, flib.base.parser.Symbol]
Search for symbols within the vocabulary.

Parameters

- **type** – Integer parameter representing the type of symbols.
- **builtin** – Optional Boolean parameter representing the 'builtin' flag of the symbols. For 'True', only symbols are returned, that are marked to be builtin symbols, for 'False' only symbols, that are marked not to be builtin. By default the 'builtin' flag is ignored in the search result.

Returns OrderedDict containing Symbols in reverse lexical order to prioritize symbols with greater lenght.

parse(*expression: str*, *variables: Optional[Tuple[str*, *...]] = None*, *vocabulary: Optional[flib.base.parser.Vocabulary] = None*) → flib.base.parser.Expression

flib.base.phonetic module

Phonetic Algorithms.

soundex(*string: str*) → str
Calculate Soundex Index.

Soundex is a phonetic algorithm for indexing names by sound, as pronounced in English. The goal is for homophones to be encoded to the same representation so that they can be matched despite minor differences in spelling.

flib.base.pkg module

Package tree helper functions.

call_attr(*name: str*, **args*, ***kwds*) → Any
Call an attribute of current module with given arguments.

Parameters

- **name** – Name of callable attribute

- ***args** – Arbitrary arguments, that are passed to the call
- ***kwds** – Arbitrary keyword arguments, that are passes to the call, if supported by the member attribute.

Returns Result of call.

crop_functions(*prefix: str*, *module: Optional[module] = None*) → list
Get list of cropped function names that satisfy a given prefix.

Parameters

- **prefix** – String conatining the initial prefix of the returned functions
- **module** – Module reference. By default the current callers module is used.

Returns List of functions, that match a given prefix.

get_attr(*name: str*, *default: Any = None*, *module: Optional[module] = None*) → Any
Get an attribute of current module.

Parameters

- **name** – Name of attribute.
- **default** – Default value, which is returned, if the attribute does not exist.
- **module** – Optional reference to module, which is used to search for the given attribute. By default the current callers module is used.

Returns Value of attribute.

get_module(*name: Optional[str] = None*, *errors: bool = True*) → Optional[module]
Get reference to a module instance.

Parameters

- **name** – Optional name of module- If provided, the name is required to be a fully qualified name. By default a refrence to the module of the current caller is returned.
- **errors** – Boolean value which determines if an error is raised, if the module could not be found. By default errors are raised.

Returns Module reference or None, if the name does not point to a valid module.

get_parent(*module: Optional[module] = None*) → module
Get parent module.

Parameters **module** – Optional reference to module. By default the current callers module is used.

Returns Module reference to the parent module of the current callers module.

get_root(*module: Optional[module] = None*) → module
Get top level module.

Parameters **module** – Optional reference to module. By default the current callers module is used.

Returns Module reference to the top level module of the current callers module.

get_root_name(*fqn: str*) → str

get_submodule(*name: str*, *parent: Optional[module] = None*) → Optional[module]
Get instance from the name of a submodule of the current module.

Parameters

- **name** – Name of submodule of given module.

- **parent** – Optional reference to module, which has to be searched for submodules. By default the current callers module is used.

Returns Module reference of submodule or None, if the current module does not contain the given module name.

get_submodules(*parent: Optional[module] = None*, *recursive: bool = False*) → List[str]
Get list with submodule names.

Parameters

- **parent** – Optional reference to module, which has to be searched for submodules. By default the current callers module is used.
- **recursive** – Boolean value which determines, if the search is performed recursively within all submodules. By default the returned list only comprises immediate submodules.

Returns List with fully qualified names of submodules.

has_attr(*name: str*, *module: Optional[module] = None*) → bool
Determine if a module has an attribute of given name.

Parameters

- **name** – Name of attribute
- **module** – Optional reference to module, which is used to search for the given attribute. By default the current callers module is used.

Returns Result of call.

search(*module: Optional[module] = None, pattern: Optional[str] = None, classinfo: Union[Type[Any], Tuple[Type[Any], ...]] = <class 'function'>, key: Optional[str] = None, val: Optional[str] = None, groupby: Optional[str] = None, recursive: bool = True, rules: Optional[Dict[str, Callable[[Any, Any], bool]]] = None, errors: bool = False, **kwds*) → dict
Recursively search for objects within submodules.

Parameters

- **module** – Optional reference to module, which is used to search objects. By default the current callers module is used.
- **pattern** – Only objects which names satisfy the wildcard pattern given by 'pattern' are returned. The format of the wildcard pattern is described in the standard library module `fnmatch`. If pattern is None, then all objects are returned. Default: None
- **classinfo** – Classinfo given as a class, a type or a tuple containing classes, types or other tuples. Only members, which are ether an instance or a subclass of classinfo are returned. By default all types are allowed.
- **key** – Name of function attribute which is used as the key for the returned dictionary. If 'key' is None, then the fully qualified function names are used as keys. Default: None
- **val** – Name of function attribute which is used as the value for the returned dictionary. If 'val' is None, then all attributes of the respective objects are returned. Default: None
- **groupby** – Name of function attribute which is used to group the results. If 'groupby' is None, then the results are not grouped. Default: None
- **recursive** – Boolean value which determines if the search is performed recursively within all submodules. Default: True
- **rules** – Dictionary with individual filter rules, used by the attribute filter. The form is {<attribute>: <lambda>, ...}, where: <attribute> is a string with the attribute name and

<lambda> is a boolean valued lambda function, which specifies the comparison of the attribute value against the argument value. Example: {'tags': lambda arg, attr: set(arg) <= set(attr)} By default any attribute, which is not in the filter rules is compared to the argument value by equality.

- **errors** – Boolean value which determines if an error is raised, if the module could not be found. By default errors are not raised.
- ****kwds** – Keyword arguments, that define the attribute filter for the returned dictionary. For example if the argument "tags = ['test']" is given, then only objects are returned, which have the attribute 'tags' and the value of the attribute equals ['test']. If, however, the filter rule of the above example is given, then any function, with attribute 'tags' and a corresponding tag list, that comprises 'test' is returned.

Returns Dictionary with function information as specified in the arguments 'key' and 'val'.

flib.base.stack module

Call stack helper functions.

get_caller_module() → module
Get reference to callers module.

get_caller_module_name(*frame: int = 0*) → str
Get name of module, which calls this function.

Parameters **frame** – Frame index relative to the current frame in the callstack, which is identified with 0. Negative values consecutively identify previous modules within the callstack. Default: 0

Returns String with name of module.

get_caller_name(*frame: int = 0*) → str
Get name of the callable, which calls this function.

Parameters **frame** – Frame index relative to the current frame in the callstack, which is identified with 0. Negative values consecutively identify previous modules within the callstack. Default: 0

Returns String with name of the caller.

flib.base.stype module

Structural / Symbolic Types.

class Domain
Bases: tuple

Class for Domain Parameters.

basis
Alias for field number 2

frame
Alias for field number 1

type
Alias for field number 0

class Field
Bases: tuple

Class for Field Parameters.

id
Alias for field number 0

type
Alias for field number 1

class Variable
Bases: tuple

Class for the storage of variable definitions.

frame
Alias for field number 2

name
Alias for field number 0

operator
Alias for field number 1

create_basis(*arg: Any*) → Tuple[Tuple[Hashable, ...], Dict[Hashable, flib.base.stype.Field]]
Create domain frame and basis from given field definitions.

Args:

Returns:

create_domain(*domain: Union[type, None, Tuple[Optional[type], Tuple[Hashable, ...]], Domain] = None, defaults: Optional[Mapping[str, Any]] = None*) → flib.base.stype.Domain
Create Domain object from domain definition.

Parameters

- **domain** – Optional domain like parameter, that specifies the type and (if required) the frame of a domain.
- **defaults** – Optional mapping which is used to complete the given domain definition. The key *'type'* specifies the default domain type and is required to be given as a type. The key *'fields'* specifies a default ordered basis for the domain and is required to be given as a single or a tuple of field definitions.

Returns Instance of the class *Domain*

create_field
Create a Field object.

Parameters **field** – Field definition

Returns Instance of class *Field*

create_variable(*var: Union[str, Tuple[str], Tuple[str, Hashable], Tuple[str, Tuple[Hashable, ...]], Tuple[str, Callable[[...], Any]], Tuple[str, Callable[[...], Any], Hashable], Tuple[str, Callable[[...], Any], Tuple[Hashable, ...]], Tuple[str, str, Hashable], Tuple[str, str, Tuple[Hashable, ...]]], default: Optional[Callable[[...], Any]] = None*) → flib.base.stype.Variable
Create variable from variable definition.

Parameters

- **var** – Variable defintion

- **default** –

Returns:

flib.base.test module

Unittests.

class Case
Bases: tuple

Class for the storage of Case parameters.

args
Alias for field number 0

kwds
Alias for field number 1

value
Alias for field number 2

class GenericTest(*methodName='runTest'*)
Bases: unittest.case.TestCase

Custom testcase.

assertAllEqual(*a: object*, *b: object*) → None
Assert that two objects are equal.

assertCaseContain(*func: Callable[[...], Any]*, *cases: List[Case]*) → None
Assert that all function evaluations comprise the given values.

assertCaseEqual(*func: Callable[[...], Any]*, *cases: List[Case]*) → None
Assert that all function evaluations equal the given values.

assertCaseFalse(*func: Callable[[...], Any]*, *cases: List[Case]*) → None
Assert that all function evaluations cast to False.

assertCaseIn(*func: Callable[[...], Any]*, *cases: List[Case]*) → None
Assert that all function evaluations are in the given values.

assertCaseIsSubclass(*func: Callable[[...], Any]*, *supercls: type*, *cases: List[Case]*) → None
Assert outcome type of a class constructor.

assertCaseNotContain(*func: Callable[[...], Any]*, *cases: List[Case]*) → None
Assert that all function evaluations comprise the given values.

assertCaseNotEqual(*func: Callable[[...], Any]*, *cases: List[Case]*) → None
Assert that all function evaluations differ from the given values.

assertCaseNotIn(*func: Callable[[...], Any]*, *cases: List[Case]*) → None
Assert that all function evaluations are in the given values.

assertCaseNotRaises(*cls: Type[BaseException]*, *func: Callable[[...], Any]*, *cases: List[Case]*) → None
Assert that no function parameter raises an exception.

assertCaseRaises(*cls: Type[BaseException]*, *func: Callable[[...], Any]*, *cases: List[Case]*) → None
Assert that all function parameters raise an exception.

assertCaseTrue(*func: Callable[[...], Any]*, *cases: List[Case]*) → None
Assert that all function evaluations cast to True.

assertExactEqual(*a: object*, *b: object*) → None
Assert that two objects are equal in type and value.

assertIsSubclass(*cls: type*, *supercls: type*) → None
Assert that a class is a subclass of another.

assertNotRaises(*cls: Type[BaseException], func: Callable[[...], Any], *args, **kwds*) → None
Assert that an exception is not raised.

class ModuleTest(*methodName='runTest'*)
Bases: *flib.base.test.GenericTest*

Custom testcase.

assertModuleIsComplete() → None
Assert that all members of module are tested.

test_completeness() → None

flib.base.thread module

Multithreading functions.

create(*func: Callable[[...], Any], *args, **kwds*) → object
Create and start thread for given callable and arguments.

Module contents

Various Elementary Modules.

flib.errors package

Module contents

Errors and Exceptions.

exception ColumnLookupError(*colname: int*)
Bases: *flib.errors.TableError*, LookupError

Column Lookup Error.

exception ConnectError(*msg: str*)
Bases: *flib.errors.ProxyError*

Raise when a proxy connection can not be established.

exception DirNotEmptyError(*msg: str*)
Bases: *flib.errors.UserAssert*, OSError

Raise on remove requests on non-empty directories.

exception DisconnectError(*msg: str*)
Bases: *flib.errors.ProxyError*

Raise when a proxy connection can not be closed.

exception DublicateError(*name: str*, *dubl: set*)
Bases: *flib.errors.UserAssert*, ValueError

Raise when a collection contains dublicates.

exception ExistsError(*msg: str*)
Bases: *flib.errors.UserAssert*, LookupError

Raise when an already existing unique object shall be created.

exception FileFormatError(*name: str*, *fmt: str*)
Bases: *flib.errors.UserAssert*, OSError

Raise when a referenced file has an invalid file format.

exception FileNotGivenError(*msg: str*)
Bases: *flib.errors.UserAssert*, OSError

Raise when a file or directory is required, but not given.

exception FoundError(*msg: str*)
Bases: *flib.errors.UserAssert*, LookupError

Raise when an already registered unique object shall be registered.

exception InvalidAttrError(*obj: object*, *attr: str*)
Bases: *flib.errors.UserAssert*, AttributeError

Raise when a not existing attribute is called.

exception InvalidClassError(*name: str*, *obj: object*, *cls: type*)
Bases: *flib.errors.UserAssert*, TypeError

Raise when an object is required to be of a given subclass.

exception InvalidFormatError(*name: str*, *val: str*, *fmt: str*)
Bases: *flib.errors.UserAssert*, ValueError

Rasise when a string has an invalid format.

exception InvalidTypeError(*name: str*, *obj: object*, *info: object = None*)
Bases: *flib.errors.UserAssert*, TypeError

Raise when an object is required to be of a given type.

exception IsNegativeError(*name: str*, *val: numbers.Number*)
Bases: *flib.errors.UserAssert*, ValueError

Raise when a value may not be negative.

exception IsPositiveError(*name: str*, *val: numbers.Number*)
Bases: *flib.errors.UserAssert*, ValueError

Raise when a value may not be positive.

exception ItemNotFoundError(*name: str*, *val: Any*, *container: str*)
Bases: *flib.errors.UserAssert*, ValueError

Raise when an item is not found within a container.

exception MaxSizeError(*name: str*, *obj: collections.abc.Sized*, *max_len: int*)
Bases: *flib.errors.UserAssert*, ValueError

Raise when a container has too many elements.

exception MinSizeError(*name: str*, *obj: collections.abc.Sized*, *min_len: int*)
Bases: *flib.errors.UserAssert*, ValueError

Raise when a sized object has too few elements.

exception MissingKwError(*argname: str*, *obj: Callable*)
Bases: *flib.errors.UserAssert*, TypeError

Raise when a required keyword argument is not given.

exception NoSubsetError(*a: str*, *seta: set*, *b: str*, *setb: set*)
Bases: *flib.errors.UserAssert*, ValueError

Raise when sequence elements are not contained within another.

exception NotCallableError(*name: str*, *obj: object*)
Bases: *flib.errors.UserAssert*, TypeError

Raise when an object is required to be callable.

exception NotClassError(*name: str*, *obj: object*)
Bases: *flib.errors.UserAssert*, TypeError

Raise when an object is required to be a class.

exception NotExistsError(*msg: str*)
Bases: *flib.errors.UserAssert*, LookupError

Raise when a non existing unique object is requested.

exception NotFoundError(*msg: str*)
Bases: *flib.errors.UserAssert*, LookupError

Raise when a unique object is not found in a registry.

exception NotNegativeError(*name: str*, *val: numbers.Number*)
Bases: *flib.errors.UserAssert*, ValueError

Raise when a value must be negative.

exception NotPositiveError(*name: str*, *val: numbers.Number*)
Bases: *flib.errors.UserAssert*, ValueError

Raise when a value must be positive.

exception ProxyError(*msg: str*)
Bases: *flib.errors.UserError*

Base Exception for Proxy Errors.

exception PullError(*msg: str*)
Bases: *flib.errors.ProxyError*

Raise when a pull-request could not be finished.

exception PushError(*msg: str*)
Bases: *flib.errors.ProxyError*

Raise when a push-request could not be finished.

exception ReadOnlyAttrError(*obj: object*, *attr: str*)
Bases: *flib.errors.UserAssert*, AttributeError

Raise when a read-only attribute's setter method is called.

exception RowLookupError(*rowid: int*)
Bases: *flib.errors.TableError*, LookupError

Row Lookup Error.

exception SizeError(*name: str*, *obj: collections.abc.Sized*, *size: int*)
Bases: *flib.errors.UserAssert*, ValueError

Raise when a sized object has an invalid size.

exception TableError(*msg: str*)
Bases: *flib.errors.UserError*

Base Exception for Table Errors.

exception UserAssert(*msg: str*)
Bases: *flib.errors.UserError*, AssertionError

Exception for user asserts.

exception UserError(*msg: str*)
Bases: *flib.errors.UserException*

Exception for user errors.

exception UserException(*msg: str*)
Bases: Exception

Base class for user exceptions.

flib.io package

Submodules

flib.io.abc module

Abstract Base Classes for file I/O.

class Connector
Bases: abc.ABC

File Connector Base Class.

name

open(**args*, ***kwds*) → IO[Any]

flib.io.csv module

File I/O for text files containing delimiter-separated values.

The delimiter-separated values format is a family of file formats, used for the storage of tabular data. In it's most common variant, the comma-separated values format, the format was used many years prior to attempts to it's standardization in **RFC 4180**, such that subtle differences often exist in the data produced and consumed by different applications. This circumstance has and basically been addressed by **PEP 305** and the standard library module csv. The current module extends the capabilities of the standard library by I/O handling of *file references*, support of non-standard CSV headers, as used in CSV exports of the R programming language, automation in CSV parameter detection and row names.

class File (*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], header: Optional[Iterable[str]] = None, comment: Optional[str] = None, dialect: Union[str, csv.Dialect, None] = None, delimiter: Optional[str] = None, namecol: Optional[str] = None, hformat: Optional[int] = None*)

Bases: *flib.base.attrib.Group*

File Class for text files containing delimiter-separated values.

Parameters

- **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, an instance of the class *Connector* or an opened file object in reading or writing mode.
- **header** – Optional list (or arbitrary iterable) of strings, that specify the column names within the CSV file. For an existing file, the header by default is extracted from the first content line (not blank and not starting with #). For a new file the header is required and an error is raised if the header is not given.
- **comment** – Optional string, which precedes the header and the rows of the CSV file, e.g. to include metadata within the file. For an existing file, the string by default is extracted from the initial comment lines (starting with #). For a new file the comment by default is empty.
- **dialect** – Optional parameter, that indicates the used CSV dialect. The parameter can be given as a dialect name from the list returned by the function `csv.list_dialects()`, or an instance of the class `csv.Dialect`.
- **delimiter** – Single character, which is used to separetate the column values within the CSV file. For an existing file, the delimiter by default is detected from it's appearance within the file. For a new file the default value is ,.
- **namecol** – Optional column name of a column, which contains row names. If a valid column is given, then the readonly attribute *rownames* returns this column as a list. By default the column is infered from the used header format. For a **RFC 4180** compliant header by default no row names are used. For a header as used in exports of the R programming language by default the first column is used to store row names.
- **hformat** – Used CSV Header format. The following formats are supported: 0: **RFC 4180**:

  The column header represents the structure of the rows.

  1: R programming language: The column header does not include the first column of the rows. This follows by the convention, that in the R programming language the CSV export adds an extra column with row names as the first column, which is omitted within the CSV header.

close() → None

Close all opened file handlers of CSV File.

comment

String containing the initial '#' lines of the CSV file or an empty string, if no initial comment lines could be detected.

delimiter

Delimiter character of the CSV file or None, if for an existing file the delimiter could not be detected.

dialect = None

fields = None

header

List of strings containing column names from first non comment, non empty line of CSV file.

hformat

RFC 4180: The column header represents the structure of the rows.

1: R programming language: The column header does not include the first column of the rows. This follows by the convention, that in the R programming language the CSV export adds an extra column with row names as the first column, which is omitted within the CSV header.

Type CSV Header format. The following formats are supported

Type 0

name = None

namecol
Readonly name of column, that contains the row names. By default the column is infered from the used header format. For a **RFC 4180** compliant header by default no row names are used. For a header as used in exports of the R programming language by default the name of the first column is returned.

open(*mode: str = 'r'*, *columns: Optional[Tuple[str, ...]] = None*) → flib.io.csv.HandlerBase
Open CSV file in reading or writing mode.

Parameters

- **mode** – String, which characters specify the mode in which the file is to be opened. The default mode is *reading mode*, which is indicated by the character *r*. The character *w* indicates *writing mode*. Thereby reading- and writing mode are exclusive and can not be used together.
- **columns** – Has no effect in writing mode. For reading mode it specifies the columns, which are return from the CSV file by their respective column names. By default all columns are returned.

Returns In *reading mode* (if mode contains the character *w*) an instance of the class *Reader* is returned and in writing mode (if mode contains the character *r*) an instance of the class *Writer* is returned.

read(*columns: Optional[Tuple[str, ...]] = None*) → List[tuple]
Read all rows from current CSV file.

Parameters **columns** – Specifies the columns, which are return from the CSV file by their respective column names. By default all columns are returned.

Returns List of tuples, which contain the values of the specified columns.

rownames = None

write(*rows: List[tuple]*) → None
Write rows to current CSV file.

Parameters **rows** – List of tuples (or arbitrary iterables), which respectively contain the values of a single row.

class HandlerBase(*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], mode: str = 'r'*)
Bases: abc.ABC

CSV file I/O Handler Base Class.

Parameters

- **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, an instance of the class *Connector* or an opened file object in reading mode.

- **mode** – String, which characters specify the mode in which the file is to be opened. The default mode is *reading mode*, which is indicated by the character *r*. The character *w* indicates *writing mode*. Thereby reading- and writing mode are exclusive and can not be used together.

close() → None
Close the CSV file handler.

read_row() → tuple
Read a single row from the referenced file as a tuple.

read_rows() → List[tuple]
Read multiple rows from the referenced file as a list of tuples.

write_row(*row: Iterable[T_co]*) → None
Write a single row to the referenced file from a tuple.

write_rows(*rows: Iterable[tuple]*) → None
Write multiple rows to the referenced file from a list of tuples.

class Reader(*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], skiprows: int, usecols: Optional[Tuple[int, ...]], fields: List[Tuple[str, type]], **kwds*)
Bases: *flib.io.csv.HandlerBase*

CSV file I/O Reader Class.

Parameters

- **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, an instance of the class *Connector* or an opened file object in reading mode.
- **skiprows** – Number of initial lines within the given CSV file before the CSV Header. By default no lines are skipped.
- **usecols** – Tuple with column IDs of the columns, which are imported from the given CSV file. By default all columns are imported.
- **fields** – List (or arbitrary iterable) of field descriptors, respectively given by a tuple, containing a column name and a column type.
- ****kwds** – Formatting parameters used by csv. See also Dialects and formatting parameters

read_row() → tuple
Read a single row from the referenced file as a tuple.

read_rows() → List[tuple]
Read multiple rows from the referenced file as a list of tuples.

write_row(*row: Iterable[T_co]*) → None
Write a single row to the referenced file from a tuple.

write_rows(*rows: Iterable[tuple]*) → None
Write multiple rows to the referenced file from a list of tuples.

class Writer(*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], header: Iterable[str], comment: str = '', **kwds*)
Bases: *flib.io.csv.HandlerBase*

CSV file I/O Writer Class.

Parameters

- **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, an instance of the class `Connector` or an opened file object in writing mode.
- **header** – List (or arbitrary iterable) of column names, that specify the header of the CSV file.
- **comment** – Initial comment of the CSV file.
- ****kwds** – Formatting parameters used by `csv`. See also Dialects and formatting parameters

read_row() → tuple
Read a single row from the referenced file as a tuple.

read_rows() → List[tuple]
Read multiple rows from the referenced file as a list of tuples.

write_row(*row: Iterable[T_co]*) → None
Write a single row to the referenced file from a tuple.

write_rows(*rows: Iterable[tuple]*) → None
Write multiple rows to the referenced file from a list of tuples.

load(*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], delimiter: Optional[str] = None, hformat: Optional[int] = None*) → flib.io.csv.File
Load CSV file.

Parameters

- **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, an instance of the class `Connector` or an opened file object in reading or writing mode.
- **delimiter** – Single character, which is used to separetate the column values within the CSV file. By default the delimiter is detected from it's appearance within the file.
- **hformat** –

Returns Instance of class `flib.io.csv.File`

save(*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], header: Iterable[str], values: List[tuple], comment: Optional[str] = None, delimiter: Optional[str] = None, hformat: Optional[int] = None*) → None
Save data to CSV file.

Parameters

- **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, an instance of the class `Connector` or an opened file object in reading or writing mode.
- **header** – Optional list (or arbitrary iterable) of strings, that specify the column names within the CSV file. For an existing file, the header by default is extracted from the first content line (not blank and not starting with #). For a new file the header is required and an error is raised if the header is not given.
- **comment** – Optional string, which precedes the header and the rows of the CSV file, e.g. to include metadata within the file. For an existing file, the string by default is extracted from the initial comment lines (starting with #). For a new file the comment by default is empty.
- **delimiter** – Single character, which is used to separetate the column values within the CSV file. For an existing file, the delimiter by default is detected from it's appearance within the file. For a new file the default value is ,.

- **hformat** –

flib.io.ini module

File I/O for text files in Microsoft Windows INI file-format.

decode(*text: str, scheme: Union[Dict[str, Optional[type]], Dict[str, Dict[str, Optional[type]]], None] = None, autocast: bool = False, flat: Optional[bool] = None*) → Union[Dict[str, Optional[type]], Dict[str, Dict[str, Optional[type]]]]
Load configuration dictionary from INI-formated text.

Parameters

- **text** – Text, that describes a configuration in INI-format.
- **scheme** – Dictionary of dictionaries, which determines the structure of the configuration dictionary. If scheme is None, the INI-file is completely imported and all values are interpreted as strings. If the scheme is a dictionary of dictionaries, the keys of the outer dictionary describe valid section names by strings, that are interpreted as regular expressions. Therupon, the keys of the respective inner dictionaries describe valid parameter names as strings, that are also interpreted as regular expressions. Finally the values of the inner dictionaries define the type of the parameters by their own type, e.g. str, int, float etc. Accepted types can be found in the documentation of the function *literal.decode*.
- **autocast** – If no scheme is given autocast determines, if the values are automatically converted to types, estimated by the function *literal.estimate*
- **flat** – Determines if the desired INI format structure contains sections or not. By default sections are used, if the first non blank, non comment line in the string identifies a section.

Returns Structured configuration dictionary.

encode(*config: dict, flat: Optional[bool] = None, comment: Optional[str] = None*) → str
Convert configuration dictionary to INI formated string.

Parameters

- **config** – Configuration dictionary
- **flat** – Determines if the desired INI format structure contains sections or not. By default sections are used, if the dictionary contains subdictionaries.
- **comment** – String containing comment lines, which are stored as initial '#' lines in the INI-file. By default no comment is written.

Returns Text with INI-file structure.

get_comment(*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector]*) → str
Read initial comment lines from INI-file.

Parameters **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, a *file accessor* or an opened file object in reading or writing mode.

Returns String containing the initial comment lines of the INI-file or an empty string, if no initial comment lines could be detected.

load(*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], scheme: Union[Dict[str, Optional[type]], Dict[str, Dict[str, Optional[type]]], None] = None, autocast: bool = False, flat: Optional[bool] = None*) → Union[Dict[str, Optional[type]], Dict[str, Dict[str, Optional[type]]]]
Import configuration dictionary from INI file.

Parameters

- **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, a `file accessor` or an opened file object in reading mode.
- **scheme** – Dictionary of dictionaries, which determines the structure of the configuration dictionary. If scheme is None, the INI-file is completely imported and all values are interpreted as strings. If the scheme is a dictionary of dictionaries, the keys of the outer dictionary describe valid section names by strings, that are interpreted as regular expressions. Therupon, the keys of the respective inner dictionaries describe valid parameter names as strings, that are also interpreted as regular expressions. Finally the values of the inner dictionaries define the type of the parameters by their own type, e.g. str, int, float etc. Accepted types can be found in the documentation of the function `literal.decode`.
- **autocast** – If no scheme is given autocast determines, if the values are automatically converted to types, estimated by the function `literal.estimate`
- **flat** – Determines if the desired INI format structure contains sections or not. By default sections are used, if the first non empty, non comment line in the string identifies a section.

Returns Structured configuration dictionary

parse(*parser: configparser.ConfigParser*, *scheme: Optional[Dict[str, Dict[str, Optional[type]]]] = None*, *autocast: bool = False*) → Dict[str, Dict[str, Any]]
Import configuration dictionary from INI formated text.

Parameters

- **parser** – ConfigParser instance that contains an unstructured configuration dictionary
- **scheme** – Dictionary of dictionaries, which determines the structure of the configuration dictionary. If scheme is None, the INI-file is completely imported and all values are interpreted as strings. If the scheme is a dictionary of dictionaries, the keys of the outer dictionary describe valid section names by strings, that are interpreted as regular expressions. Therupon, the keys of the respective inner dictionaries describe valid parameter names as strings, that are also interpreted as regular expressions. Finally the values of the inner dictionaries define the type of the parameters by their own type, e.g. str, int, float etc. Accepted types can be found in the documentation of the function `literal.decode`.
- **autocast** – If no scheme is given autocast determines, if the values are automatically converted to types, estimated by the function `literal.estimate`

Returns Structured configuration dictionary.

save(*config: dict, file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], flat: Optional[bool] = None, comment: Optional[str] = None*) → None
Save configuration dictionary to INI-file.

Parameters

- **config** – Configuration dictionary
- **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, a `file accessor` or an opened file object in writing mode.
- **flat** – Determines if the desired INI format structure contains sections. By default sections are used, if the dictionary contains subdictionaries.
- **comment** – String containing comment lines, which are stored as initial '#' lines in the INI-file. By default no comment is written.

flib.io.plain module

File I/O for plain text files.

get_comment (*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector]*) → str
Read initial comment lines from text file.

Parameters **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, a `file accessor` or an opened file object in reading mode.

Returns String containing the header of given text file.

get_content (*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], lines: int = 0*) → List[str]
Read non-blank non-comment lines from text file.

Parameters

- **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, a `file accessor` or an opened file object in reading mode.
- **lines** – Number of content lines, that are returned. By default all lines are returned.

Returns List of strings containing non-blank non-comment lines.

get_name (*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector]*) → Optional[str]
Get name of referenced file object.

Parameters **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, a `file accessor` or an opened file object in reading or writing mode.

Returns String containing the name of the referenced file object or None if the name could not be determined.

load (*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector]*) → str
Load text from file.

Parameters **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, a `file accessor` or an opened file object in reading or writing mode.

Returns Content of the given file as text.

openx (*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], mode: str = 'rt'*) → Iterator[io.TextIOBase]
Contextmanager to provide a unified interface to text files.

This context manager extends the standard implementation of `open()` by allowing the passed argument *file* to be a str or path-like object, which points to a valid filename in the directory structure of the system, or a file object. If the *file* argument is a str or a path-like object, the given path may contain application variables, like '%home%' or '%user_data_dir%', which are extended before returning a file handler to a text file. Afterwards, when exiting the *with* statement, the file is closed. If the argument *file*, however, is a file-like object, the file is not closed, when exiting the *with* statement.

Parameters

- **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, a `file accessor` or an opened file object in reading or writing mode.

- **mode** – String, which characters specify the mode in which the file stream is wrapped. The default mode is reading mode. Suported characters are: 'r': Reading mode (default) 'w': Writing mode

Yields text file in reading or writing mode.

save (*text: str, file: Union[IO[Any], os.PathLike, flib.io.abc.Connector]*) → None
Save text to file.

Parameters

- **text** – Text given as string
- **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, a `file accessor` or an opened file object in writing mode.

flib.io.raw module

File I/O for binary files.

load (*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], encoding: Optional[str] = None, compressed: bool = False*) → bytes
Load binary data from file.

Parameters

- **file** – String or path-like object that points to a readable file in the directory structure of the system, or a file object in reading mode.
- **encoding** – Encodings specified in **RFC 3548**. Allowed values are: 'base16', 'base32', 'base64' and 'base85' or None for no encoding. By default no encoding is used.
- **compressed** – Boolean value which determines, if the returned binary data shall be decompressed by using :func:zlib.decompress.

Returns Content of the given file as bytes object.

openx (*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], mode: str = 'rb'*) → Iterator[io.BufferedIOBase]
Context manager to provide a unified interface to binary files.

This context manager extends the standard implementation of :py:func'open' by allowing the passed *file* argument to be a str or path-like object, which points to a valid filename in the directory structure of the system, or a file object. If the *file* argument is a str or a path-like object, the given path may contain application variables, like *%home%* or *%user_data_dir%*, which are extended before returning a file handler to a binary file. Afterwards, when exiting the *with*-statement, the file is closed. If the *file* argument, however, is a file object, the file is not closed, when exiting the *with*-statement.

Parameters

- **file** – String or path-like object that points to a valid filename in the directory structure of the system, or a file object.
- **mode** – String, which characters specify the mode in which the file stream is opened or wrapped. The default mode is reading mode. Suported characters are: 'r': Reading mode (default) 'w': Writing mode

Yields binary file in reading or writing mode.

save(*data: Union[bytes, bytearray, memoryview, str], file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], encoding: Optional[str] = None, compression: Optional[int] = None*) → None
Save binary data to file.

Parameters

- **data** – Binary data given as bytes-like object or string
- **file** – String or path-like object that points to a writable file in the directory structure of the system, or a file object in writing mode.
- **encoding** – Encodings specified in **RFC 3548**. Allowed values are: 'base16', 'base32', 'base64' and 'base85' or None for no encoding. By default no encoding is used.
- **compression** – Determines the compression level for `zlib.compress()`. By default no zlib compression is used. For an integer ranging from -1 to 9, a zlib compression with the respective compression level is used. Thereby *-1* is the default zlib compromise between speed and compression, *0* deflates the given binary data without attempted compression, *1* is the fastest compression with minimum compression capability and *9* is the slowest compression with maximum compression capability.

flib.io.zip module

File I/O for ZIP Archives.

class File(*filepath: Union[str, os.PathLike, None] = None, pwd: Optional[bytes] = None*)
Bases: `object`

In-Memory Zip Archives.

Parameters

- **filepath** – String or path-like object, that points to a valid ZipFile or None. If the filepath points to a valid ZipFile, then the class instance is initialized with a memory copy of the file. If the given file, however, does not exist or isn't a valid ZipFile, respectively one of the errors FileNotFoundError or BadZipFile is raised. The default behaviour, if the filepath is None, is to create an empty ZipFile in the memory.
- **pwd** – Bytes representing password of ZipFile.

append(*source: Union[str, os.PathLike], target: Union[str, os.PathLike, None] = None*) → bool
Append file to the ZipFile.

Parameters

- **source** – String or path-like object, that points to a valid file in the directory structure if the system. If the file does not exist, a FileNotFoundError is raised. If the filepath points to a directory, a IsADirectoryError is raised.
- **target** – String or path-like object, that points to a valid directory in the directory structure of the ZipFile. By default the root directory is used. If the directory does not exist, a FileNotFoundError is raised. If the target directory already contains a file, which name equals the filename of the source, a FileExistsError is raised.

Returns Boolean value which is True if the file has been appended.

changed
Tells whether the ZipFile has been changed.

close() → None
Close current ZipFile and buffer.

copy (*source: Union[str, os.PathLike], target: Union[str, os.PathLike]*) → bool
Copy file within ZipFile.

Parameters

- **source** – String or path-like object, that points to a file in the directory structure of the ZipFile. If the file does not exist, a FileNotFoundError is raised. If the filepath points to a directory, an IsADirectoryError is raised.
- **target** – String or path-like object, that points to a new filename or an existing directory in the directory structure of the ZipFile. If the target is a directory the target file consists of the directory and the basename of the source file. If the target file already exists a FileExistsError is raised.

Returns Boolean value which is True if the file was copied.

files
List of all files within the ZipFile.

folders
List of all folders within the ZipFile.

get_file_accessor (*path: Union[str, os.PathLike]*) → flib.io.abc.Connector
Get connector to ZipFile member.

Parameters **path** – String or path-like object, that represents a ZipFile member. In reading mode the path has to point to a valid ZipFile, or a FileNotFoundError is raised. In writing mode the path by default is treated as a file path. New directories can be written by setting the argument is_dir to True.

Returns *File accessor* to ZipFile member.

load (*filepath: Union[str, os.PathLike], pwd: Optional[bytes] = None*) → None
Load Workspace from file.

Parameters

- **filepath** – String or path-like object, that points to a valid ZipFile file. If the filepath points to a valid ZipFile, then the class instance is initialized with a memory copy of the file. If the given file, however, does not exist or isn't a valid ZipFile respectively one of the errors FileNotFoundError or BadZipFile is raised.
- **pwd** – Bytes representing password of ZipFile.

mkdir (*dirpath: Union[str, os.PathLike], ignore_exists: bool = False*) → bool
Create a new directory at the given path.

Parameters

- **dirpath** – String or path-like object, that represents a valid directory name in the directory structure of the ZipFile. If the directory already exists, the argument ignore_exists determines, if a FileExistsError is raised.
- **ignore_exists** – Boolean value which determines, if FileExistsError is raised, if the target directory already exists. The default behaviour is to raise an error, if the file already exists.

Returns Boolean value, which is True if the given directory was created.

move (*source: Union[str, os.PathLike], target: Union[str, os.PathLike]*) → bool
Move file within ZipFile.

Parameters

- **source** – String or path-like object, that points to a file in the directory structure of the ZipFile. If the file does not exist, a FileNotFoundError is raised. If the filepath points to a directory, an IsADirectoryError is raised.
- **target** – String or path-like object, that points to a new filename or an existing directory in the directory structure of the ZipFile. If the target is a directory the target file consists of the directory and the basename of the source file. If the target file already exists a FileExistsError is raised.

Returns Boolean value which is True if the file has been moved.

name
Filename of the ZipFile without file extension.

open(*path: Union[str, os.PathLike], mode: str = 'r', encoding: Optional[str] = None, is_dir: bool = False*) → IO[Any]
Open file within the ZipFile.

Parameters

- **path** – String or path-like object, that represents a ZipFile member. In reading mode the path has to point to a valid ZipFile, or a FileNotFoundError is raised. In writing mode the path by default is treated as a file path. New directories can be written by setting the argument is_dir to True.
- **mode** – String, which characters specify the mode in which the file is to be opened. The default mode is reading in text mode. Suported characters are: 'r': Reading mode (default) 'w': Writing mode 'b': Binary mode 't': Text mode (default)
- **encoding** – In binary mode encoding has not effect. In text mode encoding specifies the name of the encoding, which in reading and writing mode respectively is used to decode the stream's bytes into strings, and to encode strings into bytes. By default the preferred encoding of the operating system is used.
- **is_dir** – Boolean value which determines, if the path is to be treated as a directory or not. This information is required for writing directories to the ZipFile. The default behaviour is not to treat paths as directories.

Returns File object in reading or writing mode.

Examples

```
>>> with self.open('config.ini') as file:
>>>     print(file.read())
```

path
Filepath of the ZipFile.

read_bytes(*filepath: Union[str, os.PathLike]*) → bytes
Read bytes from file.

Parameters **filepath** – String or path-like object, that points to a valid file in the dirctory structure of the ZipFile. If the file does not exist a FileNotFoundError is raised.

Returns Contents of the given filepath as bytes.

read_text(*filepath: Union[str, os.PathLike], encoding: Optional[str] = None*) → str
Read text from file.

Parameters

- **filepath** – String or path-like object, that points to a valid file in the directory structure of the ZipFile. If the file does not exist a FileNotFoundError is raised.
- **encoding** – Specifies the name of the encoding, which is used to decode the stream's bytes into strings. By default the preferred encoding of the operating system is used.

Returns Contents of the given filepath encoded as string.

rmdir(*dirpath: Union[str, os.PathLike], recursive: bool = False, ignore_missing: bool = False*) → bool
Remove directory from ZipFile.

Parameters

- **dirpath** – String or path-like object, that points to a directory in the directory structure of the ZipFile. If the directory does not exist, the argument ignore_missing determines, if a FileNotFoundError is raised.
- **ignore_missing** – Boolean value which determines, if FileNotFoundError is raised, if the target directory does not exist. The default behaviour, is to raise an error if the directory is missing.
- **recursive** – Boolean value which determines, if directories are removed recursively. If recursive is False, then only empty directories can be removed. If recursive, however, is True, then all files and subdirectories are alse removed. By default recursive is False.

Returns Boolean value, which is True if the given directory was removed.

save() → None
Save the ZipFile to it's filepath.

saveas(*filepath: Union[str, os.PathLike]*) → None
Save the ZipFile to a file.

Parameters **filepath** – String or path-like object, that represents the name of a ZipFile.

search(*pattern: Optional[str] = None*) → List[str]
Search for files in the ZipFile.

Parameters **pattern** – Search pattern that contains Unix shell-style wildcards: '*': Matches arbitrary strings '?': Matches single characters [seq]: Matches any character in seq [!seq]: Matches any character not in seq By default a list of all files and directories is returned.

Returns List of files and directories in the directory structure of the ZipFile, that match the search pattern.

unlink(*filepath: Union[str, os.PathLike], ignore_missing: bool = True*) → bool
Remove file from ZipFile.

Parameters

- **filepath** – String or path-like object, that points to a file in the directory structure of the ZipFile. If the filepath points to a directory, an IsADirectoryError is raised. For the case, that the file does not exist, the argument ignore_missing determines, if a FileNotFoundError is raised.
- **ignore_missing** – Boolean value which determines, if FileNotFoundError is raised, if the target file does not exist. The default behaviour, is to ignore missing files.

Returns Boolean value, which is True if the given file was removed.

write_bytes(*blob: Union[bytes, bytearray, memoryview], filepath: Union[str, os.PathLike]*) → int
Write bytes to file.

Parameters

- **blob** – Bytes, which are to be written to the given file.
- **filepath** – String or path-like object, that represents a valid filename in the dirctory structure of the ZipFile.

Returns Number of bytes, that are written to the file.

write_text (*text: str, filepath: Union[str, os.PathLike], encoding: Optional[str] = None*) → int
Write text to file.

Parameters

- **text** – String, which has to be written to the given file.
- **filepath** – String or path-like object, that represents a valid filename in the dirctory structure of the ZipFile.
- **encoding** – Specifies the name of the encoding, which is used to encode strings into bytes. By default the preferred encoding of the operating system is used.

Returns Number of characters, that are written to the file.

Module contents

File I/O.

class FileConnector (*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], *args, **kwds*)
Bases: object

File Connector Class.

close () → None

name
Name of the referenced file object.

open (**args*, ***kwds*) → IO[Any]
Open file reference as file object.

class FileInfo (*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector]*)
Bases: object

File Info Class.

Parameters **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, an instance of the class *Connector* or an opened file object in reading or writing mode.

name
Name of the referenced file object.

class FileProxy (*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], mode: str = 'rw'*)
Bases: *flib.base.abc.Proxy*

File buffer for referenced files.

Creates a temporary file within the tempdir of the system, which acts as a local proxy for a referenced file object.

Parameters

- **file** – *File reference* to a file object. The reference can ether be given as a String or path-like object, that points to a valid entry in the file system, an instance of the class *Connector* or an opened file object in reading or writing mode.

- **mode** – String, which characters specify the mode in which the file stream is wrapped. If mode contains the character 'r', then a *pull()*-request is executed during the initialisation, otherwise any pull-request raises a *PullError*. If mode contains the character 'w', then a *push()*-request is executed when closing the FileProxy instance with *close()*, otherwise any push-request raises a *PushError*. The default mode is 'rw'.

close() → None
Execute push request and release bound resources.

connect(*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector]*) → None
Connect to given file reference.

disconnect() → None
Close connection to referenced file.

name
Name of the referenced file object

open(**args*, ***kwds*) → IO[Any]
Open file handler to temporary file.

path
Path to the temporary file in use.

pull() → None
Copy referenced file object to temporary file.

push() → None
Copy temporary file to referenced file object.

openx(*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector], *args, **kwds*) → Iterator[IO[Any]]
Open file reference.

This context manager extends :py:func'open' by allowing the passed *file* argument to be an arbitrary *file reference*. If the *file* argument is a String or path-like object, the given path may contain application variables, like *%home%* or *%user_data_dir%*, which are extended before returning a file handler. Afterwards, when exiting the *with*-statement, the file is closed. If the passes passed *file*, however, is a file object, the file is not closed, when exiting the *with*-statement.

Parameters

- **file** – *File reference* that points to a valid filename in the directory structure of the system, a file object or an instance of the class *Connector*.
- **mode** – String, which characters specify the mode in which the file stream is opened. The default mode is text reading mode. Supported characters are:

  r Reading mode (default)

  w Writing mode

  t Text mode

  b Binary mode

Yields file object in reading or writing mode.

tmpfile(*file: Union[IO[Any], os.PathLike, flib.io.abc.Connector]*) → Iterator[pathlib.Path]
Create a temporary file for a given file reference.

Parameters **file** – *File reference* that points to a valid filename in the directory structure of the system, a file object or an instance of the class *Connector*.

Yields path-like object that points to a temporary file.

flib.typing package

Submodules

flib.typing.check module

Check type and value of objects.

has_attr (*obj: object*, *attr: str*) → None
Check if object has an attribute.

has_opt_type (*name: str, obj: object, hint: Generic[T]*) → None
Check type of optional object.

has_size (*name: str*, *obj: Sized*, *size: Optional[int] = None*, *min_size: Optional[int] = None*, *max_size: Optional[int] = None*) → None
Check the size of a sized object.

has_type (*name: str, obj: object, hint: Generic[T]*) → None
Check type of object.

is_callable (*name: str*, *obj: object*) → None
Check if object is callable.

is_class (*name: str*, *obj: object*) → None
Check if object is a class.

is_identifier (*name: str*, *string: str*) → None
Check if a string is a valid identifier.

is_negative (*name: str, obj: Union[int, float]*) → None
Check if number is negative.

is_not_negative (*name: str, obj: Union[int, float]*) → None
Check if number is not negative.

is_not_positive (*name: str, obj: Union[int, float]*) → None
Check if number is not positive.

is_positive (*name: str, obj: Union[int, float]*) → None
Check if number is positive.

is_subclass (*name: str, obj: object, ref: Type[Any]*) → None
Check if object is a subclass of given class.

is_subset (*a: str*, *seta: set*, *b: str*, *setb: set*) → None
Check if a set is a subset of another.

is_typehint (*name: str*, *obj: object*) → None
Check if object is a supported typeinfo object.

no_dublicates (*name: str, coll: Collection[T_co]*) → None
Check if all elements of a collection are unique.

not_empty (*name: str*, *obj: Sized*) → None
Check if a sized object is not empty.

Module contents

Collection of Structural Types for Static Typing.

void(**args*, ***kwds*)

3.1.2 Module contents

Frootlab Shared Libarary.

The Frootlab Shared Library is a multi-purpose Python library, which primarily aims to support projects at Frootlab by a common base library. The majority of the comprised modules, however, is kept generic and well documented, to facilitate their application in other open source projects as well.

CHAPTER 4

Glossary

4.1 API Glossary

File Reference *File References* aggregate different types, that identify files, including: File objects, Strings and path-like objects, that point to filenames in the directory structure of the system and instances of the generic class `Connector`.

CHAPTER 5

License

The Frootlab Shared Library is distributed with the GNU General Public License v3 [GPLV3]:

```
GNU General Public License
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies of this license
document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and
other kinds of works.

The licenses for most software and other practical works are designed to
take away your freedom to share and change the works.  By contrast, the GNU
General Public License is intended to guarantee your freedom to share and
change all versions of a program--to make sure it remains free software for
all its users.  We, the Free Software Foundation, use the GNU General Public
License for most of our software; it applies also to any other work released
this way by its authors.  You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price.  Our
General Public Licenses are designed to make sure that you have the freedom
to distribute copies of free software (and charge for them if you wish),
that you receive source code or can get it if you want it, that you can
change the software or use pieces of it in new free programs, and that you
know you can do these things.

To protect your rights, we need to prevent others from denying you these
rights or asking you to surrender the rights.  Therefore, you have certain
responsibilities if you distribute copies of the software, or if you modify
it: responsibilities to respect the freedom of others.
```

(continues on next page)

(continued from previous page)

```
For example, if you distribute copies of such a program, whether gratis or
for a fee, you must pass on to the recipients the same freedoms that you
received.  You must make sure that they, too, receive or can get the source
code.  And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1)
assert copyright on the software, and (2) offer you this License giving you
legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that
there is no warranty for this free software.  For both users' and authors'
sake, the GPL requires that modified versions be marked as changed, so that
their problems will not be attributed erroneously to authors of previous
versions.

Some devices are designed to deny users access to install or run modified
versions of the software inside them, although the manufacturer can do so.
This is fundamentally incompatible with the aim of protecting users' freedom
to change the software.  The systematic pattern of such abuse occurs in the
area of products for individuals to use, which is precisely where it is most
unacceptable.  Therefore, we have designed this version of the GPL to
prohibit the practice for those products.  If such problems arise
substantially in other domains, we stand ready to extend this provision to
those domains in future versions of the GPL, as needed to protect the
freedom of users.

Finally, every program is threatened constantly by software patents. States
should not allow patents to restrict development and use of software on
general-purpose computers, but in those that do, we wish to avoid the
special danger that patents applied to a free program could make it
effectively proprietary.  To prevent this, the GPL assures that patents
cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification
follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of
works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License.
Each licensee is addressed as "you".  "Licensees" and "recipients" may be
individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a
fashion requiring copyright permission, other than the making of an exact
copy.  The resulting work is called a "modified version" of the earlier work
or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the
Program.
```

(continues on next page)

(continued from previous page)

```
To "propagate" a work means to do anything with it that, without permission,
would make you directly or secondarily liable for infringement under
applicable copyright law, except executing it on a computer or modifying a
private copy.  Propagation includes copying, distribution (with or without
modification), making available to the public, and in some countries other
activities as well.

To "convey" a work means any kind of propagation that enables other parties
to make or receive copies.  Mere interaction with a user through a computer
network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the
extent that it includes a convenient and prominently visible feature that
(1) displays an appropriate copyright notice, and (2) tells the user that
there is no warranty for the work (except to the extent that warranties are
provided), that licensees may convey the work under this License, and how to
view a copy of this License.  If the interface presents a list of user
commands or options, such as a menu, a prominent item in the list meets this
criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making
modifications to it.  "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official
standard defined by a recognized standards body, or, in the case of
interfaces specified for a particular programming language, one that is
widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than
the work as a whole, that (a) is included in the normal form of packaging a
Major Component, but which is not part of that Major Component, and (b)
serves only to enable use of the work with that Major Component, or to
implement a Standard Interface for which an implementation is available to
the public in source code form.  A "Major Component", in this context, means
a major essential component (kernel, window system, and so on) of the
specific operating system (if any) on which the executable work runs, or a
compiler used to produce the work, or an object code interpreter used to run
it.

The "Corresponding Source" for a work in object code form means all the
source code needed to generate, install, and (for an executable work) run
the object code and to modify the work, including scripts to control those
activities.  However, it does not include the work's System Libraries, or
general-purpose tools or generally available free programs which are used
unmodified in performing those activities but which are not part of the
work.  For example, Corresponding Source includes interface definition files
associated with source files for the work, and the source code for shared
libraries and dynamically linked subprograms that the work is specifically
designed to require, such as by intimate data communication or control flow
between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate
automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.
```

(continues on next page)

(continued from previous page)

```
2. Basic Permissions.

All rights granted under this License are granted for the term of copyright
on the Program, and are irrevocable provided the stated conditions are met.
This License explicitly affirms your unlimited permission to run the
unmodified Program.  The output from running a covered work is covered by
this License only if the output, given its content, constitutes a covered
work.  This License acknowledges your rights of fair use or other
equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey,
without conditions so long as your license otherwise remains in force.  You
may convey covered works to others for the sole purpose of having them make
modifications exclusively for you, or provide you with facilities for
running those works, provided that you comply with the terms of this License
in conveying all material for which you do not control copyright.  Those
thus making or running the covered works for you must do so exclusively on
your behalf, under your direction and control, on terms that prohibit them
from making any copies of your copyrighted material outside their
relationship with you.

Conveying under any other circumstances is permitted solely under the
conditions stated below. Sublicensing is not allowed; section 10 makes it
unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure
under any applicable law fulfilling obligations under article 11 of the WIPO
copyright treaty adopted on 20 December 1996, or similar laws prohibiting or
restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention is
effected by exercising rights under this License with respect to the covered
work, and you disclaim any intention to limit operation or modification of
the work as a means of enforcing, against the work's users, your or third
parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive
it, in any medium, provided that you conspicuously and appropriately publish
on each copy an appropriate copyright notice; keep intact all notices
stating that this License and any non-permissive terms added in accord with
section 7 apply to the code; keep intact all notices of the absence of any
warranty; and give all recipients a copy of this License along with the
Program.

You may charge any price or no price for each copy that you convey, and you
may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce
it from the Program, in the form of source code under the terms of section
```

(continues on next page)

(continued from previous page)

```
4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and
giving a relevant date.

b) The work must carry prominent notices stating that it is released under
this License and any conditions added under section 7.  This requirement
modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to
anyone who comes into possession of a copy.  This License will therefore
apply, along with any applicable section 7 additional terms, to the whole of
the work, and all its parts, regardless of how they are packaged.  This
License gives no permission to license the work in any other way, but it
does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display
Appropriate Legal Notices; however, if the Program has interactive
interfaces that do not display Appropriate Legal Notices, your work need not
make them do so.

A compilation of a covered work with other separate and independent works,
which are not by their nature extensions of the covered work, and which are
not combined with it such as to form a larger program, in or on a volume of
a storage or distribution medium, is called an "aggregate" if the
compilation and its resulting copyright are not used to limit the access or
legal rights of the compilation's users beyond what the individual works
permit.  Inclusion of a covered work in an aggregate does not cause this
License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of
sections 4 and 5, provided that you also convey the machine-readable
Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including
a physical distribution medium), accompanied by the Corresponding Source
fixed on a durable physical medium customarily used for software
interchange.

b) Convey the object code in, or embodied in, a physical product (including
a physical distribution medium), accompanied by a written offer, valid for
at least three years and valid for as long as you offer spare parts or
customer support for that product model, to give anyone who possesses the
object code either (1) a copy of the Corresponding Source for all the
software in the product that is covered by this License, on a durable
physical medium customarily used for software interchange, for a price no
more than your reasonable cost of physically performing this conveying of
source, or (2) access to copy the Corresponding Source from a network server
at no charge.

c) Convey individual copies of the object code with a copy of the written
offer to provide the Corresponding Source.  This alternative is allowed only
occasionally and noncommercially, and only if you received the object code
with such an offer, in accord with subsection 6b.
```

(continues on next page)

(continued from previous page)

```
d) Convey the object code by offering access from a designated place (gratis
or for a charge), and offer equivalent access to the Corresponding Source in
the same way through the same place at no further charge.  You need not
require recipients to copy the Corresponding Source along with the object
code.  If the place to copy the object code is a network server, the
Corresponding Source may be on a different server (operated by you or a
third party) that supports equivalent copying facilities, provided you
maintain clear directions next to the object code saying where to find the
Corresponding Source.  Regardless of what server hosts the Corresponding
Source, you remain obligated to ensure that it is available for as long as
needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you
inform other peers where the object code and Corresponding Source of the
work are being offered to the general public at no charge under subsection
6d.

A separable portion of the object code, whose source code is excluded from
the Corresponding Source as a System Library, need not be included in
conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family, or
household purposes, or (2) anything designed or sold for incorporation into
a dwelling.  In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage.  For a particular
product received by a particular user, "normally used" refers to a typical
or common use of that class of product, regardless of the status of the
particular user or of the way in which the particular user actually uses, or
expects or is expected to use, the product.  A product is a consumer product
regardless of whether the product has substantial commercial, industrial or
non-consumer uses, unless such uses represent the only significant mode of
use of the product.

"Installation Information" for a User Product means any methods, procedures,
authorization keys, or other information required to install and execute
modified versions of a covered work in that User Product from a modified
version of its Corresponding Source.  The information must suffice to ensure
that the continued functioning of the modified object code is in no case
prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as part of
a transaction in which the right of possession and use of the User Product
is transferred to the recipient in perpetuity or for a fixed term
(regardless of how the transaction is characterized), the Corresponding
Source conveyed under this section must be accompanied by the Installation
Information.  But this requirement does not apply if neither you nor any
third party retains the ability to install modified object code on the User
Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a
requirement to continue to provide support service, warranty, or updates for
a work that has been modified or installed by the recipient, or for the User
Product in which it has been modified or installed.  Access to a network may
be denied when the modification itself materially and adversely affects the
operation of the network or violates the rules and protocols for
```

(continues on next page)

(continued from previous page)

```
communication across the network.

Corresponding Source conveyed, and Installation Information provided, in
accord with this section must be in a format that is publicly documented
(and with an implementation available to the public in source code form),
and must require no special password or key for unpacking, reading or
copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License
by making exceptions from one or more of its conditions. Additional
permissions that are applicable to the entire Program shall be treated as
though they were included in this License, to the extent that they are valid
under applicable law.  If additional permissions apply only to part of the
Program, that part may be used separately under those permissions, but the
entire Program remains governed by this License without regard to the
additional permissions.

When you convey a copy of a covered work, you may at your option remove any
additional permissions from that copy, or from any part of it.  (Additional
permissions may be written to require their own removal in certain cases
when you modify the work.)  You may place additional permissions on
material, added by you to a covered work, for which you have or can give
appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to
a covered work, you may (if authorized by the copyright holders of that
material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of
sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author
attributions in that material or in the Appropriate Legal Notices displayed
by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or
requiring that modified versions of such material be marked in reasonable
ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors
of the material; or

e) Declining to grant rights under trademark law for use of some trade
names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by
anyone who conveys the material (or modified versions of it) with
contractual assumptions of liability to the recipient, for any liability
that these contractual assumptions directly impose on those licensors and
authors.

All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further restriction,
```

(continues on next page)

(continued from previous page)

```
you may remove that term.  If a license document contains a further
restriction but permits relicensing or conveying under this License, you may
add to a covered work material governed by the terms of that license
document, provided that the further restriction does not survive such
relicensing or conveying.

If you add terms to a covered work in accord with this section, you must
place, in the relevant source files, a statement of the additional terms
that apply to those files, or a notice indicating where to find the
applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of
a separately written license, or stated as exceptions; the above
requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided
under this License.  Any attempt otherwise to propagate or modify it is
void, and will automatically terminate your rights under this License
(including any patent licenses granted under the third paragraph of section
11).

However, if you cease all violation of this License, then your license from
a particular copyright holder is reinstated (a) provisionally, unless and
until the copyright holder explicitly and finally terminates your license,
and (b) permanently, if the copyright holder fails to notify you of the
violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated
permanently if the copyright holder notifies you of the violation by some
reasonable means, this is the first time you have received notice of
violation of this License (for any work) from that copyright holder, and you
cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under this
License.  If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same material
under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a
copy of the Program.  Ancillary propagation of a covered work occurring
solely as a consequence of using peer-to-peer transmission to receive a copy
likewise does not require acceptance.  However, nothing other than this
License grants you permission to propagate or modify any covered work.
These actions infringe copyright if you do not accept this License.
Therefore, by modifying or propagating a covered work, you indicate your
acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a
license from the original licensors, to run, modify and propagate that work,
subject to this License.  You are not responsible for enforcing compliance
```

(continues on next page)

(continued from previous page)

```
by third parties with this License.

An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations.  If propagation of a covered work
results from an entity transaction, each party to that transaction who
receives a copy of the work also receives whatever licenses to the work the
party's predecessor in interest had or could give under the previous
paragraph, plus a right to possession of the Corresponding Source of the
work from the predecessor in interest, if the predecessor has it or can get
it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights
granted or affirmed under this License.  For example, you may not impose a
license fee, royalty, or other charge for exercise of rights granted under
this License, and you may not initiate litigation (including a cross-claim
or counterclaim in a lawsuit) alleging that any patent claim is infringed by
making, using, selling, offering for sale, or importing the Program or any
portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License
of the Program or a work on which the Program is based.  The work thus
licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or
controlled by the contributor, whether already acquired or hereafter
acquired, that would be infringed by some manner, permitted by this License,
of making, using, or selling its contributor version, but do not include
claims that would be infringed only as a consequence of further modification
of the contributor version.  For purposes of this definition, "control"
includes the right to grant patent sublicenses in a manner consistent with
the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent
license under the contributor's essential patent claims, to make, use, sell,
offer for sale, import and otherwise run, modify and propagate the contents
of its contributor version.

In the following three paragraphs, a "patent license" is any express
agreement or commitment, however denominated, not to enforce a patent (such
as an express permission to practice a patent or covenant not to sue for
patent infringement).  To "grant" such a patent license to a party means to
make such an agreement or commitment not to enforce a patent against the
party.

If you convey a covered work, knowingly relying on a patent license, and the
Corresponding Source of the work is not available for anyone to copy, free
of charge and under the terms of this License, through a publicly available
network server or other readily accessible means, then you must either (1)
cause the Corresponding Source to be so available, or (2) arrange to deprive
yourself of the benefit of the patent license for this particular work, or
(3) arrange, in a manner consistent with the requirements of this License,
to extend the patent license to downstream recipients.  "Knowingly relying"
means you have actual knowledge that, but for the patent license, your
conveying the covered work in a country, or your recipient's use of the
```

(continues on next page)

(continued from previous page)

```
covered work in a country, would infringe one or more identifiable patents
in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement,
you convey, or propagate by procuring conveyance of, a covered work, and
grant a patent license to some of the parties receiving the covered work
authorizing them to use, propagate, modify or convey a specific copy of the
covered work, then the patent license you grant is automatically extended to
all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope
of its coverage, prohibits the exercise of, or is conditioned on the
non-exercise of one or more of the rights that are specifically granted
under this License.  You may not convey a covered work if you are a party to
an arrangement with a third party that is in the business of distributing
software, under which you make payment to the third party based on the
extent of your activity of conveying the work, and under which the third
party grants, to any of the parties who would receive the covered work from
you, a discriminatory patent license (a) in connection with copies of the
covered work conveyed by you (or copies made from those copies), or (b)
primarily for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement, or that
patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any
implied license or other defenses to infringement that may otherwise be
available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot convey a
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
not convey it at all.  For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey the
Program, the only way you could satisfy both those terms and this License
would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to
link or combine any covered work with a work licensed under version 3 of the
GNU Affero General Public License into a single combined work, and to convey
the resulting work.  The terms of this License will continue to apply to the
part which is the covered work, but the special requirements of the GNU
Affero General Public License, section 13, concerning interaction through a
network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the
GNU General Public License from time to time.  Such new versions will be
similar in spirit to the present version, but may differ in detail to
address new problems or concerns.
```

(continues on next page)

(continued from previous page)

```
Each version is given a distinguishing version number.  If the Program
specifies that a certain numbered version of the GNU General Public License
"or any later version" applies to it, you have the option of following the
terms and conditions either of that numbered version or of any later version
published by the Free Software Foundation.  If the Program does not specify
a version number of the GNU General Public License, you may choose any
version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of
the GNU General Public License can be used, that proxy's public statement of
acceptance of a version permanently authorizes you to choose that version
for the Program.

Later license versions may give you additional or different permissions.
However, no additional obligations are imposed on any author or copyright
holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE
LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR
OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND,
EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.  THE
ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU.
SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY
SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL
ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE
PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE
OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA
OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above
cannot be given local legal effect according to their terms, reviewing
courts shall apply local law that most closely approximates an absolute
waiver of all civil liability in connection with the Program, unless a
warranty or assumption of liability accompanies a copy of the Program in
return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible
use to the public, the best way to achieve this is to make it free software
which everyone can redistribute and change under these terms.
```

(continues on next page)

(continued from previous page)

```
To do so, attach the following notices to the program.  It is safest to
attach them to the start of each source file to most effectively state the
exclusion of warranty; and each file should have at least the "copyright"
line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software: you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the Free
Software Foundation, either version 3 of the License, or (at your option)
any later version.

This program is distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
more details.

You should have received a copy of the GNU General Public License along with
this program.  If not, see <https://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like
this when it starts in an interactive mode:

<program>  Copyright (C) <year>  <name of author> This program comes with
ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software,
and you are welcome to redistribute it under certain conditions; type `show
c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, your program's commands
might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary. For
more information on this, and how to apply and follow the GNU GPL, see
<https://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program
into proprietary programs.  If your program is a subroutine library, you may
consider it more useful to permit linking proprietary applications with the
library.  If this is what you want to do, use the GNU Lesser General Public
License instead of this License.  But first, please read
<https://www.gnu.org/licenses/why-not-lgpl.html>.
```

Python Module Index

f

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z