py-modelrunner Documentation

Release unknown

David Zwicker

Apr 18, 2024

CONTENTS

The *py-modelrunner* python package provides python classes for handling and running physical simulations. The main aim is to easily wrap simulation code and deal with input and output automatically. The package also facilitates submitting simulations to high performance computing environments and it provides functions for running parameter scans.

CHAPTER

ONE

CONTENTS

1.1 Installation

This *py-modelrunner* package is developed for python 3.8+ and should run on all common platforms.

1.1.1 Install using pip

The package is available on pypi, so you should be able to install it by running

```
pip install py-modelrunner
```

1.1.2 Install using conda

The *py-modelrunner* package is also available on conda using the *conda-forge* channel. You can thus install it using

```
conda install -c conda-forge py-modelrunner
```

This installation includes all required dependencies to have all features of *py-modelrunner*.

1.1.3 Installing from source

Prerequisites

The code builds on other python packages, which need to be installed for this package to function properly. The required packages are listed in the table below:

Package	Version	Usage
jinja2	>=2.7	Dealing with templates for launching simulations
h5py	>=3.5	Storing data in the HDF format
numpy	>=1.18.0	Array library used for storing data
pandas	>=1.3	Data tables for structured data access
PyYAML	>=5	Storing data in the YAML format
tqdm	>=4.45	Show progress bar

These package can be installed via your operating system's package manager, e.g. using **`conda`**, or **`pip`**. The package versions given above are minimal requirements, although this is not tested systematically. Generally, it should help to install the latest version of the package.

Downloading the package

The package can be simply checked out from github.com/zwicker-group/py-modelrunner. To import the package from any python session, it might be convenient to include the root folder of the package into the `PYTHONPATH` environment variable.

This documentation can be built by calling the **`make html`** in the `docs` folder. The final documentation will be available in `docs/build/html`. Note that a LaTeX documentation can be build using **`make latexpdf`**.

1.2 User Manual

1.2.1 Main structure of the package

The package offers several different components that can be used separately or together:

Hierarchical input/output using *`storage`*:
The module provides an abstract interface for writing and reading data using an hierarchical organization. Storages are conveniently created by *`open_storage()`*, which returns a storage object that offers a dict-like interface.

Defining parameters using *`parameters`*:
The module provides the *`Parameter`* class, describing a single parameter. This can be used together with the mixin *`Parameterized`*, which allows to equip classes with default parameters and some convenience methods.

Defining models using *`model`*:
Models are augmented functions, which define input, calculations, and output. Models can be conveniently created by decorating a function or by subclassing *`ModelBase`*, which is built on the parameter classes.

Model results are captured by *`results`*:
Results are returned as the special *`Result`*, which keeps track of the input parameters and the data calculated by the model. *`ResultCollection`* describes a collections of the same model evoked with different parameter values.

Submitting models to HPC using *`run`*:
A single model can be submitted to a compute node using *`submit_job()`*, e.g., to run the computation on a high performance compute cluster. A parameter study using multiple jobs can be conveniently submitted using *`submit_jobs()`*. The results written to one directory can then be conveniently analyzed using *`from_folder()`*.

1.2.2 Design philosophy

The main requirements for the package can be summarized as follows:

- **Usability**: The user should not need to think about how data is stored in different files. The *`Result`* class should simply work.
- **Flexibility**: *`storage`* should provide a unified interface to write data in multiple file formats (JSON, YAML, HDF, zarr, ...)
- **Stability**: Future versions of the package should be able to read older files even when the internal definitions of file formats change
- **Modularity**: Different parts of the package (like *`storage`*, *`parameters`*, and *`run`*) should be rather independent of each other, so they can be used in isolation
- **Extensibility**: Models inherting from *`ModelBase`* should be easy to subclass to implement more complicated requirements (e.g., additional parameters)

- **Self-explainability**: The files should in principle contain all information to reconstruct the data, even if the *modelrunner* package is no longer available.
- **Efficiency**: The files should only store necessary information.

The last point results in particular constraints if we want to store temporal simulation results. In most cases, there are are some data that are kept fixed for the simulation (describing physical parameters) and others that evolve with time. We denote by *attributes* the parameters that are kept fixed and by *data* the data that varies over time. The *trajectory* module deals with such data.

1.3 Examples

The following examples showcase some functionality of the package

1.3.1 io_hdf.py

This example shows reading and writing data using HDF files.

```
import numpy as np

from modelrunner import Result, run_function_with_cmd_args


def number_range(start: float = 1, length: int = 3):
    """create an ascending list of numbers"""
    return start + np.arange(length)


if __name__ == "__main__":
    # write result to file
    result = run_function_with_cmd_args(number_range)
    result.to_file("test.hdf")

    # write result from file
    read = Result.from_file("test.hdf")
    print(read.parameters, "-- start + [0..length-1] =", read.result)
```

1.3.2 io_json.py

This example shows reading and writing data using JSON files.

```
from modelrunner import Result, run_function_with_cmd_args


def multiply(a: float = 1, b: float = 2):
    """Multiply two numbers"""
    return a * b


if __name__ == "__main__":
    result = run_function_with_cmd_args(multiply)
```

(continues on next page)

(continued from previous page)

```
    result.to_file("test.json")

    read = Result.from_file("test.json")
    print(read.parameters, "-- a * b =", read.result)
```

1.3.3 io_yaml.py

This example shows reading and writing data using JSON files.

```
from modelrunner import Result, run_function_with_cmd_args


def multiply(a: float = 1, b: float = 2):
    """Multiply two numbers"""
    return a * b


if __name__ == "__main__":
    result = run_function_with_cmd_args(multiply)
    result.to_file("test.yaml")

    read = Result.from_file("test.yaml")
    print(read.parameters, "-- a * b =", read.result)
```

1.3.4 model_class.py

This example shows defining a custom model class by subclassing.

```
import numpy as np

from modelrunner import ModelBase


class MyModel(ModelBase):
    parameters_default = {"a": 1, "b": 2}

    def __call__(self):
        self.storage.write_array("arr", np.arange(4))
        self.storage.write_attrs("arr", {"key": "value"})  # write extra information
        return self.parameters["a"] * self.parameters["b"]


# create an instance of the model
model = MyModel({"a": 3}, output="test.yaml")
# run the instance
print(model())
```

1.3.5 model_decorator.py

This example shows defining a custom model class using a decorator on a function.

```
import modelrunner


@modelrunner.make_model
def multiply(a, b=2):
    return a * b


# use the model instance
print(multiply.parameters)
print(multiply(a=3))
```

1.3.6 model_function.py

This example shows defining a custom model class using a function.

```
from modelrunner import make_model


def multiply(a, b=2):
    return a * b


# create an instance of the model defined by the function
model = make_model(multiply, {"a": 3})
# run the instance
print(model())
```

1.3.7 model_storage_output.py

This example shows defining a custom model that stores additional data

```
import tempfile

from modelrunner import make_model, open_storage


def multiply(a, b=2, storage=None):
    storage["data"] = {"additional": "information"}
    return a * b


with tempfile.NamedTemporaryFile(suffix=".yaml") as fp:
    # create an instance of the model defined by the function
    model = make_model(multiply, {"a": 3}, output=fp.name)
    # run the instance and store the data
    model.write_result()
```

(continues on next page)

(continued from previous page)

```
    # read the file and check whether all the data is there
    with open_storage(fp.name) as storage:
        print("Stored data:", storage["storage/data"])
        print("Model result:", storage["result"])
```

1.3.8 read_many.py

This example shows how a collection of results can be read.

```
import os

from modelrunner import ResultCollection, make_model_class


def multiply(a: float = 1, b: float = 2):
    """Multiply two numbers"""
    return a * b


if __name__ == "__main__":
    # create model class
    model = make_model_class(multiply)

    # write data
    os.makedirs("data")
    for n, a in enumerate(range(5, 10)):
        result = model({"a": a}).get_result()
        result.to_file(f"data/test_{n}.json")

    # read data
    rc = ResultCollection.from_folder("data")
    print(rc.dataframe)
```

1.3.9 script_custom_func.py

This example shows how a function is turned into a model using decorators.

```
#!/usr/bin/env python3 -m modelrunner

import modelrunner


def do_not_calculate(a=1, b=2):
    """This function should not be run"""
    raise RuntimeError("This must not run")


@modelrunner.make_model
def calculate(a=1, b=2):
```

(continues on next page)

(continued from previous page)

```
    """This function has been marked as a model"""
    print(a * b)
```

1.3.10 script_function.py

This example shows how a function is turned into a model explicitly.

```
from modelrunner import run_function_with_cmd_args


def multiply(a: float = 1, b: float = 2):
    """Multiply two numbers"""
    return a * b


if __name__ == "__main__":
    result = run_function_with_cmd_args(multiply)
    print(result.result)
```

1.3.11 script_many_classes.py

This example displays a minimal script containing two model classes

```
import sys

from modelrunner import ModelBase


class MyModel(ModelBase):
    parameters_default = {"a": 1, "b": 2}

    def __call__(self):
        return self.parameters["a"] * self.parameters["b"]


class MyDerivedModel(MyModel):
    parameters_default = {"c": 3}

    def __call__(self):
        print(super().__call__() + self.parameters["c"])


if __name__ == "__main__":
    MyDerivedModel.run_from_command_line(sys.argv[1:])
```

1.3.12 script_minimal.py

This example displays a minimal script defining a model using a *main* function.

```
#!/usr/bin/env python3 -m modelrunner

def main(a=1, b=2):
    """Multiply two numbers"""
    print(a * b)
```

1.3.13 script_model_class.py

This example displays a minimal script containing a model class.

```
#!/usr/bin/env python3 -m modelrunner

from modelrunner import ModelBase


class MyModel(ModelBase):
    parameters_default = {"a": 1, "b": 2}

    def __call__(self):
        print(self.parameters["a"] * self.parameters["b"])
```

1.3.14 submit_function.py

This example shows how to submit a model to a queuing system.

Note that the method *foreground* just runs the script locally, thus not really queuing. To actually queue a job on a high performance computing cluster, replace the *method* argument by something more suitable; see the documentation.

```
from modelrunner import make_model, submit_job


@make_model
def multiply(a: float = 1, b: float = 2):
    """Multiply two numbers"""
    return a * b


if __name__ == "__main__":
    submit_job(__file__, output="data.json", method="foreground")
```

1.3.15 submit_many_func.py

This example shows how to submit the same model with multiple parameters.

Note that the method *foreground* just runs the script locally, thus not really queuing. To actually queue a job on a high performance computing cluster, replace the *method* argument by something more suitable; see the documentation.

```
from modelrunner import make_model, submit_jobs


@make_model
def main(a: float = 1, b: float = 2):
    """Multiply two numbers"""
    return a * b


if __name__ == "__main__":
    submit_jobs(
        __file__,  # submit this file as a job module
        output_folder="data",
        parameters={"a": [1, 2], "b": [4, 5]},
        method="foreground",  # run job locally
    )
```

1.3.16 submit_many_iter.py

This example shows how to submit the same model with multiple parameters.

Note that the method *foreground* just runs the script locally, thus not really queuing. To actually queue a job on a high performance computing cluster, replace the *method* argument by something more suitable; see the documentation.

```
from modelrunner import make_model, submit_job


@make_model
def main(a: float = 1, b: float = 2):
    """Multiply two numbers"""
    return a * b


if __name__ == "__main__":
    for a in [1, 2]:
        for b in [4, 5]:
            name = f"job_a_{a}_b_{b}"
            submit_job(
                __file__,  # submit this file as a job module
                output=f"data/{name}.json",
                name=name,
                parameters={"a": a, "b": b},
                method="foreground",  # run job locally
            )
```

1.3.17 submit_shebang.py

This example shows how to submit a model to a queuing system using the magic line above.

Note that the method *foreground* just runs the script locally, thus not really queuing. To actually queue a job on a high performance computing cluster, replace the *method* argument by something more suitable; see the documentation.

```
#!/usr/bin/env python3 -m modelrunner.run --output data.hdf5 --method foreground

def multiply(a: float = 1, b: float = 2):
    """Multiply two numbers"""
    return a * b
```

1.4 modelrunner package

Subpackages:

1.4.1 modelrunner.model package

Defines models that handle a simulation and its inputs

ModelBase	base class for describing models
make_model	create model from a function and a dictionary of parameters
make_model_class	create a model from a function by interpreting its signature
set_default	sets the function or model as the default model
Parameter	class representing a single parameter
Parameterized	a mixin that manages the parameters of a class

modelrunner.model.base module

Base class describing a model

class ModelBase(*parameters=None*, *output=None*, ***, *mode='insert'*, *strict=False*)

Bases: *Parameterized*

base class for describing models

initialize the parameters of the object

Parameters

- **parameters** (*dict*) – A dictionary of parameters to change the defaults of this model. The allowed parameters can be obtained from `get_parameters()` or displayed by calling `show_parameters()`.
- **output** (*str*) – Path where the output file will be written. The output will be written using *storage* and might contain two groups: *result* to which the final result of the model is written, and *data*, which can contain extra information that is written using *storage*.
- **mode** (str or `ModeType`) – The file mode with which the storage is accessed, which determines the allowed operations. Common options are “read”, “full”, “append”, and “truncate”.

- **strict** (*bool*) – Flag indicating whether parameters are strictly interpreted. If *True*, only parameters listed in *parameters_default* can be set and their type will be enforced.

close()

close any opened storages

Return type
None

description: str | None = None

a longer description of the model

Type
str

classmethod **from_command_line**(*args=None*, *name=None*)

create model from command line parameters

Parameters

- **args** (*list*) – Sequence of strings corresponding to the command line arguments
- **name** (*str*) – Name of the program, which will be shown in the command line help

Returns
An instance of this model with appropriate parameters

Return type
ModelBase

get_result(*data=None*)

get the result as a `Result` object

Parameters
data (*Any*) – The result data. If omitted, the model is run to obtain results

Returns
The result after the model is run

Return type
`Result`

name: str | None = None

the name of the model

Type
str

classmethod **run_from_command_line**(*args=None*, *name=None*)

run model using command line parameters

Parameters

- **args** (*list*) – Sequence of strings corresponding to the command line arguments
- **name** (*str*) – Name of the program, which will be shown in the command line help

Returns
The result of running the model

Return type
`Result`

property storage: *StorageGroup*

Storage to which data can be written

Type
StorageGroup

write_result(*result=None*)

write the result to the output file

Parameters
result (Result | None) – The result data. If omitted, the model is run to obtain results

Return type
None

modelrunner.model.factory module

Functions for creating models and model classes from other input

cleared_default_model(*func*)

run the function with a cleared _DEFAULT_MODEL and restore it afterwards

Parameters
func (*TFunc*) –

Return type
TFunc

make_model(*func*, *parameters=None*, *output=None*, *, *mode='insert'*, *default=False*)

create model from a function and a dictionary of parameters

Parameters

- **func** (*callable*) – The function that will be turned into a Model
- **parameters** (*dict*) – Paramter values with which the model is initialized
- **output** (*str*) – Path where the output file will be written.
- **mode** (str or ModeType) – The file mode with which the storage is accessed, which determines the allowed operations. Common options are “read”, “full”, “append”, and “truncate”.
- **default** (*bool*) – If True, set this model as the default one for the current script

Returns
An instance of a subclass of ModelBase encompassing *func*

Return type
ModelBase

make_model_class(*func*, *, *default=False*)

create a model from a function by interpreting its signature

Parameters

- **func** (*callable*) – The function that will be turned into a Model
- **default** (*bool*) – If True, set this model as the default one for the current script

Returns
A subclass of ModelBase, which encompasses *func*

Return type
ModelBase

set_default(*func_or_model*)

sets the function or model as the default model

The last model that received this flag will be run automatically. This only affects the behavior when the script is run using *modelrunner* from the command line, e.g., using `python -m modelrunner script.py`.

Parameters
func_or_model (callabel or ModelBase, optional) – The function or model that should be called when the script is run.

Returns
func_or_model, so the function can be used as a decorator

Return type
TModel

modelrunner.model.parameters module

Infrastructure for managing classes with parameters.

One aim is to allow easy management of inheritance of parameters.

Parameter	class representing a single parameter
DeprecatedParameter	a parameter that can still be used normally but is deprecated
HideParameter	a helper class that allows hiding parameters of the parent classes
Parameterized	a mixin that manages the parameters of a class
get_all_parameters	get a dictionary with all parameters of all registered classes

class DeprecatedParameter(*name*, *default_value=None*, *cls=<class 'object'>*, *description=''*, *choices=None*, *required=False*, *hidden=False*, *extra=<factory>*)

Bases: *Parameter*

a parameter that can still be used normally but is deprecated

Parameters
- **name** (*str*) –
- **default_value** (*Any*) –
- **cls** (*type | Callable*) –
- **description** (*str*) –
- **choices** (*Container | None*) –
- **required** (*bool*) –
- **hidden** (*bool*) –
- **extra** (*dict[str, Any]*) –

extra: dict[str, Any]

name: str

class HideParameter(*name*)

Bases: object

a helper class that allows hiding parameters of the parent classes

This parameter will still appear in the `parameters` dictionary, but it will typically not be visible to the user, e.g., when calling `show_parameters()`.

Parameters

name (*str*) – The name of the parameter

class NoValueType

Bases: object

special value to indicate no value for a parameter

class Parameter(*name*, *default_value=None*, *cls=<class 'object'>*, *description=''*, *choices=None*, *required=False*, *hidden=False*, *extra=<factory>*)

Bases: object

class representing a single parameter

Parameters

- **name** (*str*) – The name of the parameter
- **default_value** (*Any*) – The default value of the parameter
- **cls** (*type | Callable*) – The type of the parameter, which is used for conversion. The conversion and parsing of values can be disabled by using the default class *object*.
- **description** (*str*) – A string describing the impact of this parameter. This description appears in the parameter help.
- **choices** (*container*) – A list or set of values that the parameter can take. Values (including the default value) that are not in this list will be rejected. Note that values are check after they have been converted by *cls*, so specifying *cls* is particularly important to convert command line parameters from strings.
- **required** (*bool*) – Whether the parameter is required
- **hidden** (*bool*) – Whether the parameter is hidden in the description summary
- **extra** (*dict*) – Extra arguments that are stored with the parameter

choices: Container | None = None

cls

alias of object

convert(*value=NoValue*, ***, *strict=True*)

converts a *value* into the correct type for this parameter. If *value* is not given, the default value is converted.

Note that this does not make a copy of the values, which could lead to unexpected effects where the default value is changed by an instance.

Parameters

- **value** – The value to convert
- **strict** (*bool*) – Flag indicating whether conversion to the type indicated by *cls* is enforced. If *False*, the original value is returned when conversion fails.

Returns
The converted value, which is of type *self.cls*

default_value: Any = None

description: str = ''

extra: dict[str, Any]

hidden: bool = False

name: str

required: bool = False

property short_description: str
return only the first sentence of the description

class Parameterized(*parameters=None*, *, *strict=True*)
Bases: object

a mixin that manages the parameters of a class

initialize the parameters of the object

Parameters

- **parameters** (*dict*) – A dictionary of parameters to change the defaults. The allowed parameters can be obtained from *get_parameters()* or displayed by calling *show_parameters()*.
- **strict** (*bool*) – Flag indicating whether parameters are strictly interpreted. If *True*, only parameters listed in *parameters_default* can be set and their type will be enforced.

classmethod get_parameter_default(*name*)
return the default value for the parameter with *name*

Parameters
name (*str*) – The parameter name

classmethod get_parameters(*include_hidden=False*, *include_deprecated=False*, *sort=True*)
return a dictionary of parameters that the class supports

Parameters

- **include_hidden** (*bool*) – Include hidden parameters
- **include_deprecated** (*bool*) – Include deprecated parameters
- **sort** (*bool*) – Return ordered dictionary with sorted keys

Returns
a dictionary mapping names to instances of *Parameter*

Return type
dict

parameters_default: ParameterListType = []
parameters (with default values) of this subclass

Type
list

show_parameters(*description=False*, *sort=False*, *show_hidden=False*, *show_deprecated=False*)

show all parameters in human readable format

Parameters

- **description** (*bool*) – Flag determining whether the parameter description is shown.
- **sort** (*bool*) – Flag determining whether the parameters are sorted
- **show_hidden** (*bool*) – Flag determining whether hidden parameters are shown
- **show_deprecated** (*bool*) – Flag determining whether deprecated parameters are shown

Return type
None

All flags default to *False*.

auto_type(*value*)

convert value to float or int if reasonable

get_all_parameters(*data='name'*)

get a dictionary with all parameters of all registered classes

Parameters
data (*str*) – Determines what data is returned. Possible values are 'name', 'value', or 'description', to return the respective information about the parameters.

Return type
dict[str, Any]

1.4.2 modelrunner.run package

Defines classes and function used to run models defined using *model*

submit_job	submit a script to the cluster queue
submit_jobs	submit many jobs of the same script with different parameters to the cluster
run_function_with_cmd_args	create model from a function and obtain parameters from command line
run_script	helper function that runs a model script
Result	describes the result of a single model run together with auxillary information
ResultCollection	represents a collection of results

Subpackages:

modelrunner.run.compatibility package

Code for maintaining backwards compatibility, particularly for loading results

modelrunner.run.compatibility.triage module

Contains code necessary for deciding which format version was used to write a file

guess_format(*path*)

guess the format of a given store

Parameters
path (str or Path) – Path pointing to a file

Returns
The store format

Return type
str

normalize_zarr_store(*store*, *mode='a'*)

determine best file format for zarr storage

In particular, we use a ZipStore when a path looking like a file is given.

Parameters

- **store** (*Store*) – User-provided store
- **mode** (*str*) – The mode with which the file will be opened

Return type
Store | None

Returns:

result_check_load_old_version(*path*, *loc*, *, *model=None*)

check whether the resource can be loaded with an older version of the package

Parameters

- **path** (str or Path) – The path to the resource to be loaded
- **loc** (*str*) – Label, key, or location of the item to be loaded
- **model** (ModelBase, optional) – Optional model that was used to write this result

Returns
The loaded result or *None* if we cannot load it with the old versions

Return type
Result

modelrunner.run.compatibility.version0 module

Contains code necessary for loading results from format version 0

read_hdf_data(*node*)

read structured data written with `write_hdf_dataset()` from an HDF node

result_from_file_v0(*path*, ***kwargs*)

load object from a file using format version 1

Parameters

- **store** (str or `zarr.Store`) – Path or instance describing the storage, which is either a file path or a `zarr.Storage`.
- **path** (*Path*) –

Return type

Result

modelrunner.run.compatibility.version1 module

Contains code necessary for loading results from format version 1

class ArrayCollectionState

Bases: *StateBase*

class ArrayState

Bases: *StateBase*

class DictState

Bases: *StateBase*

class NoData

Bases: `object`

helper class that marks data omission

class ObjectState

Bases: *StateBase*

class StateBase

Bases: `object`

Base class for specifying the state of a simulation

A state contains values of all degrees of freedom of a physical system (called the *data*) and some additional information (called *attributes*). The *data* is mutable and often a numpy array or a collection of numpy arrays. Conversely, the *attributes* are stroed in a dictionary with immutable values. To allow flexible storage, we define the properties *_state_data* and *_state_attributes*, which by default return *attributes* and *data* directly, but may be overwritten to process the data before storage (e.g., by additional serialization).

classmethod from_data(*attributes*, *data=<class 'modelrunner.run.compatibility.version1.NoData'>*)

create instance of any state class from attributes and data

Parameters

- **attributes** (*dict*) – Additional (unserialized) attributes
- **data** – The data of the degerees of freedom of the physical system

Returns
The object containing the given attributes and data

Return type
TState

`result_from_file_v1`(*store*, *, *label='data'*, ***kwargs*)

load object from a file using format version 1

Parameters

- **store** (*Path*) – Path of file to read
- **fmt** (*str*) – Explicit file format. Determined from *store* if omitted.
- **label** (*str*) – Name of the node in which the data was stored. This applies to some hierarchical storage formats.

Return type
Result

modelrunner.run.compatibility.version2 module

Contains code necessary for loading results from format version 2

`result_from_file_v2`(*store*, *, *loc='result'*, ***kwargs*)

load object from a file using format version 1

Parameters

- **store** (*Path*) – Path of file to read
- **fmt** (*str*) – Explicit file format. Determined from *store* if omitted.
- **label** (*str*) – Name of the node in which the data was stored. This applies to some hierarchical storage formats.
- **loc** (*str*) –

Return type
Result

modelrunner.run.job module

Provides functions for submitting models as jobs

`ensure_directory_exists`(*folder*)

creates a folder if it not already exists

`escape_string`(*obj*)

escape a string for the command line

Return type
str

`get_config`(*config=None*, *, *load_user_config=True*)

create the job configuration

Parameters

- **config** (*str or dict*) – Configuration settings that will be used to update the default config
- **load_user_config** (*bool*) – Determines whether the file *~/.modelrunner* is loaded as a YAML document to provide user-defined settings.

Returns
the established configuration

Return type
Config

get_job_name(*base*, *args=None*, *length=7*)

create a suitable job name

Parameters

- **base** (*str*) – The stem of the job name
- **args** (*dict*) – Parameters to include in the job name
- **length** (*int*) – Length of the abbreviated parameter name

Returns
A suitable job name

Return type
str

submit_job(*script*, *output=None*, *name='job'*, *parameters=None*, *config=None*, ***, *log_folder=None*, *method='auto'*, *use_modelrunner=True*, *template=None*, *overwrite_strategy='error'*, ***kwargs*)

submit a script to the cluster queue

Parameters

- **script** (str of Path) – Path to the script file, which contains the model
- **output** (str of Path) – Path to the output file, where all the results are saved
- **name** (*str*) – Name of the job
- **parameters** (*str or dict*) – Parameters for the script, either as a python dictionary or a string containing a JSON-encoded dictionary.
- **config** (*str or dict*) – Configuration for the job, which determines how the job is run. Can be either a python dictionary or a string containing a JSON-encoded dictionary.
- **log_folder** (str of Path) – Path to the logging folder. If omitted, the default of the template is used, which typically sends data to stdout for local scripts (which is thus captured and returned by this function) or writes log files to the current working directory for remote jobs.
- **method** (*str*) – Specifies the submission method. Currently *background*, *foreground*, 'srun', and *qsub* are supported. The special value *auto* reads the method from the *config* argument.
- **use_modelrunner** (*bool*) – If True, *script* is envoked with the modelrunner library, e.g. by calling *python -m modelrunner {script}*.
- **template** (str of Path) – Jinja template file for submission script. If omitted, a standard template is chosen based on the submission method.
- **overwrite_strategy** (*str*) – Determines what to do when files already exist. Possible options include *error*, *warn_skip*, *silent_skip*, *overwrite*, and *silent_overwrite*.

Returns

The result *(stdout, stderr)* of the submission call. These two strings
can contain the output from the actual scripts that is run when *log_folder* is *None*.

Return type
tuple

submit_jobs(*script*, *output_folder*, *name_base='job'*, *parameters=None*, *config=None*, ***, *output_format='hdf'*, *list_params=None*, ***kwargs*)

submit many jobs of the same script with different parameters to the cluster

Parameters

- **script** (str of Path) – Path to the script file, which contains the model
- **output_folder** (str of Path) – Path to the output folder, where all the results are saved
- **name_base** (*str*) – Base name of the job. An automatic name is generated on this basis.
- **parameters** (*str or dict*) – Parameters for the script, either as a python dictionary or a string containing a JSON-encoded dictionary. All combinations of parameter values that are iterable and not strings and not part of *keep_list* are submitted as separate jobs.
- **config** (*str or dict*) – Configuration for the job, which determines how the job is run. Can be either a python dictionary or a string containing a JSON-encoded dictionary.
- **output_format** (*str*) – File extension determining the output format
- **list_params** (*list*) – List of parameters that are meant to be lists. They will be submitted as individual parameters and not iterated over to produce multiple jobs.
- ****kwargs** – All additional parameters are forwarded to *submit_job()*.

Returns
The number of jobs that have been submitted

Return type
int

modelrunner.run.launch module

Base class describing a model

run_function_with_cmd_args(*func*, *args=None*, ***, *name=None*)

create model from a function and obtain parameters from command line

Parameters

- **func** (*callable*) – The function that will be turned into a Model
- **args** (*list of str*) – Command line arguments, typically `sys.argv[1:]`
- **name** (*str*) – Name of the program, which will be shown in the command line help

Returns
An instance of a subclass of ModelBase encompassing *func*

Return type
ModelBase

run_script(*script_path*, *model_args*)

helper function that runs a model script

The function detects models automatically by trying several methods until one yields a unique model to run:

- A model that have been marked as default by `set_default()`
- A function named *main*
- A model instance if there is exactly one (throw error if there are many)
- A model class if there is exactly one (throw error if there are many)
- A function if there is exactly one (throw error if there are many)

Parameters

- **script_path** (*str*) – Path to the script that contains the model definition
- **model_args** (*sequence*) – Additional arugments that define how the model is run

Returns
The result of the run

Return type
`Result`

modelrunner.run.results module

Classes that describe results of simulations of models

class MockModel(*parameters=None*)

Bases: *ModelBase*

helper class to store parameter values when the original model is not present

Parameters
parameters (*dict*) – A dictionary of parameters

class Result(*model*, *result*, *, *storage=None*, *info=None*)

Bases: `object`

describes the result of a single model run together with auxillary information

Besides storing the final outcome of the model in *result*, the class also stores information about the original model in *model*, additional information in *info*, and potentially arbitrary objects that were added during the model run in *storage*.

Note: The result is represented as a hierarchical structure when safed using the *storage*. The actual result is stored in the *result* group, whereas the model information can be found in *_model* group. Additional information is stored in the *storage* group. Thus, the full *Result* can be read using `storage[loc]`, where *loc* denotes the result location. If only the actual result is needed, `storage[loc + "/result"]` can be read.

Parameters

- **model** (`ModelBase`) – The model from which the result was obtained
- **result** (*Any*) – The actual result
- **storage** (`StorageGroup | None`) – A storage containing additional data from the model run
- **info** (*dict*) – Additional information for this result

property data

direct access to the underlying state data

classmethod from_data(*model_data*, *result*, *, *model=None*, *storage=None*, *info=None*)

create result from data

Parameters

- **model_data** (*dict*) – The data identifying the model
- **result** – The actual result data
- **model** (ModelBase) – The model from which the result was obtained
- **storage** (StorageGroup | *None*) – A storage containing additional data from the model run
- **info** (*dict*) – Additional information for this result

Returns

The result object

Return type

Result

classmethod from_file(*storage*, *loc=None*, *, *model=None*)

load object from a file

This function loads the results from a hierachical storage. It also attempts to read information about the model that was used to create this result and additional data that might have been stored in a `storage` while the model was running.

Parameters

- **store** (str or `zarr.Store`) – Path or instance describing the storage, which is either a file path or a `zarr.Storage`.
- **loc** (*Location*) – The location where the result is stored in the storage. This should rarely be modified.
- **model** (ModelBase) – The model which lead to this result
- **storage** (*StorageID*) –

info: dict[str, Any] | None

Additional information for this result

Type

dict

model: *ModelBase*

Model that was run. This is a *MockModel* instance if details are not available

Type

ModelBase

property parameters: dict[str, Any]

result: Any

the final outcome of the model

storage: *StorageGroup* | None

Storage that might contain additional information, e.g., stored during the model run

Type
StorageGroup

to_file(*storage*, *loc=None*, *, *mode='insert'*)

write the results to a file

Note that this does only write the actual *results* but omits additional data that might have been stored in a storage that is associated with the results.

Parameters

- **storage** (StorageBase or StorageGroup) – The storage where the group is defined. If this is a StorageGroup itself, *loc* is interpreted relative to that group
- **loc** (*str or list of str*) – Denotes the location (path) of the group within the storage
- **mode** (str or ModeType) – The file mode with which the storage is accessed, which determines the allowed operations. Common options are "read", "full", "append", and "truncate".

Return type
None

class ResultCollection(*iterable=()*, */*)

Bases: List[*Result*]

represents a collection of results

as_dataframe(*, *enforce_same_model=True*)

create a pandas dataframe summarizing the data

Parameters
enforce_same_model (*bool*) – If True, forces all model results to derive from the same model

property constant_parameters: dict[str, Any]

the parameters that are constant in this result collection

Type
dict

property dataframe

create a pandas dataframe summarizing the data

filtered(***kwargs*)

return a subset of the results

Parameters
****kwargs** – Specify parameter values of results that are retained

Returns
The filtered collection

Return type
ResultColelction

classmethod from_folder(*folder*, *pattern='*.*'*, *model=None*, *, *strict=False*, *progress=False*)

create results collection from a folder

Parameters

- **folder** (*str*) – Path to the folder that is scanned
- **pattern** (*str*) – Filename pattern that is used to detect result files
- **model** (ModelBase) – Base class from which models are initialized
- **strict** (*bool*) – Whether to raise an exception or just emit a warning when a file cannot be read
- **progress** (*bool*) – Flag indicating whether a progress bar is shown

get(***kwargs*)

return a single result with the given parameters

Warning: If there are multiple results compatible with the specified parameters, only the first one is returned.

Parameters
****kwargs** – Specify parameter values of result that is returned

Returns
A single result from the collection

Return type
Result

groupby(**args*)

group results according to the given variables

Parameters
***args** – Specify parameters according to which the results are sorted

Returns
generator that allows iterating over the groups. Each iteration returns a dictionary with the current parameters and the associated *ResultCollection*.

Return type
Iterator[tuple[dict[str, list[Any]], *ResultCollection*]]

property parameters: dict[str, set[Any]]

the parameter values in this result collection

Note that parameters that are lists in the individual models are turned into tuples, so they can be handled efficiently, e.g., in sets.

Type
dict

remove_duplicates()

remove duplicates in the result collection

Return type
ResultCollection

property same_model: bool

flag determining whether all results are from the same model

Type
bool

sorted(**args*, *reverse=False*)

return a sorted version of the results

Parameters

- ***args** – Specify parameters according to which the results are sorted
- **reverse** (*bool*) – If True, sort in descending order

Returns
The filtered collection

Return type
ResultColelction

property varying_parameters: dict[str, list[Any]]

the parameters that vary in this result collection

Type
dict

1.4.3 modelrunner.storage package

Defines stroages, which contain store objects in a hierarchical format

open_storage	open a storage and return the root StorageGroup
Trajectory	Reads trajectories written with TrajectoryWriter
TrajectoryWriter	writes trajectories into a storage

Subpackages:

modelrunner.storage.backend package

class HDFStorage(*file_or_path*, ***, *mode='read'*, *compression=True*)

Bases: *StorageBase*

storage that stores data in an HDF file

Parameters

- **file_or_path** (str or Path or Store) – File path to the file/folder or a zarr Store
- **mode** (str or *AccessMode*) – The file mode with which the storage is accessed. Determines allowed operations.
- **compression** (*bool*) – Whether to store the data in compressed form. Automatically enabled chunked storage.

close()

closes the storage, potentially writing data to a persistent place

Return type
None

extensions: list[str] = ['h5', 'hdf', 'hdf5']

all file extensions supported by this storage

Type
list of str

`is_group`(*loc*)

determine whether the location is a group

Parameters
loc (*sequence of str*) – A list of strings determining the location in the storage

Returns
True if the loation is a group

Return type
bool

`keys`(*loc=None*)

return all sub-items defined at a given location

Parameters
loc (*sequence of str*) – A list of strings determining the location in the storage

Returns
a list of all items defined at this location

Return type
list

`mode: AccessMode`

access mode

Type
AccessMode

`class JSONStorage`(*path*, *, *mode='read'*, *simplify=True*, ***kwargs*)

Bases: *TextStorageBase*

storage that stores data in a JSON text file

Note that the data is only written once the storage is closed.

Parameters

- **path** (str or Path) – File path to the file
- **mode** (str or *AccessMode*) – The file mode with which the storage is accessed. Determines allowed operations.
- **simplify** (*bool*) – Flag indicating whether the data is stored in a simplified form

`extensions: list[str] = ['json']`

all file extensions supported by this storage

Type
list of str

`class MemoryStorage`(*, *mode='insert'*)

Bases: *StorageBase*

store items in memory

Parameters
mode (str or *AccessMode*) – The file mode with which the storage is accessed. Determines allowed operations.

`clear()`

truncate the storage by removing all stored data.

Parameters
clear_data_shape (*bool*) – Flag determining whether the data shape is also deleted.

Return type
None

`is_group(loc)`

determine whether the location is a group

Parameters
loc (*sequence of str*) – A list of strings determining the location in the storage

Returns
True if the loation is a group

Return type
bool

`keys(loc)`

return all sub-items defined at a given location

Parameters
loc (*sequence of str*) – A list of strings determining the location in the storage

Returns
a list of all items defined at this location

Return type
list

`class YAMLStorage(path, *, mode='read', simplify=True, **kwargs)`

Bases: *TextStorageBase*

storage that stores data in a YAML text file

Note that the data is only written once the storage is closed.

Parameters

- **path** (str or Path) – File path to the file
- **mode** (str or *AccessMode*) – The file mode with which the storage is accessed. Determines allowed operations.
- **simplify** (*bool*) – Flag indicating whether the data is stored in a simplified form

`encode_internal_attrs = True`

`extensions: list[str] = ['yaml', 'yml']`

all file extensions supported by this storage

Type
list of str

`class ZarrStorage(store_or_path, *, mode='read')`

Bases: *StorageBase*

storage that stores data in an zarr file or database

Parameters

- **store_or_path** (str or Path or Store) – File path to the file/folder or a zarr Store
- **mode** (str or *AccessMode*) – The file mode with which the storage is accessed. Determines allowed operations.

property can_update: bool

indicates whether the storage supports updating items

Type
bool

close()

closes the storage, potentially writing data to a persistent place

Return type
None

extensions: list[str] = ['zarr', 'zip', 'sqldb', 'lmdb']

all file extensions supported by this storage

Type
list of str

is_group(*loc*, *, *ignore_cls=False*)

determine whether the location is a group

Parameters

- **loc** (*sequence of str*) – A list of strings determining the location in the storage
- **ignore_cls** (*bool*) –

Returns
True if the loation is a group

Return type
bool

keys(*loc=None*)

return all sub-items defined at a given location

Parameters
loc (*sequence of str*) – A list of strings determining the location in the storage

Returns
a list of all items defined at this location

Return type
list

mode: *AccessMode*

access mode

Type
AccessMode

modelrunner.storage.backend.hdf module

Defines a class storing data on the file system using the hierarchical data format (hdf)

Requires the optional `h5py` module.

class HDFStorage(*file_or_path*, *, *mode='read'*, *compression=True*)

Bases: *StorageBase*

storage that stores data in an HDF file

Parameters

- **file_or_path** (str or Path or Store) – File path to the file/folder or a `zarr` Store
- **mode** (str or *AccessMode*) – The file mode with which the storage is accessed. Determines allowed operations.
- **compression** (*bool*) – Whether to store the data in compressed form. Automatically enabled chunked storage.

close()

closes the storage, potentially writing data to a persistent place

Return type
None

extensions: list[str] = ['h5', 'hdf', 'hdf5']

all file extensions supported by this storage

Type
list of str

is_group(*loc*)

determine whether the location is a group

Parameters
loc (*sequence of str*) – A list of strings determining the location in the storage

Returns
True if the loation is a group

Return type
bool

keys(*loc=None*)

return all sub-items defined at a given location

Parameters
loc (*sequence of str*) – A list of strings determining the location in the storage

Returns
a list of all items defined at this location

Return type
list

mode: *AccessMode*

access mode

Type
AccessMode

modelrunner.storage.backend.json module

Defines a class storing data in memory and writing it to a file in JSON format

class JSONStorage(*path*, *, *mode='read'*, *simplify=True*, ***kwargs*)

Bases: *TextStorageBase*

storage that stores data in a JSON text file

Note that the data is only written once the storage is closed.

Parameters

- **path** (str or Path) – File path to the file
- **mode** (str or *AccessMode*) – The file mode with which the storage is accessed. Determines allowed operations.
- **simplify** (*bool*) – Flag indicating whether the data is stored in a simplified form

extensions: list[str] = ['json']

all file extensions supported by this storage

Type
list of str

mode: *AccessMode*

access mode

Type
AccessMode

modelrunner.storage.backend.memory module

Defines a class storing data in memory.

class MemoryStorage(*, *mode='insert'*)

Bases: *StorageBase*

store items in memory

Parameters
mode (str or *AccessMode*) – The file mode with which the storage is accessed. Determines allowed operations.

clear()

truncate the storage by removing all stored data.

Parameters
clear_data_shape (*bool*) – Flag determining whether the data shape is also deleted.

Return type
None

is_group(*loc*)

determine whether the location is a group

Parameters
loc (*sequence of str*) – A list of strings determining the location in the storage

Returns
True if the loation is a group

Return type
bool

`keys`(*loc*)
return all sub-items defined at a given location

Parameters
loc (*sequence of str*) – A list of strings determining the location in the storage

Returns
a list of all items defined at this location

Return type
list

modelrunner.storage.backend.text_base module

`class TextStorageBase`(*path*, *, *mode='read'*, *simplify=True*, ***kwargs*)
Bases: *MemoryStorage*

base class for storage that stores data in a text file

Note that the data is only written once the storage is closed.

Parameters

- **path** (str or Path) – File path to the file
- **mode** (str or *AccessMode*) – The file mode with which the storage is accessed. Determines allowed operations.
- **simplify** (*bool*) – Flag indicating whether the data is stored in a simplified form

`close`()
close the file and write the data to the file

Return type
None

`flush`()
write (cached) data to storage

Return type
None

`to_text`(*simplify=None*)
serialize the data and return it as a string

Parameters
simplify (*bool*) – Flag indicating whether the data is stored in a simplified form. If *None*, the object-level value is used.

Return type
str

modelrunner.storage.backend.utils module

simplify_data(*data*)

simplify data (e.g. for writing to json or yaml)

This function for instance turns sets and numpy arrays into lists.

modelrunner.storage.backend.yaml module

Defines a class storing data in memory and writing it to a file in YAML format

Requires the optional `yaml` module.

class YAMLStorage(*path*, *, *mode='read'*, *simplify=True*, ***kwargs*)

Bases: *TextStorageBase*

storage that stores data in a YAML text file

Note that the data is only written once the storage is closed.

Parameters

- **path** (str or Path) – File path to the file
- **mode** (str or *AccessMode*) – The file mode with which the storage is accessed. Determines allowed operations.
- **simplify** (*bool*) – Flag indicating whether the data is stored in a simplified form

encode_internal_attrs = True

extensions: list[str] = ['yaml', 'yml']

all file extensions supported by this storage

Type
list of str

mode: *AccessMode*

access mode

Type
AccessMode

modelrunner.storage.backend.zarr module

Defines a class storing data in various storages

Requires the optional `zarr` module.

class ZarrStorage(*store_or_path*, *, *mode='read'*)

Bases: *StorageBase*

storage that stores data in an zarr file or database

Parameters

- **store_or_path** (str or Path or Store) – File path to the file/folder or a `zarr` Store
- **mode** (str or *AccessMode*) – The file mode with which the storage is accessed. Determines allowed operations.

property can_update: bool

indicates whether the storage supports updating items

Type
bool

close()

closes the storage, potentially writing data to a persistent place

Return type
None

extensions: list[str] = ['zarr', 'zip', 'sqldb', 'lmdb']

all file extensions supported by this storage

Type
list of str

is_group(*loc*, *, *ignore_cls=False*)

determine whether the location is a group

Parameters

- **loc** (*sequence of str*) – A list of strings determining the location in the storage
- **ignore_cls** (*bool*) –

Returns
True if the loation is a group

Return type
bool

keys(*loc=None*)

return all sub-items defined at a given location

Parameters
loc (*sequence of str*) – A list of strings determining the location in the storage

Returns
a list of all items defined at this location

Return type
list

mode: *AccessMode*

access mode

Type
AccessMode

modelrunner.storage.access_modes module

exception AccessError

Bases: RuntimeError

an error indicating that an access credential was not present

class AccessMode(*name*, *description*, *file_mode*, *read=False*, *set_attrs=False*, *insert=False*, *overwrite=False*, *dynamic_append=False*)

Bases: object

Determines access modes for storages

Parameters

- **name** (*str*) –
- **description** (*str*) –
- **file_mode** (*FileMode*) –
- **read** (*bool*) –
- **set_attrs** (*bool*) –
- **insert** (*bool*) –
- **overwrite** (*bool*) –
- **dynamic_append** (*bool*) –

description: str

dynamic_append: bool = False

file_mode: FileMode

insert: bool = False

name: str

overwrite: bool = False

classmethod parse(*obj_or_name*)

gets access mode from various formats

Parameters
obj_or_name (str or *AccessMode*) – An *AccessMode* object or the name of a registered access mode

Returns
the access mode object

Return type
AccessMode

read: bool = False

set_attrs: bool = False

modelrunner.storage.attributes module

decode_attrs(*attrs*)

decode many attributes

Parameters
attrs (*dict*) – The attributes dictionary

Returns
The decoded attributes

Return type
dict

encode_attrs(*attrs*)

encode many attributes

Parameters
attrs (*dict*) – The attributes dictionary

Returns
The encoded attributes

Return type
dict

modelrunner.storage.base module

Base classes for managing hierarchical storage in which data is stored

The storage classes provide low-level abstraction to store data in a hierarchical format and should thus not be used directly. Instead, the user typically interacts with *StorageGroup* objects, i.e., returned by *open_storage()*.

The role of *StorageBase* is to ensure access rights and provide an interface that can be specified easily by subclasses to provide new storage formats. In contrast, the interface of *StorageGroup* is more user-friendly and provides additional convenience methods.

The main structure of the storage is a hierarchical tree of *groups*, which can contain other groups or specific data items. Currently, items can be either arrays or arbitrary objects, which are serialized transparently. Moreover, each group and each item can have attributes, which are a mapping with string keys and arbitrary values, which are also serialized transparently. Note that keys with double underscores are reserved for internal use and should thus not be used.

class StorageBase(*, *mode='read'*)

Bases: object

base class for storing data

Parameters
mode (str or *AccessMode*) – The file mode with which the storage is accessed. Determines allowed operations.

property can_update: bool

indicates whether the storage supports updating items

Type
bool

close()

closes the storage, potentially writing data to a persistent place

Return type
None

property closed: bool

determines whether the storage has been closed

Type
bool

property codec: Codec

A codec used to encode binary data

Type
Codec

create_dynamic_array(*loc*, *shape*, *, *dtype=<class 'float'>*, *record_array=False*, *attrs=None*, *cls=None*)

creates a dynamic array of flexible size

Parameters

- **loc** (*list of str*) – The location in the storage where the dynamic array is created
- **shape** (*tuple of int*) – The shape of the individual arrays. A singular axis is prepended to the shape, which can then be extended subsequently.
- **dtype** (*DTypeLike*) – The data type of the array to be written
- **record_array** (*bool*) – Flag indicating whether the array is of type recarray
- **attrs** (*dict, optional*) – Attributes stored with the array
- **cls** (*type*) – A class associated with this array

Return type
None

create_group(*loc*, *, *attrs=None*, *cls=None*)

create a new group at a particular location

Parameters

- **loc** (*list of str*) – The location in the storage where the group will be created
- **attrs** (*dict, optional*) – Attributes stored with the group
- **cls** (*type*) – A class associated with this group. The class will be used to re-create the object when this group is later accessed directly.

Returns
The reference of the new group

Return type
StorageGroup

default_codec = Pickle(protocol=5)

the default codec used for encoding binary data

Type
numcodecs.Codec

ensure_group(*loc*)

ensures the a group exists in the storage

If the group is not already in the storage, it is created (recursively).

Parameters
loc (*list of str*) – The group location in the storage

Return type
None

extend_dynamic_array(*loc*, *arr*)

extend a dynamic array previously created

Parameters

- **loc** (*list of str*) – The location in the storage where the dynamic array is located
- **arr** (*array*) – The array that will be appended to the dynamic array

Return type
None

extensions: list[str] = []

all file extensions supported by this storage

Type
list of str

flush()

write (cached) data to storage

Return type
None

abstract is_group(*loc*)

determine whether the location is a group

Parameters
loc (*sequence of str*) – A list of strings determining the location in the storage

Returns
True if the loation is a group

Return type
bool

abstract keys(*loc*)

return all sub-items defined at a given location

Parameters
loc (*sequence of str*) – A list of strings determining the location in the storage

Returns
a list of all items defined at this location

Return type
list

mode: *AccessMode*

access mode

Type
AccessMode

read_array(*loc*, *, *out=None*, *index=None*)

read an array from a particular location

Parameters

- **loc** (*list of str*) – The location in the storage where the array is read
- **out** (*array*) – An array to which the results are written
- **index** (*int, optional*) – An index denoting the subarray that will be read

Returns
An array containing the data. Identical to *out* if specified.

Return type
ndarray

read_attrs(*loc*)

read attributes associated with a particular location

Parameters
loc (*list of str*) – The location in the storage where the attributes are read

Returns
A copy of the attributes at this location

Return type
dict

read_object(*loc*)

read an object from a particular location

Parameters
loc (*list of str*) – The location in the storage where the object is created

Returns
The object that has been read from the storage

Return type
Any

write_array(*loc*, *arr*, *, *attrs=None*, *cls=None*)

write an array to a particular location

Parameters

- **loc** (*list of str*) – The location in the storage where the array is read
- **arr** (ndarray) – The array that will be written
- **attrs** (*dict, optional*) – Attributes stored with the array
- **cls** (*type*) – A class associated with this array. The class will be used to re-create the object when this array is later accessed. If no class is supplied, a generic *~modelrunner.storage.utils.Array* will be returned.

Return type
None

write_attrs(*loc*, *attrs*)

write attributes to a particular location

Parameters

- **loc** (*list of str*) – The location in the storage where the attributes are written
- **attrs** (*dict*) – The attributes to be added to this location

Return type
None

`write_object`(*loc*, *obj*, *, *attrs=None*, *cls=None*)

write an object to a particular location

Parameters

- **loc** (*list of str*) – The location in the storage where the object is read.
- **obj** (*Any*) – The object that will be written
- **attrs** (*dict, optional*) – Attributes stored with the object
- **cls** (*type*) – A class associated with this object. The class will be used to re-create the object when this object is later accessed. If no class is supplied, a generic python object will be returned.

Return type
None

modelrunner.storage.group module

`class StorageGroup`(*storage*, *loc=None*)

Bases: `object`

refers to a group within a storage

Parameters

- **storage** (StorageBase or *StorageGroup*) – The storage where the group is defined. If this is a *StorageGroup* itself, *loc* is interpreted relative to that group
- **loc** (*str or list of str*) – Denotes the location (path) of the group within the storage

`property attrs: Dict[str, Any]`

the attributes associated with this group

Type
dict

`create_dynamic_array`(*loc*, *, *arr=None*, *shape=None*, *dtype=<class 'float'>*, *record_array=False*, *attrs=None*, *cls=None*)

creates a dynamic array of flexible size

Parameters

- **loc** (*str or list of str*) – The location where the dynamic array is created
- **shape** (*tuple of int*) – The shape of the individual arrays. A singular axis is prepended to the shape, which can then be extended subsequently.
- **dtype** (*DTypeLike*) – The data type of the array to be written
- **record_array** (*bool*) – Flag indicating whether the array is of type `recarray`
- **attrs** (*dict, optional*) – Attributes stored with the array
- **cls** (*type*) – A class associated with this array
- **arr** (*np.ndarray | None*) –

create_group(*loc*, ***, *attrs=None*, *cls=None*)

create a new group at a particular location

Parameters

- **loc** (*str or list of str*) – The location where the group will be created
- **attrs** (*dict, optional*) – Attributes stored with the group
- **cls** (*type*) – A class associated with this group

Returns
The reference of the new group

Return type
StorageGroup

extend_dynamic_array(*loc*, *data*)

extend a dynamic array previously created

Parameters

- **loc** (*str or list of str*) – The location where the dynamic array is located
- **arr** (*array*) – The array that will be appended to the dynamic array
- **data** (*_SupportsArray[dtype] | _NestedSequence[_SupportsArray[dtype]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) –

get(*loc*, *default=None*)

Parameters

- **loc** (*None | str | Sequence[Location]*) –
- **default** (*Any | None*) –

Return type
Any

get_class(*loc=None*)

get the class associated with a particular location

Class information can be written using the *cls* attribute of *write_array*, *write_object*, and similar functions.

Parameters
loc (*str or list of str*) – The location where the class information is read from

Return type
type | None

Retruns: the class associated with the lcoation

is_group(*loc=None*)

determine whether the location is a group

Parameters
loc (*sequence of str*) – A list of strings determining the location in the storage

Returns
True if the loation is a group

Return type
bool

items()

iterate over stored items, yielding the location and item of each

Return type
Iterator[tuple[str, Any]]

keys()

return name of all stored items in this group

Return type
Collection[str]

open_group(*loc*)

open an existing group at a particular location

Parameters
loc (*str or list of str*) – The location where the group will be opened

Returns
The reference to the group

Return type
StorageGroup

property parent: *StorageGroup*

Parent group

Raises
RuntimeError – If current group is root group

Type
StorageGroup

read_array(*loc*, *, *out=None*, *index=None*)

read an array from a particular location

Parameters

- **loc** (*str or list of str*) – The location where the array is created
- **out** (*array, optional*) – An array to which the results are written
- **index** (*int, optional*) – An index denoting the subarray that will be read

Returns
An array containing the data. Identical to *out* if specified.

Return type
ndarray

read_attrs(*loc=None*)

read attributes associated with a particular location

Parameters
loc (*str or list of str*) – The location in the storage where the attributes are read

Returns
A copy of the attributes at this location

Return type
dict

read_item(*loc*, *, *use_class=True*)

read an item from a particular location

Parameters

- **loc** (*str or list of str*) – The location where the item is read from
- **use_class** (*bool*) – If *True*, looks for class information in the attributes and evokes a potentially registered hook to instantiate the associated object. If *False*, only the current data or object is returned.

Returns

The reconstructed python object

Return type

Any

read_object(*loc*)

read an object from a particular location

Parameters

loc (*str or list of str*) – The location where the object is created

Returns

The object that has been read from the storage

Return type

Any

tree()

print the hierarchical storage as a tree structure

Return type

None

write_array(*loc*, *arr*, *, *attrs=None*, *cls=None*)

write an array to a particular location

Parameters

- **loc** (*str or list of str*) – The location where the array is read
- **arr** (ndarray) – The array that will be written
- **attrs** (*dict, optional*) – Attributes stored with the array
- **cls** (*type*) – A class associated with this array

write_attrs(*loc=None*, *attrs=None*)

write attributes to a particular location

Parameters

- **loc** (*str or list of str*) – The location in the storage where the attributes are written
- **attrs** (*dict*) – The attributes to be added to this location

Return type

None

write_item(*loc*, *item*, *, *attrs=None*, *use_class=True*)

write an item to a particular location

Parameters

- **loc** (*sequence of str*) – The location where the item is written to
- **item** (*Any*) – The item that will be written
- **attrs** (*dict, optional*) – Attributes stored with the object
- **use_class** (*bool*) – If *True*, looks for class information in the attributes and evokes a potentially registered hook to instantiate the associated object. If *False*, only the current data or object is returned.

Return type
None

write_object(*loc*, *obj*, *, *attrs=None*, *cls=None*)

write an object to a particular location

Parameters

- **loc** (*str or list of str*) – The location where the object is read
- **obj** (*Any*) – The object that will be written
- **attrs** (*dict, optional*) – Attributes stored with the object
- **cls** (*type*) – A class associated with this object

modelrunner.storage.tools module

Functions that provide convenience on top of the storage classes

class open_storage(*storage=None*, *, *loc=None*, ***kwargs*)

Bases: *StorageGroup*

open a storage and return the root `StorageGroup`

Example

This can be either used like a function

```
storage = open_storage(...)
# use the storage
storage.close()
```

or as a context manager

```
with open_storage(...) as storage:
    # use the storage
```

Parameters

- **storage** (*StorageID*) – The path to a file or directory or a `StorageBase` instance. The special value *None* creates a *MemoryStorage*
- **loc** (*str or list of str*) – Denotes the location that will be opened within the storage. The default *None* opens the root group of the storage.
- **mode** (str or `ModeType`) – The file mode with which the storage is accessed, which determines the allowed operations. Common options are “read”, “full”, “append”, and “truncate”.
- ****kwargs** – All other arguments are passed on to the storage class

close()

close the storage (and flush all data to persistent storage if necessary)

Return type
None

property closed: bool

determines whether the storage group has been closed

Type
bool

property mode: *AccessMode*

access mode

Type
AccessMode

modelrunner.storage.trajectory module

Classes that describe time-dependent data, i.e., trajectories.

TrajectoryWriter	writes trajectories into a storage
Trajectory	Reads trajectories written with *TrajectoryWriter*

class Trajectory(*storage*, *loc='trajectory'*)

Bases: object

Reads trajectories written with *TrajectoryWriter*

The class permits direct access to indivdual items in the trajcetory using the square bracket notation. It is also possible to iterate over all items.

times

Time points at which data is available

Type
ndarray

Parameters

- **storage** (*MutableMapping or string*) – Store or path to directory in file system or name of zip file.
- **loc** (*str or list of str*) – The location in the storage where the trajectory data is read.

close()

close the openend storage

Return type
None

class TrajectoryWriter(*storage*, *loc='trajectory'*, ***, *attrs=None*, *mode=None*)

Bases: object

writes trajectories into a storage

Stored data can then be read using *Trajectory*.

Example

```
# write data using context manager
with TrajectoryWriter("test.zarr") as writer:
    for t, data in simulation:
        writer.append(data, t)

# append to same file using explicit class interface
writer = TrajectoryWriter("test.zarr", mode="append")
writer.append(data0)
writer.append(data1)
writer.close()

# read data
trajectory = Trajectory("test.zarr")
assert trajectory[-1] == data1
```

Parameters

- **store** – Store or path to directory in file system or name of zip file.
- **loc** (*str or list of str*) – The location in the storage where the trajectory data is written.
- **attrs** (*dict*) – Additional attributes stored in the trajectory.
- **mode** (str or *AccessMode*) – The file mode with which the storage is accessed. Determines allowed operations. The meaning of the special (default) value *None* depends on whether the file given by *store* already exists. If yes, a RuntimeError is raised, otherwise the choice corresponds to *mode="full"* and thus creates a new trajectory. If the file exists, use *mode="truncate"* to overwrite file or *mode="append"* to insert new data into the file.
- **storage** (*str | Path | StorageGroup | StorageBase*) –

append(*data*, *time=None*)

append data to the trajectory

Parameters

- **data** (*Any*) – The data to append to the trajectory
- **time** (*float, optional*) – The associated time point. If omitted, the last time point is incremented by one.

Return type

None

close()

property times: ndarray

Time points written so far

Type

ndarray

modelrunner.storage.utils module

Functions and classes that are used commonly used by the storage classes.

class Array(*input_array*, *attrs=None*)

Bases: ndarray

Numpy array augmented with attributes

Parameters
attrs (*Attrs | None*) –

decode_binary(*obj_str*)

decode an object encoded with *encode_binary()*.

Parameters
obj_str (*str or bytes*) – The string that encodes the object

Returns
the object

Return type
Any

decode_class(*class_path*, *, *guess=None*)

decode a class encoded with *encode_class()*.

Parameters

- **class_path** (*str*) – The string that encodes the class
- **guess** (*type*) – A class that is used if the encoded class cannot be found and the name of the guess matches the encoded class.

Returns
the class or *None* if class_path was None

Return type
type

encode_binary(*obj: Any*, *, *binary: Literal[True]*) → bytes

encode_binary(*obj: Any*, *, *binary: Literal[False]*) → str

encodes an arbitrary object as a string

The object can be decoded using *decode_binary()*.

Parameters

- **obj** – The object to encode
- **binary** (*bool*) – Encode as a byte array if *True*. Otherwise, a unicode string is returned

Returns
The encoded object

Return type
str or bytes

encode_class(*cls*)

encode a class such that it can be restored

The class can be decoded using *decode_class()*.

Parameters
cls (*type*) – The class

Returns
the encoded class

Return type
str

1.4.4 modelrunner.config module

Handles configuration variables

Config	class handling the package configuration

class Config(*default=None*, *mode='update'*, ***, *check_validity=True*, *include_deprecated=True*)

Bases: UserDict

class handling the package configuration

Parameters

- **default** (sequence of Parameter, optional) – Default configuration values. The default configuration also defines what parameters can typically be defined and it provides additional information for these parameters.
- **mode** (*str*) – Defines the mode in which the configuration is used. Possible values are
  - *insert*: any new configuration key can be inserted
  - *update*: only the values defined by *default* can be updated
  - *locked*: no values can be changed

  Note that the items specified by *items* will always be inserted, independent of the *mode*.
- **check_validity** (*bool*) – Determines whether a *ValueError* is raised if there are keys in parameters that are not in the defaults. If *False*, additional items are simply stored in *self.parameters*
- **include_deprecated** (*bool*) – Include deprecated parameters

copy()

return a copy of the configuration

Return type
Config

load(*path*)

load configuration from yaml file

Parameters
path (*str | Path*) –

save(*path*)

save configuration to yaml file

Parameters
path (*str | Path*) –

to_dict()

convert the configuration to a simple dictionary

Returns
A representation of the configuration in a normal dict.

Return type
dict

1.4.5 modelrunner.utils module

Miscellaneous utility methods

hybridmethod	decorator to use a method both as a classmethod and an instance method
import_class	import a class or module given an identifier

class hybridmethod(*fclass*, *finstance=None*, *doc=None*)

Bases: object

decorator to use a method both as a classmethod and an instance method

---

Note: The decorator can be used like so:

```
@hybridmethod
def method(cls, ...): ...

@method.instancemethod
def method(self, ...): ...
```

Adapted from https://stackoverflow.com/a/28238047

classmethod(*fclass*)

instancemethod(*finstance*)

import_class(*identifier*)

import a class or module given an identifier

Parameters
identifier (*str*) – The identifier can be a module or a class. For instance, calling the function with the string *identifier == 'numpy.linalg.norm'* is roughly equivalent to running *from numpy.linalg import norm* and would return a reference to *norm*.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

F

G

H

I

J

K

L

M

N

O

P