so-magic

Release 0.7.1

Konstantinos Lampridis

Jun 24, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

This is SO MaGIc - Self-Organising Map Graceful Interface.
A python library providing an elegant interface (API) to infer models of the Self-Organing Map family, based on structured data.

This documentation aims to help people understand what are the features of the library and how they can use it. It presents some use cases and an overview of the library capabilities and overall design.

CHAPTER

TWO

WHY THIS LIBRARY?

So Magic main capability is to infer Self-Organising Map models out of structured data.

Practically speaking there is a lot of effort to converting raw data into the “structured format”, that is ready to be digested by the “learning” algorithm. So Magic facilitates this process (aka pre-processing), providing with pre-built commands and also supporting user-made commands for performing common operations on “raw” data.

Apart from the ability to train SOM models (using the “learning” algorithm) So Magic provides with hyper-parameter tuning given intrinsic and user-made extrinsic evaluation criteria.

Another feature is the easy way to persist models parameters and/or model evaluation statistics. The library provides serialization and deserialization supporting JSON format.

CHAPTER

THREE

INSTALLATION

So Magic was desinged with the intention to be pip installable. Unfortunately,
even though $pip install so-magic$ will install the library from the pypi (index) server,
it is possible that the somoclu library (which is a dependency) will be missing a critical feature.

If you have successfully (build and) installed somoclu in your environment
then you can indeed install so-magic from pypi.

Example command:

```
pip install so-magic
```

See the somoclu documentation pages to read more about the library features.

3.1 Linux & macOS

Unfortunately, it is possible that none of the two official installation methods for somoclu would work as expected. Namely the method of invoking `pip install somoclu` and the method of building somoclu from source.

It has been documented (see issue 136), and verified by the author on a macOS Catalina, that you might run into an issue with clang and OpenMP that breaks the build process. It is also possible that installation succeeds, but when your client code invokes the "wrap_train" function, a NameError exception (see issue 28) is thrown on the Python side.

Thus, it is recommended that you install so-magic using anaconda. Anaconda succeeds in doing conda install somoclu in contrast to pip

Prepare a conda environment (just like a virtualenv):

```
ENV_PATH=so-magic-env
conda create -p $ENV_PATH -y python=3.7
FILE_TO_SOURCE="$HOME/miniconda/etc/profile.d/conda.sh"
if [[ ! -f $FILE_TO_SOURCE ]]; then
  source "$HOME/miniconda3/etc/profile.d/conda.sh"
else
  source "$HOME/miniconda/etc/profile.d/conda.sh"
fi
hash -r
```

(continues on next page)

(continued from previous page)

```
conda config --set always_yes yes --set changeps1 no
conda update -q conda
conda info -a
conda activate $ENV_PATH
```

Install somoclu dependency:

```
conda install somoclu --channel conda-forge
```

Install latest so-magic:

```
pip install so-magic
```

CHAPTER

FOUR

QUICKSTART

The most common case is to infer a self-organising map given some data.
Assuming you have a file with data in JSON-lines format, then you could "train" a model uising the following code:

```
# Please re-configure
my_json_lines_file_path = 'path-to-json-lines-data'


from so_magic import init_so_magic
somagic = init_so_magic()

somagic.load_data(my_json_lines_file_path, id='test_data')
ATTRS = ['hybrid', 'indica', 'sativa']
ATTRS2 = ['type_hybrid', 'type_indica', 'type_sativa']
from functools import reduce
UNIQUE_FLAVORS = reduce(lambda i, j: set(i).union(set(j)),
                        [_ for _ in somagic._data_manager.datapoints.observations[
↪'flavors'] if _ is not None])

if not getattr(somagic.dataset, 'feature_vectors', None):
    cmd = somagic._data_manager.command.select_variables
    cmd.args = [[{'variable': 'type', 'columns': ATTRS2}, {'variable': 'flavors',
↪'columns': list(UNIQUE_FLAVORS)}]]
    cmd.execute()

    cmd = somagic._data_manager.command.one_hot_encoding
    cmd.args = [somagic._data_manager.datapoints, 'type']
    cmd.execute()

    # cmd2
    cmd = somagic._data_manager.command.one_hot_encoding_list
    cmd.args = [somagic._data_manager.datapoints, 'flavors']
    cmd.execute()

import numpy as np
setattr(somagic.dataset, 'feature_vectors', np.array(somagic._data_manager.datapoints.
↪observations[ATTRS2 + list(UNIQUE_FLAVORS)]))
# somagic.dataset.feature_vectors = np.array(somagic._data_manager.datapoints.
↪observations[ATTRS2 + list(UNIQUE_FLAVORS)])
```

(continues on next page)

(continued from previous page)

```
print("ID", id(somagic.dataset))

som = somagic.map.train(*train_args[:2], maptype=train_args[2], gridtype=train_args[3])
```

CHAPTER

FIVE

SO_MAGIC

5.1 so_magic package

5.1.1 Subpackages

so_magic.data package

Subpackages

so_magic.data.backend package

Subpackages

so_magic.data.backend.panda_handling package

Submodules

so_magic.data.backend.panda_handling.df_backend module

class so_magic.data.backend.panda_handling.df_backend.**Delegate**(*tabular_operator*)
Bases: object

class so_magic.data.backend.panda_handling.df_backend.**EngineBackends**(*backend_interfaces*)
Bases: object

add(**backend_implementations*)

define_operator(*backend_id*, *operator_type: str*)

property defined_backend_names

property defined_interfaces

static from_initial_available(*backends*)

name(*interface_implementation*)

register(*backend_implementation: dict*)

so_magic.data.backend.panda_handling.df_backend.**magic_backends**()

so_magic.data.backend.panda_handling.df_backend.**with_self**(*function*)

Module contents

Submodules

so_magic.data.backend.backend module

This module defines a way to create Data Engines and to register new commands that a Data Engine can execute.

class so_magic.data.backend.backend.**BackendType**(**args*, ***kwargs*)
Bases: *so_magic.data.backend.backend.CommandRegistrator*

Tabular Data Backend type representation.

Classes using this class as metaclass gain certain class attributes such as attributes related to tabular data operations (retriever, iterator, mutator) and attributes related to constructing command object prototypes (command_factory attribute).

dec(*data_structure='tabular-data'*) → Callable[[Callable], Callable]
Register a new command that executes the business logic supplied at runtime.

Decorate a function so that its body acts as the business logic that runs as part of a Command. The name of the function can be used to later reference the Command (or a prototype object of the Command).

Using the 'observations' name for your function will register a command that upon execution creates a new instance of Datapoints (see Datapoints class), provided that the runtime function returns an object that acts as the 'observations' attribute of a Datapoints object.

Parameters **data_structure** (*str, optional*) – useful when the function name is 'observations'. Defaults to 'tabular-data'.

class so_magic.data.backend.backend.**CommandRegistrator**(**args*, ***kwargs*)
Bases: *so_magic.data.backend.backend.MyDecorator*

Classes can use this class as metaclass to obtain a single registration point accessible as class attribute.

func_decorator()

class so_magic.data.backend.backend.**EngineBackend**
Bases: object

Facility to create Data Engines.

command_factory = MagicCommandFactory(command_factory=<so_magic.data.backend.engine_command_factory.CommandFactory object>, subject=<so_magic.utils.notification.Subject object>)

datapoints_factory = None

iterator = None

mutator = None

classmethod new(*engine_name: str*) → *so_magic.data.backend.backend.BackendType*
Create a Data Engine object and register it under the given name, to be able to reference it by name.

Creates a Data Engine that serves as an empty canvas to add attributes and Commands.

Parameters **engine_name** (*str*) – the name under which to register the Data Engine

Returns the Data Engine object

Return type *BackendType*

classmethod register_as_subclass(*backend_type: str*)
Indicate that a class is a subclass of DataEngine and register it under the given name.

It also sets the engine_type attribute on the decorate class to be equal to the subclass.

Parameters **backend_type** (*str*) – the name under which to register the Data Engine

registry = {}

retriever = None

state = None

subclasses = {}

class so_magic.data.backend.backend.**MyDecorator**
Bases: type

Metaclass that provides a decorator able to be invoked both with and without parenthesis. The wrapper function logic should be implemented by the client code.

classmethod magic_decorator(*arg=None*)

so_magic.data.backend.engine module

Define a wrapper around an Engine as the Backend class which constructor can initialize Backend instances.

class so_magic.data.backend.engine.**Engine**(*backend_instance*, *backends=NOTHING*)
Bases: object

Wrapper of a data engine, a datapoints manager and a datapoints factory.

Instances of this class act as data placeholders (aka data classes) and take at runtime a data engine (eg a set of pandas-dependent implementations of the "Tabular Data interfaces" defined in so_magic.data.interfaces).

Parameters **engine_instance** (*DataEngine*) – a data engine represented as a class object (eg class MyClass: pass)

property backend
The Data Engine instance, that this object wraps around.

Returns the Data Engine instance object

Return type DataEngine

static from_backend(*backend_id: str*)

so_magic.data.backend.engine_command_factory module

class so_magic.data.backend.engine_command_factory.**BaseCommandFactory**
Bases: object

subclasses = {'function': <class 'so_magic.data.backend.engine_command_factory.FunctionCommandFactory'>, 'generic': <class 'so_magic.data.backend.engine_command_factory.GenericCommandFactory'>}

class so_magic.data.backend.engine_command_factory.**CommandFactory**
Bases: object

A factory class able to construct new command objects.

constructors = {'function': <bound method FunctionCommandFactory.construct of <so_magic.data.backend.engine_command_factory.FunctionCommandFactory object>>, 'generic': <bound method GenericCommandFactory.construct of <so_magic.data.backend.engine_command_factory.GenericCommandFactory object>>}

classmethod create(**args*, ***kwargs*) → Tuple[*so_magic.utils.commands.Command*, str]

classmethod pick(**args*, ***kwargs*)

class so_magic.data.backend.engine_command_factory.**FunctionCommandFactory**

Bases: so_magic.utils.command_factory_interface.CommandFactoryInterface

Command Factory that constructs a command assuming the 1st argument is a python function.

Assumes that the function (1st argument) acts as the the 'receiver' (see Command module), 2nd is the method to call on the receiver and the rest are the method's runtime arguments.

construct(**args*, ***kwargs*) → *so_magic.utils.commands.Command*

Construct a command object (Command class instance).

Assumes that the 1st argument is a python function and that it acts as the the 'receiver' (see Command module). The rest are the function's runtime arguments.

Raises **RuntimeError** – [description]

Returns [description]

Return type *Command*

class so_magic.data.backend.engine_command_factory.**GenericCommandFactory**

Bases: so_magic.utils.command_factory_interface.CommandFactoryInterface

Command Factory that constructs a generic command given all the necessary arguments.

Assumes the 1st argument is the 'receiver' (see Command module), 2nd is the method to call on the receiver and the rest are the method's runtime arguments.

construct(**args*, ***kwargs*) → *so_magic.utils.commands.Command*

Construct a command object (Command class instance).

Assumes the 1st argument is the 'receiver' (see Command module), 2nd is the method to call on the receiver and the rest are the method's runtime arguments.

Returns the command object

Return type *Command*

class so_magic.data.backend.engine_command_factory.**MagicCommandFactory**(*command_factory=<so_magic.data.backe object>*)

Bases: object

Instances of this class act as callable command factories that notify, subscribed observers/listeners upon new command object creation.

Parameters **command_factory** (CommandFactory, *optional*) – an instance of a CommandFActory

subject: *so_magic.utils.notification.Subject*

so_magic.data.backend.backend_specs module

class so_magic.data.backend.backend_specs.**BackendSpecifications**(*name_abbreviation*, *name=NOTHING*)

Bases: object

classmethod from_dict(*a_dict*)

class so_magic.data.backend.backend_specs.**EngineTabularIterator**

Bases: object

subclasses = {}

class so_magic.data.backend.backend_specs.**EngineTabularMutator**

Bases: object

subclasses = {}

class so_magic.data.backend.backend_specs.**EngineTabularRetriever**

Bases: object

subclasses = {}

Module contents

so_magic.data.backend.**init_engine**(*engine_type='pd'*)

so_magic.data.datapoints package

Submodules

so_magic.data.datapoints.datapoints module

class so_magic.data.datapoints.datapoints.**AbstractTabularData**(*observations*, *attributes*)

Bases: *so_magic.data.datapoints.datapoints.StructuredData*, *so_magic.data.datapoints.tabular_data_interface.TabularDataInterface*, abc.ABC

Tabular Data with known attributes of interest.

Classes inheriting from this abstract class, gain both capabilities of structured data in terms of their attributes and capabilities of a data table in terms of column, rows, etc.

exception so_magic.data.datapoints.datapoints.**DatapointsCreationError**(*msg*)

Bases: Exception

class so_magic.data.datapoints.datapoints.**DatapointsFactory**

Bases: object

Factory to construct Datapoints objects.

A class that registers objects (constructors), which can be "called" to return (create) an object that implements the DatapointsInterface interface.

Also, exposes the 'create' factory method that given runtime arguments, returns an object that implements the DatapointsInterface interface by delegating the creation process to one of the registered constructors.

constructors = {'structured-data': <class 'so_magic.data.datapoints.datapoints.StructuredData'>, 'tabular-data': <class 'so_magic.data.datapoints.datapoints.TabularData'>}

classmethod create(*name*, **args*, ***kwargs*) → Iterable
Create a Datapoints instance by using a registered "constructor".

Parameters **name** (*str*) – the registered name of the "constructor" to use

Raises

- **KeyError** – happens if the input name is not found in the registry
- *DatapointsCreationError* – in case the object instantiation operation fails

Returns instance implementing the DatapointsInterface

Return type Iterable

classmethod register_constructor(*name: str*)
Register, using a unique name, an object as a "runnable" constructor.

A decorator method that should decorate a callable" The callable should return (create) an object that implements the DatapointsInterface interface.

Parameters **name** (*str*) – the name under which to register the "constructor"

class so_magic.data.datapoints.datapoints.**DatapointsInterface**
Bases: abc.ABC

Represent multiple data points out of a collection of data.

Classes implementing this interface, provide to their object instances (eg objects created using the classes constructor method) the 'observations' property.

The 'observations' property should hold the information about the datapoints.

abstract property observations: Iterable
The collection of datapoints is referenced through this property.

class so_magic.data.datapoints.datapoints.**StructuredData**(*observations*, *attributes*)
Bases: *so_magic.data.datapoints.datapoints.DatapointsInterface*, *so_magic.data.datapoints.datapoints.StructuredDataInterface*

Structured data. There are specific attributes/variables per observation.

Instances of this class represent collections of data (multiple data points aka observations). Each data point is expected to hold information about the specified attributes and that is why we are dealing with structured data/information in contrast to ie image data or sound data.

Parameters

- **observations** (*object*) – a reference to the actual datapoints object
- **attributes** (*object*) – a reference to the attributes object

property attributes
The set of attributes is referenced through this property.

property observations
The collection of datapoints is referenced through this property.

class so_magic.data.datapoints.datapoints.**StructuredDataInterface**
Bases: abc.ABC

Data points that are expected to have a specific set of attributes.

Classes implementing this interface, provide to their object instances (eg objects created using the classes constructor method) the 'attributes' property.

The 'attributes' property should hold the information about the attributes, that each data point (observation) is expected to have.

abstract property attributes: Iterable
The set of attributes is referenced through this property.

class so_magic.data.datapoints.datapoints.**TabularData**(*observations*, *attributes*, *retriever*, *iterator*, *mutator*)

Bases: *so_magic.data.datapoints.datapoints.AbstractTabularData*

Table-like datapoints that are loaded in memory

add_column(*values*, *column_name*, ***kwargs*)

property attributes
The set of attributes is referenced through this property.

column(*identifier*)
Get the data inside a column of the table.

Parameters **identifier** (*Union[str, int]*) – a primitive identifier to distinguish between the columns

Returns the data contained in the table's requested column

Return type Iterable

property columns: Iterable
List of the column identifiers.

get_categorical_attributes()

get_numerical_attributes()

itercolumns()
Iterate over the table's columns.

iterrows()
Iterate over the table's rows.

property nb_columns
The number of the table's columns.

Returns the number of columns

Return type int

property nb_rows
The number of the table's rows.

Returns the number of rows

Return type int

row(*identifier*)
Get the data inside a row of the table.

Parameters **identifier** (*Union[str, int]*) – a primitive identifier to distinguish between the rows

Returns the data contained in the table's requested row

Return type Iterable

property rows: Iterable
List of the row identifiers.

so_magic.data.datapoints.tabular_data_interface module

This module defines the TabularDataInterface interface.

class so_magic.data.datapoints.tabular_data_interface.**TabularDataInterface**
Bases: abc.ABC

Data points that have tabular structure and are loaded in memory.

Classes implementing this interface represent Data points that can be represented as a table of rows an columns. One can imagine that each row (or column) represents a single observation (single data point) and each column (or row) one single attribute out of possibly many attributes.

Classes implementing this interface have the ability to report on various elements and properties (eg rows, columns) of the underlying table-like data-structure.

abstract column(*identifier: Union[str, int]*) → Iterable
Get the data inside a column of the table.

Parameters **identifier** (*Union[str, int]*) – a primitive identifier to distinguish between the columns

Returns the data contained in the table's requested column

Return type Iterable

abstract property columns: Iterable
List of the column identifiers.

abstract itercolumns() → Iterable
Iterate over the table's columns.

abstract iterrows() → Iterable
Iterate over the table's rows.

abstract property nb_columns: int
The number of the table's columns.

Returns the number of columns

Return type int

abstract property nb_rows: int
The number of the table's rows.

Returns the number of rows

Return type int

abstract row(*identifier: Union[str, int]*) → Iterable
Get the data inside a row of the table.

Parameters **identifier** (*Union[str, int]*) – a primitive identifier to distinguish between the rows

Returns the data contained in the table's requested row

Return type Iterable

abstract property rows: Iterable
List of the row identifiers.

Module contents

class so_magic.data.datapoints.**DatapointsFactory**

Bases: object

Factory to construct Datapoints objects.

A class that registers objects (constructors), which can be "called" to return (create) an object that implements the DatapointsInterface interface.

Also, exposes the 'create' factory method that given runtime arguments, returns an object that implements the DatapointsInterface interface by delegating the creation process to one of the registered constructors.

constructors = {'structured-data': <class 'so_magic.data.datapoints.datapoints.StructuredData'>, 'tabular-data': <class 'so_magic.data.datapoints.datapoints.TabularData'>}

classmethod create(*name*, **args*, ***kwargs*) → Iterable

Create a Datapoints instance by using a registered "constructor".

Parameters **name** (*str*) – the registered name of the "constructor" to use

Raises

- **KeyError** – happens if the input name is not found in the registry
- *DatapointsCreationError* – in case the object instantiation operation fails

Returns instance implementing the DatapointsInterface

Return type Iterable

classmethod register_constructor(*name: str*)

Register, using a unique name, an object as a "runnable" constructor.

A decorator method that should decorate a callable" The callable should return (create) an object that implements the DatapointsInterface interface.

Parameters **name** (*str*) – the name under which to register the "constructor"

so_magic.data.features package

Submodules

so_magic.data.features.features module

class so_magic.data.features.features.**AttributeReporter**(*label*, *reporter=<so_magic.data.features.features.BaseAttributeReport object>*)

Bases: object

value_set(*datapoints*)

values(*datapoints*)

A default implementation of the values method

variable_type(*datapoints*)

A default implementation of the values method

class so_magic.data.features.features.**AttributeReporterInterface**
Bases: abc.ABC

A class implementing this interface has the ability to report information on an attribute/variable of some structured data (observations)

abstract value_set(*datapoints*, *attribute*, ***kwargs*)

abstract values(*datapoints*, *attribute*, ***kwargs*)
Get the values ([N x 1] vector) of all datapoints (N x D) corresponding to the input variable/attribute.

Parameters

- **datapoints** (*Datapoints*) – [description]
- **attribute** (*str*) – [description]

Returns the values in a [N x 1] vector

Return type (numpy.ndarray)

abstract variable_type(*datapoints*, *attribute*, ***kwargs*)
Call to get the variable type of the datapoints, given the attribute.

Parameters

- **datapoints** (*Datapoints*) – [description]
- **attribute** (*str*) – [description]

Returns [description]

Return type (str)

class so_magic.data.features.features.**BaseAttributeReporter**
Bases: *so_magic.data.features.features.AttributeReporterInterface*

value_set(*datapoints*, *attribute*, ***kwargs*)

values(*datapoints*, *attribute*, ***kwargs*)
Get the values ([N x 1] vector) of all datapoints (N x D) corresponding to the input variable/attribute.

Parameters

- **datapoints** (*Datapoints*) – [description]
- **attribute** (*str*) – [description]

Returns the values in a [N x 1] vector

Return type (numpy.ndarray)

variable_type(*datapoints*, *attribute*, ***kwargs*)
Call to get the variable type of the datapoints, given the attribute.

Parameters

- **datapoints** (*Datapoints*) – [description]
- **attribute** (*str*) – [description]

Returns [description]

Return type (str)

class so_magic.data.features.features.**FeatureFunction**(*function*, *label=None*)
Bases: object

Example: Assume we have a datapoint v = [v_1, v_2, .., v_n, and 2 feature functions f_1, f_2

Then we can produce an encoded vector (eg to feed for training a ML model) like: encoded_vector = [f_1(v), f_2(v)]

is_label(*_attribute*, *value*)

property state

values(*dataset*)

class so_magic.data.features.features.**FeatureIndex**(*keys*)
Bases: object

class so_magic.data.features.features.**FeatureState**(*key*, *reporter*)
Bases: object

class so_magic.data.features.features.**PhiFeatureFunction**
Bases: object

class so_magic.data.features.features.**StateMachine**(*states*, *init_state*)
Bases: object

property current

property state
Construct an object representing the current state

update(**args*, ***kwargs*)

class so_magic.data.features.features.**TrackingFeature**(*feature*, *state_machine*, *variable_type=None*)
Bases: object

classmethod from_callable(*a_callable*, *label=None*, *variable_type=None*)
Construct a feature that has one extract/report capability. Input id is correlated to the features position on the vector (see FeatureFunction above)

label()

property state
Returns the current state

update(**args*, ***kwargs*)

values(*dataset*)

so_magic.data.features.features.**is_callable**(*_self*, *_attribute*, *value*)

so_magic.data.features.features_set module

class so_magic.data.features.features_set.**BaseFeatureSet**(*features=[]*)
Bases: object

classmethod from_raw_extractors(*data*)
Create a Feature for each of the lists in the input data (list). Inner lists must satisfy 0 < len(l)

class so_magic.data.features.features_set.**FeatureConfiguration**(*variables*, *feature_vectors=NOTHING*)
Bases: *so_magic.utils.notification.Observer*

update(*subject:* so_magic.utils.notification.Subject) → None
Receive an update (from a subject); handle an event notification.

valid_encoding(*feature*)

property valid_variables

class so_magic.data.features.features_set.**FeatureManager**(*feature_configuration*, *subject=<so_magic.utils.notification.Subject object>*)

Bases: object

property feature_configuration

class so_magic.data.features.features_set.**FeatureSet**(*features=[]*)

Bases: *so_magic.data.features.features_set.BaseFeatureSet*

property binnable

property encoded

property not_encoded

so_magic.data.features.phi module

This module is responsible to provide a formal way of registering phi functions at runtime. See the 'PhiFunctionRegistrator' class and its 'register' decorator method

class so_magic.data.features.phi.**PhiFunctionMetaclass**(**args*, ***kwargs*)

Bases: type

Class type with a single broadcasting (notifies listeners) facility.

Classes using this class as metaclass, obtain a single broadcasting facility as a class attribute. The class attribute is called 'subject' can be referenced as any class attribute.

Example

class MyExampleClass(metaclass=PhiFunctionMetaclass): pass

instance_object_1 = MyExampleClass() instance_object_2 = MyExampleClass() assert id(MyExampleClass.subject) == id(instance_object_1.subject) == id(instance_object_2.subject)

exception so_magic.data.features.phi.**PhiFunctionNameDeterminationError**

Bases: Exception

class so_magic.data.features.phi.**PhiFunctionRegistrator**

Bases: object

Add phi functions to the registry and notify observers/listeners.

This class provides the 'register' decorator, that client can use to decorate either functions (defined with the def python special word), or classes (defined with the python class special word).

static get_name(*a_callable: Callable*)

Get the 'name' of the input callable object

Parameters **a_callable** (*Callable*) – a callable object to get its name

Returns the name of the callable object

Return type str

classmethod register(*phi_name=''*)

Add a new phi function to phi function registry and notify listeners/observers.

Use this decorator around either a callable function (defined with the 'def' python special word) or a class with a takes-no-arguments (or all-optional-arguments) constructor and a __call__ magic method.

All phi functions are expected to be registered with a __name__ and a __doc__ attribute.

You can select your custom phi_name under which to register the phi function or default to an automatic determination of the phi_name to use.

Automatic determination of phi_name is done by examining either the __name__ attribute of the function or the class name of the class.

Example

```
>>> from so_magic.data.features.phi import PhiFunctionRegistrator
>>> from so_magic.utils import Observer, ObjectRegistry
```

```
>>> class PhiFunctionRegistry(Observer):
...  def __init__(self):
...   self.registry = ObjectRegistry()
...  def update(self, subject, *args, **kwargs):
...   self.registry.add(subject.name, subject.state)
```

```
>>> phis = PhiFunctionRegistry()
```

```
>>> PhiFunctionRegistrator.subject.add(phis)
```

```
>>> @PhiFunctionRegistrator.register()
... def f1(x):
...  '''Multiply by 2.'''
...  return x * 2
Registering input function f1 as phi function, at key f1.
```

```
>>> phis.registry.get('f1').__doc__
'Multiply by 2.'
```

```
>>> input_value = 5
>>> print(f"{input_value} * 2 = {phis.registry.get('f1')(input_value)}")
5 * 2 = 10
```

```
>>> @PhiFunctionRegistrator.register()
... class f2:
...  def __call__(self, data, **kwargs):
...   return data + 5
Registering input class f2 instance as phi function, at key f2.
```

```
>>> input_value = 1
>>> print(f"{input_value} + 5 = {phis.registry.get('f2')(input_value)}")
1 + 5 = 6
```

```
>>> @PhiFunctionRegistrator.register('f3')
... class MyCustomClass:
...  def __call__(self, data, **kwargs):
...   return data + 1
Registering input class MyCustomClass instance as phi function, at key f3.
```

```
>>> input_value = 3
>>> print(f"{input_value} + 1 = {phis.registry.get('f3')(input_value)}")
3 + 1 = 4
```

Parameters **phi_name** (*str, optional*) – custom name to register the phi function. Defaults to automatic computation.

subject = <so_magic.utils.notification.Subject object>

so_magic.data.features.phis module

class so_magic.data.features.phis.**DatapointsAttributePhi**(*datapoints*)
Bases: object

class so_magic.data.features.phis.**ListOfCategoricalPhi**(*datapoints_attribute_phi*)
Bases: object

property attribute

Module contents

class so_magic.data.features.**FeatureManager**(*feature_configuration*, *subject=<so_magic.utils.notification.Subject object>*)
Bases: object

property feature_configuration

so_magic.data.variables package

Submodules

so_magic.data.variables.types module

class so_magic.data.variables.types.**IntervalVariableType**
Bases: so_magic.data.variables.types.NumericalVariableType

Interval numerical variable type

Variables of type interval have interpretable differences; supported operations: [+, -]. There is no true zero.

Example: temperature in Celsius can be measured with an interval variable interval variable

Interpretable difference:

10 degrees drop from 30 degrees Celsius actually means 30 - 10 = 20 degrees Celsius

5 degrees rise 20 degrees Celsius actually means 20 + 5 = 25 degrees Celsius degrees Celsius - 10 degrees Celsius = 20 degrees Celsius

There is no true zero:

Theoretically we can go plus infinite degrees Celsius and minus infinite

There is no number that can “eliminate” (even zero has valid Celsius degrees smaller than 0) a temperature measurement in Celsius degrees

class so_magic.data.variables.types.**NominalVariableType**
Bases: so_magic.data.variables.types.CategoricalVariableType

Nominal variable; discrete variables with undefined ordering; eg country-names

class so_magic.data.variables.types.**OrdinalVariableType**
Bases: so_magic.data.variables.types.CategoricalVariableType

Ordinal variable; discrete variables with a defined ordering; eg days-of-the-week

class so_magic.data.variables.types.**RatioVariableType**
Bases: so_magic.data.variables.types.NumericalVariableType

Ratio numerical variable where all operations are supported (+, -, *, /) and true zero is defined; eg weight

class so_magic.data.variables.types.**VariableTypeFactory**
Bases: object

static create(*variable_type: str*, **args*, ***kwargs*)

static infer(*datapoints*, *attribute*, *sortable=True*, *ratio=None*)
Semi-automatic identification; requires some input to assist;

Module contents

Submodules

so_magic.data.command_factories module

class so_magic.data.command_factories.**DataManagerCommandFactory**(*data_manager*, *command_factory=<class 'so_magic.data.command_factories.DataManagerC*
Bases: object

build_command_prototype()

name: str

subject: *so_magic.utils.notification.Subject*

class so_magic.data.command_factories.**DataManagerCommandFactoryBuilder**
Bases: object

classmethod create_factory(*name*, *callback*)

subclasses = {}

so_magic.data.commands_manager module

class so_magic.data.commands_manager.**CommandGetter**(*commands_accumulator=CommandsAccumulator(commands={})*)
Bases: object

property accumulator

class so_magic.data.commands_manager.**CommandsAccumulator**
Bases: *so_magic.utils.notification.Observer*

update(*subject*, **args*, ***kwargs*) → None
Receive an update (from a subject); handle an event notification.

class so_magic.data.commands_manager.**CommandsManager**(*commands_getter=CommandGetter(_commands_accumulator=C decorators=None*)

Bases: object

[summary]

Parameters

- **prototypes** (*dict, optional*) – initial prototypes to be supplied
- **command_factory** (*callable, optional*) – a callable that returns an instance of Command

property command

property commands_dict

so_magic.data.data_manager module

class so_magic.data.data_manager.**DataManager**(*engine*, *phi_function_class*, *feature_manager*, *commands_manager=CommandsManager(_commands_getter=CommandGetter decorators=None)*)

Bases: object

property command

property commands

property datapoints

property phi_class

property phis

class so_magic.data.data_manager.**Phis**(*registry:* so_magic.utils.registry.ObjectRegistry = *NOTHING*)

Bases: *so_magic.utils.notification.Observer*

registry: *so_magic.utils.registry.ObjectRegistry*

update(*subject*, **args*, ***kwargs*)

Receive an update (from a subject); handle an event notification.

so_magic.data.datapoints_manager module

Defines the DatapointsManager type (class); a centralized facility where all datapoints objects should arrived and be retrieved from.

class so_magic.data.datapoints_manager.**DatapointsManager**(*datapoints_objects=NOTHING*)

Bases: *so_magic.utils.notification.Observer*

Manage operations revolved around datapoints collection objects.

Instances of this class are able to monitor (listener/observer pattern) the creation of datapoints collection objects and store them in a dictionary structure. They also provide retrieval methods to the client to "pick up" a datapoints object.

Parameters **datapoints_objects** (*dict, optional*) – the initial structure that stores datapoints objects

property datapoints: Optional[Iterable]

The most recently stored datapoints object.

Returns the reference to the datapoints object

Return type Optional[Iterable]

property state
The latest (most recent) key used to store a datapoints object.

Returns the key under which we stored a datapoints object last time

Return type str

update(*subject:* so_magic.utils.notification.Subject)
Update our state based on the event/observation captured/made.

Stores the datapoints object observed in a dictionary using a the Subject name attribute as key.

Parameters **subject** (Subject) – the subject object observed; it acts as an event

Raises

- **RuntimeError** – in case there is no 'name' attribute on the subject or if it is an empty string ''
- **RuntimeError** – in case the 'name' attribute on the subject has already been used to store a datapoints object

so_magic.data.dataset module

class so_magic.data.dataset.**Dataset**(*datapoints*, *name=None*, *features=[]*)
Bases: object

High level representation of data, of some form.

Instances of this class encapsulate observations in the form of datapoints as well as their respective feature vectors. Feature vectors can then be trivially "fed" into a Machine Learning algorithm (eg SOM).

Parameters

- **()** (*datapoints*) –
- **name** (*str, optional*) –

Returns [description]

Return type [type]

property features

so_magic.data.discretization module

class so_magic.data.discretization.**AbstractAlgorithm**(*callback: callable*, *arguments: list = NOTHING*, *parameters: dict = NOTHING*)
Bases: *so_magic.data.discretization.AlgorithmInterface*, abc.ABC

arguments: list

callback: callable

parameters: dict

class so_magic.data.discretization.**AbstractDiscretizer**
Bases: *so_magic.data.discretization.DiscretizerInterface*

discretize(**args*, ***kwargs*)

class so_magic.data.discretization.**AlgorithmArguments**(*arg_types*, *default_values*)
Bases: object

An algorithms expected positional arguments.

values(**args*)

exception so_magic.data.discretization.**AlgorithmArgumentsError**
Bases: Exception

class so_magic.data.discretization.**AlgorithmInterface**
Bases: abc.ABC

abstract run(**args*, ***kwargs*)

class so_magic.data.discretization.**BaseBinner**(*algorithm*)
Bases: *so_magic.data.discretization.BinnerInterface*

bin(*values*, *bins*)
It is assumed numerical (ratio or interval) variable or ordinal (not nominal) categorical variable.

class so_magic.data.discretization.**BaseDiscretizer**(*binner*)
Bases: *so_magic.data.discretization.AbstractDiscretizer*

discretize(**args*, ***kwargs*)
Expects args: dataset, feature and kwargs; 'nb_bins'.

class so_magic.data.discretization.**BinnerClass**
Bases: object

subclasses = {}

class so_magic.data.discretization.**BinnerFactory**
Bases: object

create_binner(**args*, ***kwargs*) → *so_magic.data.discretization.BaseBinner*

equal_length_binner(**args*, ***kwargs*) → *so_magic.data.discretization.BaseBinner*
Binner that create bins of equal size (max_value - min_value)

parent_class
alias of *so_magic.data.discretization.BinnerClass*

quantisized_binner(**args*, ***kwargs*) → *so_magic.data.discretization.BaseBinner*
Binner that will adjust the bin sizes so that the observations are evenly distributed in the bins

Raises **NotImplementedError** – [description]

Returns [description]

Return type *BaseBinner*

class so_magic.data.discretization.**BinnerInterface**
Bases: abc.ABC

abstract bin(*values*, *bins*)

class so_magic.data.discretization.**BinningAlgorithm**
Bases: object

classmethod from_built_in(*algorithm_id*)

subclasses = {'pd.cut': <class
'so_magic.data.discretization.PDCutBinningAlgorithm'>}

class so_magic.data.discretization.**Discretizer**(*binner*)
Bases: *so_magic.data.discretization.BaseDiscretizer*

property algorithm

classmethod from_algorithm(*alg*)

class so_magic.data.discretization.**DiscretizerInterface**
Bases: abc.ABC

discretize(**args*, ***kwargs*)

class so_magic.data.discretization.**FeatureDiscretizer**(*binner*, *feature*)
Bases: *so_magic.data.discretization.BaseDiscretizer*

discretize(**args*, ***kwargs*)
Expects args: dataset, nb_bins.

class so_magic.data.discretization.**FeatureDiscretizerFactory**(*binner_factory*)
Bases: object

categorical(*feature*, ***kwargs*) → *so_magic.data.discretization.FeatureDiscretizer*

numerical(*feature*, ***kwargs*) → *so_magic.data.discretization.FeatureDiscretizer*

class so_magic.data.discretization.**MagicAlgorithm**(*callback: callable*, *arguments: list = NOTHING*, *parameters: dict = NOTHING*)
Bases: *so_magic.data.discretization.AbstractAlgorithm*

arguments: list

callback: callable

property output

parameters: dict

run(**args*, ***kwargs*)

set_default_parameters()

update_parameters(***kwargs*)

exception so_magic.data.discretization.**MagicAlgorithmError**
Bases: Exception

exception so_magic.data.discretization.**MagicAlgorithmParametersError**
Bases: Exception

class so_magic.data.discretization.**PDCutBinningAlgorithm**(*callback: callable*, *arguments: list = NOTHING*, *parameters: dict = NOTHING*)
Bases: *so_magic.data.discretization.MagicAlgorithm*

arguments: list

callback: callable

parameters: dict

so_magic.data.discretization.**call_method**(*a_callable*)

so_magic.data.encoding module

class so_magic.data.encoding.**EncoderInterface**
Bases: abc.ABC

abstract encode(**args*, ***kwargs*)

class so_magic.data.encoding.**NominalAttributeEncoder**(*values_set: list = NOTHING*)
Bases: *so_magic.data.encoding.EncoderInterface*, abc.ABC

Encode the observations of a categorical nominal variable.

The client code can supply the possible values for the nominal variable, if known a priori. The possible values are stored in the 'values_set' attribute/property. If they are not supplied they should be computed at runtime (when running the encode method).

It also defines and stores the string identifiers for each column produced in the 'columns attribute/property.

Parameters **values_set** (*list*) – the possible values of the nominal variable observations, if known a priori

columns

values_set

so_magic.data.interfaces module

Defines interfaces related to various operations on table-like data.

class so_magic.data.interfaces.**TabularIterator**
Bases: abc.ABC

Iterate over the rows or columns of a table-lie data structure.

Classes implementing this interface gain the ability to iterate over the values found in the rows or the columns of a table-like data structure. They can also iterate over the columns indices/identifiers.

abstract columnnames(*data*) → Union[Iterable[str], Iterable[int]]
Iterate over data (table) column indices/identifiers.

Parameters **data** (*object*) – the (data) table to iterate over its columns indices/identifiers

Returns the column indices/identifiers of the (data) table

Return type Union[Iterable[str], Iterable[int]]

abstract itercolumns(*data*) → Iterable
Iterate over the (data) table's columns.

Get an iterable over the table's columns.

Parameters **data** (*object*) – the (data) table to iterate over its columns

Returns the columns of the (data) table

Return type Iterable

abstract iterrows(*data*) → Iterable
Iterate over the (data) table's rows.

Get an iterable over the table's rows.

Parameters **data** (*object*) – the (data) table to iterate over its rows

Returns the rows of the (data) table

Return type Iterable

class so_magic.data.interfaces.**TabularMutator**
Bases: abc.ABC

Mutate (alter) the contents of a table-like data structure.

Classes implementing this interface supply their instances the ability to alter the contents of a table-like data structure.

abstract add_column(**args*, ***kwargs*)
Add a new column to table-like data.

Raises **NotImplementedError** – [description]

class so_magic.data.interfaces.**TabularRetriever**
Bases: abc.ABC

Operations on table-like data.

Classes implementing this interface gain the ability to perform various operations on data structures that resemble a table (have indexable columns, rows, etc):

most importantly they can slice through the data (retrieve specific row or column)

abstract column(*identifier: Union[str, int]*, *data*) → Iterable
Slice though the data (table) and get the specified column's values.

Parameters

- **identifier** (*Union[str, int]*) – unique identifier/index of column
- **data** (*object*) – the data to slice through

Returns the values contained in the column requested

Return type Iterable

abstract get_numerical_attributes(*data*) → Iterable
Get the data's attributes that represent numerical values.

Returns the attributes that fall under the Numerical Variables: either Ratio or Interval type of variables.

Two type of numerical variables are supported:

Ratio variable: numerical variable where all operations are supported (+, -, *, /) and true zero is defined; eg weight.

Interval variable: numerical variable where differences are interpretable; supported operations: [+, -]; no true zero; eg temperature in centigrade (ie Celsius).

Parameters **data** (*object*) – the data from which to retrieve the numerical attributes

Returns the numerical attributes found

Return type Iterable

abstract nb_columns(*data*) → int
Get the number of columns that the data (table) have.

Parameters **data** (*object*) – the data (table) to count its columns

Returns the number of the (data) table's columns

Return type int

abstract nb_rows(*data*) → int
Get the number of rows that the data (table) have.

Parameters **data** (*object*) – the data (table) to count its rows

Returns the number of the (data) table's rows

Return type int

abstract row(*identifier*, *data*)
Slice though the data (table) and get the specified row's values.

Parameters

- **identifier** (*Union[str, int]*) – unique identifier/index of row
- **data** (*object*) – the data to slice through

Returns the values contained in the row requested

Return type Iterable

so_magic.data.magic_datapoints_factory module

This module is responsible to provide means of creating (instantiating) objects representing Datapoints collections.

class so_magic.data.magic_datapoints_factory.**BroadcastingDatapointsFactory**(*subject: so_magic.utils.notification.Subject = NOTHING*)

Bases: *so_magic.data.datapoints.datapoints.DatapointsFactory*

Creates Datapoints objects and informs its subscribers when that happens.

A factory class that informs its subscribers when a new object that implements the DatapointsInterface is created (following a request).

Parameters **subject** (Subject, *optional*) – the subject of observation; the "thing" that others listen to

create(*datapoints_factory_type: str*, **args*, ***kwargs*) → Iterable
Create new Datapoints and inform subscribers.

The factory method that returns a new object of DatapointsInterface, by looking at the registered constructors to delegate the object creation.

Parameters **datapoints_factory_type** (*str*) – the name of the "constructor" to use

Raises **RuntimeError** – [description]

Returns instance implementing the DatapointsInterface

Return type Iterable

name: str

subject: *so_magic.utils.notification.Subject*

Module contents

so_magic.data.**init_data_manager**(*engine*)

so_magic.som package

Submodules

so_magic.som.factory module

class so_magic.som.factory.**SelfOrganizingMapFactory**(*trainer=NOTHING*, *subject=NOTHING*)
Bases: object

create(*dataset*, *nb_cols*, *nb_rows*, ***kwargs*)

so_magic.som.manager module

class so_magic.som.manager.**MagicMapManager**(*so_master*)
Bases: object

train(*nb_cols*, *nb_rows*, ***kwargs*)

class so_magic.som.manager.**MapId**(*dataset_name*, *n_columns*, *n_rows*, *initialization*, *map_type*, *grid_type*)
Bases: object

static from_self_organizing_map(*somap*, ***kwargs*)

class so_magic.som.manager.**MapManager**(*map_factory=SelfOrganizingMapFactory(trainer=SomTrainer(infer_map=<function infer_map>)*, *subject=<so_magic.utils.notification.Subject object>)*)
Bases: object

get_map(**args*, ***kwargs*)
args: 'dataset', 'nb_cols', 'nb_rows' kwargs: 'initialization', 'maptype', 'gridtype'

train(*dataset*, *nb_cols*, *nb_rows*, ***kwargs*)

so_magic.som.self_organising_map module

exception so_magic.som.self_organising_map.**NoFeatureVectorsError**
Bases: Exception

class so_magic.som.self_organising_map.**SelfOrganizingMap**(*som*, *dataset_name*)
Bases: object

cluster(*nb_clusters*, *random_state=None*)

datapoint_coordinates(*index*)
Get the best-matching unit (bmu) coordinates of the datapoint indexed by the input pointer.

Bmu is simply the neuron on the som grid that is closest to the projected-into-2D-space datapoint.

get_map_id()

property grid_type

property height

property nb_clusters

neurons_coordinates()

project(*datapoint*)
Compute the coordinates of a (potentially unseen) datapoint.

It is assumed that the codebook has been computed already.

property type

property visual_umatrix

property width

class so_magic.som.self_organising_map.SomTrainer(*infer_map: callable*)
Bases: object

static from_callable()

infer_map

so_magic.som.self_organising_map.infer_map(*nb_cols*, *nb_rows*, *dataset*, ***kwargs*)
Infer a self-organizing map from dataset.

initialcodebook = None, kerneltype = 0, maptype = 'planar', gridtype = 'rectangular', compactsupport = False, neighborhood = 'gaussian', std_coeff = 0.5, initialization = None

Module contents

class so_magic.som.MagicMapManager(*so_master*)
Bases: object

train(*nb_cols*, *nb_rows*, ***kwargs*)

so_magic.utils package

The utils package provides a set of useful classes (and functions) that are commonly used by multiple SoMagic components. These classes serve as building blocks for the SoMagic library and they have been implemented following well-known and established software engineering patterns and best-practices. After all we want our Magic software to be based on solid foundations.

Submodules Overview

so_magic.utils.command_factory_interface module

This module is responsible to define an interface to construct Command objects (instances of the Command class).

so_magic.utils.command_interface module

so_magic.utils.commands module

so_magic.utils.linear_mapping module

This module exposes the MapOnLinearSpace class and the 'universal_constructor' method to create instances of it. Instances of MapOnLinearSpace can be used to project a number from one linear space to another.

so_magic.utils.mediator module

so_magic.utils.memoize module

Implementation of the object pool

so_magic.utils.notification module

Typical subject/observers pattern implementation. You can see this pattern mentioned also as event/notification or broadcast/listeners.

Provides the Observer class, serving as the interface that needs to be implemented by concrete classes; the update method needs to be overrode. Concrete Observers react to the notifications/updates issued by the Subject they had been attached to.

Provides the Subject class, serving with mechanisms to subscribe/unsubscribe (attach/detach) observers and also with a method to "notify" all subscribers about events.

so_magic.utils.registry module

so_magic.utils.singleton module

so_magic.utils.transformations module

This module provides the Transformer class. Its constructor can be used to create data transformation methods.

Modules of utils package

so_magic.utils.commands module

class so_magic.utils.commands.**Command**(*receiver*, *method: str*, **args*)

Bases: so_magic.utils.commands.BaseCommand

An runnable/executable Command that acts as a prototype through the 'copy' python magic function.

When a command instance is invoked with 'copy', the receiver is copied explicitly in a shallow way. The rest of the command arguments are assumed to be performance invariant (eg it is not expensive to copy the 'method' attribute, which is a string) and are handled automatically.

class so_magic.utils.commands.**CommandHistory**
Bases: object

The global command history is just a stack; supports 'push' and 'pop' methods.

pop() → *so_magic.utils.commands.Command*

push(*command:* so_magic.utils.commands.Command)

property stack

class so_magic.utils.commands.**CommandInterface**
Bases: abc.ABC

Standalone command, encapsulating all logic and data needed, required for execution.

abstract execute() → None
Execute the command; run the commands logic.

class so_magic.utils.commands.**Invoker**(*history:* so_magic.utils.commands.CommandHistory)
Bases: object

A class that simply executes a command and pushes it into its internal command history stack.

Parameters **history** (CommandHistory) – the command history object which acts as a stack

execute_command(*command:* so_magic.utils.commands.Command)

so_magic.utils.linear_mapping module

This module exposes the MapOnLinearSpace class and the 'universal_constructor' method to create instances of it. Instances of MapOnLinearSpace can be used to project a number from one linear space to another.

class so_magic.utils.linear_mapping.**LinearScale**(*lower_bound: int*, *upper_bound: int*)
Bases: object

A numerical linear scale (range between 2 numbers) where numbers fall in between.

Raises **ValueError** – in case the lower bound is not smaller than the upper

Parameters

- **lower_bound** (*int*) – the minimum value a number can take on the scale
- **upper_bound** (*int*) – the maximum value a number can take on the scale

classmethod create(*two_element_list_like*)

class so_magic.utils.linear_mapping.**MapOnLinearSpace**(*from_scale:* so_magic.utils.linear_mapping.LinearScale, *target_scale:* so_magic.utils.linear_mapping.LinearScale, *reverse: bool = False*)
Bases: object

Projection of a number from one linear scale to another.

Instances of this class can transform an input number and map it from an initial scale to a target scale.

Parameters

- **_from_scale** (LinearScale) – the scale where the number is initially mapped
- **_target_scale** (LinearScale) – the (target) scale where the number should be finally transformed/mapped to

- **_reverse** (*bool*) – whether the target scale is inverted or not

property from_scale

property reverse

property target_scale

transform(*number*)
Transform the input number to a different linear scale.

classmethod universal_constructor(*from_scale*, *target_scale*, *reverse=False*)

so_magic.utils.mediator module

class so_magic.utils.mediator.**BaseComponent**(*mediator: Optional[so_magic.utils.mediator.Mediator] = None*)
Bases: object

The Base Component provides the basic functionality of storing a mediator's instance inside component objects.

property mediator: so_magic.utils.mediator.Mediator

class so_magic.utils.mediator.**GenericMediator**(**components*, ***kwargs*)
Bases: so_magic.utils.mediator.Mediator

Abstract Mediator class that automatically configures components received as *args through the constructor.

so_magic.utils.memoize module

Implementation of the object pool

class so_magic.utils.memoize.**ObjectsPool**(*constructor*, *build_hash*, *objects={}*)
Bases: object

Class of objects that are able to return a reference to an object upon request.

Whenever an object is requested, it is checked whether it exists in the pool. Then if it exists, a reference is returned, otherwise a new object is constructed (given the provided callable) and its reference is returned.

Parameters

- **constructor** (*callable*) – able to construct the object given arguments
- **objects** (*dict*) – the data structure representing the object pool

get_object(**args*, ***kwargs*)
Request an object from the pool.

Get or create an object given the input parameters. Existence in the pool is done using the python-build-in hash function. The input *args and **kwargs serve as input in the hash function to create unique keys with which to "query" the object pool.

Returns the reference to the object that corresponds to the input arguments, regardless of whether it was found in the pool or not

Return type object

classmethod new_empty(*constructor*, *build_hash=None*)

so_magic.utils.notification module

Typical subject/observers pattern implementation. You can see this pattern mentioned also as event/notification or broadcast/listeners.

Provides the Observer class, serving as the interface that needs to be implemented by concrete classes; the update method needs to be overrode. Concrete Observers react to the notifications/updates issued by the Subject they had been attached to.

Provides the Subject class, serving with mechanisms to subscribe/unsubscribe (attach/detach) observers and also with a method to "notify" all subscribers about events.

class so_magic.utils.notification.**Observer**
Bases: so_magic.utils.notification.ObserverInterface, abc.ABC

abstract update(**args*, ***kwargs*) → None
Receive an update (from a subject); handle an event notification.

class so_magic.utils.notification.**Subject**(**args*, ***kwargs*)
Bases: so_magic.utils.notification.SubjectInterface

The Subject owns some important state and can notify observers.

Both the _state and _observers attributes have a simple implementation, but can be overrode to accommodate for more complex scenarios.

The observers/subscribers are implemented as a python list. In more complex scenarios, the list of subscribers can be stored more comprehensively (categorized by event type, etc.).

The subscription management methods provided are 'attach' and 'detach' to add or remove a subscriber respectively

add(**observers*)
Subscribe multiple observers at once.

attach(*observer:* so_magic.utils.notification.Observer) → None
Attach an observer to the subject; subscribe the observer.

detach(*observer:* so_magic.utils.notification.Observer) → None
Detach an observer from the subject; unsubscribe the observer.

notify() → None
Trigger an update in each subscriber/observer.

property state

so_magic.utils.registry module

class so_magic.utils.registry.**ObjectRegistry**(**args*, ***kwargs*)
Bases: abc.ABC

Simple dict-like retrieval/inserting "store" facility.

add(*key*, *value*)

get(*key*)

pop(*key*)

remove(*key*)

exception so_magic.utils.registry.**ObjectRegistryError**
Bases: Exception

so_magic.utils.singleton module

class so_magic.utils.singleton.**Singleton**
Bases: type

so_magic.utils.transformations module

This module provides the Transformer class. Its constructor can be used to create data transformation methods.

class so_magic.utils.transformations.**Transformer**(**args*, ***kwargs*)
Bases: so_magic.utils.transformations.RuntimeTransformer

Exposed python objects (package's public interface)

class so_magic.utils.**BaseComponent**(*mediator: Optional[so_magic.utils.mediator.Mediator] = None*)
Bases: object

The Base Component provides the basic functionality of storing a mediator's instance inside component objects.

property mediator: so_magic.utils.mediator.Mediator

class so_magic.utils.**Command**(*receiver*, *method: str*, **args*)
Bases: so_magic.utils.commands.BaseCommand

An runnable/executable Command that acts as a prototype through the 'copy' python magic function.

When a command instance is invoked with 'copy', the receiver is copied explicitly in a shallow way. The rest of the command arguments are assumed to be performance invariant (eg it is not expensive to copy the 'method' attribute, which is a string) and are handled automatically.

class so_magic.utils.**CommandFactoryInterface**
Bases: abc.ABC

Define a way to create objects of type Command.

Classes implementing this interface define a way to construct (initialize) new Command objects (class instances).

abstract construct(**args*, ***kwargs*) → *so_magic.utils.command_interface.CommandInterface*
Construct a new Command object (new class instance) that can be executed.

Returns the command object (instance)

Return type *Command*

class so_magic.utils.**CommandFactoryType**(**args*, ***kwargs*)
Bases: so_magic.utils.subclass_registry.SubclassRegistry

class so_magic.utils.**CommandHistory**
Bases: object

The global command history is just a stack; supports 'push' and 'pop' methods.

pop() → *so_magic.utils.commands.Command*

push(*command:* so_magic.utils.commands.Command)

property stack

class so_magic.utils.**GenericMediator**(**components*, ***kwargs*)
Bases: so_magic.utils.mediator.Mediator

Abstract Mediator class that automatically configures components received as *args through the constructor.

class so_magic.utils.**Invoker**(*history:* so_magic.utils.commands.CommandHistory)
Bases: object

A class that simply executes a command and pushes it into its internal command history stack.

Parameters **history** (CommandHistory) – the command history object which acts as a stack

execute_command(*command:* so_magic.utils.commands.Command)

class so_magic.utils.**MapOnLinearSpace**(*from_scale:* so_magic.utils.linear_mapping.LinearScale, *target_scale:* so_magic.utils.linear_mapping.LinearScale, *reverse: bool = False*)
Bases: object

Projection of a number from one linear scale to another.

Instances of this class can transform an input number and map it from an initial scale to a target scale.

Parameters

- **_from_scale** (LinearScale) – the scale where the number is initially mapped
- **_target_scale** (LinearScale) – the (target) scale where the number should be finally transformed/mapped to
- **_reverse** (*bool*) – whether the target scale is inverted or not

property from_scale

property reverse

property target_scale

transform(*number*)
Transform the input number to a different linear scale.

classmethod universal_constructor(*from_scale*, *target_scale*, *reverse=False*)

class so_magic.utils.**ObjectRegistry**(**args*, ***kwargs*)
Bases: abc.ABC

Simple dict-like retrieval/inserting "store" facility.

add(*key*, *value*)

get(*key*)

pop(*key*)

remove(*key*)

exception so_magic.utils.**ObjectRegistryError**
Bases: Exception

class so_magic.utils.**ObjectsPool**(*constructor*, *build_hash*, *objects={}*)
Bases: object

Class of objects that are able to return a reference to an object upon request.

Whenever an object is requested, it is checked whether it exists in the pool. Then if it exists, a reference is returned, otherwise a new object is constructed (given the provided callable) and its reference is returned.

Parameters

- **constructor** (*callable*) – able to construct the object given arguments
- **objects** (*dict*) – the data structure representing the object pool

get_object(**args*, ***kwargs*)
Request an object from the pool.

Get or create an object given the input parameters. Existence in the pool is done using the python-build-in hash function. The input *args and **kwargs serve as input in the hash function to create unique keys with which to "query" the object pool.

Returns the reference to the object that corresponds to the input arguments, regardless of whether it was found in the pool or not

Return type object

classmethod new_empty(*constructor*, *build_hash=None*)

class so_magic.utils.**Observer**
Bases: so_magic.utils.notification.ObserverInterface, abc.ABC

class so_magic.utils.**Singleton**
Bases: type

class so_magic.utils.**SubclassRegistry**(**args*, ***kwargs*)
Bases: type

Subclass Registry

A (parent) class using this class as metaclass gains the 'subclasses' class attribute as well as the 'create' and 'register_as_subclass' class methods.

The 'subclasses' attribute is a python dictionary having string identifiers as keys and subclasses of the (parent) class as values.

The 'register_as_subclass' class method can be used as a decorator to indicate that a (child) class should belong in the parent's class registry. An input string argument will be used as the unique key to register the subclass.

The 'create' class method can be invoked with a (string) key and suitable constructor arguments to later construct instances of the corresponding child class.

Example

```
>>> from so_magic.utils import SubclassRegistry
```

```
>>> class ParentClass(metaclass=SubclassRegistry):
...  pass
```

```
>>> ParentClass.subclasses
{}
```

```
>>> @ParentClass.register_as_subclass('child')
... class ChildClass(ParentClass):
...  def __init__(self, child_attribute):
...   self.attr = child_attribute
```

```
>>> child_instance = ParentClass.create('child', 'attribute-value')
>>> child_instance.attr
'attribute-value'
```

```
>>> type(child_instance).__name__
'ChildClass'
```

```
>>> isinstance(child_instance, ChildClass)
True
```

```
>>> isinstance(child_instance, ParentClass)
True
```

```
>>> {k: v.__name__ for k, v in ParentClass.subclasses.items()}
{'child': 'ChildClass'}
```

create(*subclass_identifier*, **args*, ***kwargs*)
Create an instance of a registered subclass, given its unique identifier and runtime (constructor) arguments.

Invokes the identified subclass constructor passing any supplied arguments. The user needs to know the arguments to supply depending on the resulting constructor signature.

Parameters **subclass_identifier** (*str*) – the unique identifier under which to look for the corresponding subclass

Raises **ValueError** – In case the given identifier is unknown to the parent class

Returns the instance of the registered subclass

Return type object

register_as_subclass(*subclass_identifier*)
Register a class as subclass of the parent class.

Adds the subclass' constructor in the registry (dict) under the given (str) identifier. Overrides the registry in case of "identifier collision". Can be used as a python decorator.

Parameters **subclass_identifier** (*str*) – the user-defined identifier, under which to register the subclass

class so_magic.utils.**Subject**(**args*, ***kwargs*)
Bases: so_magic.utils.notification.SubjectInterface

The Subject owns some important state and can notify observers.

Both the _state and _observers attributes have a simple implementation, but can be overrode to accommodate for more complex scenarios.

The observers/subscribers are implemented as a python list. In more complex scenarios, the list of subscribers can be stored more comprehensively (categorized by event type, etc.).

The subscription management methods provided are 'attach' and 'detach' to add or remove a subscriber respectively

add(**observers*)
Subscribe multiple observers at once.

attach(*observer:* so_magic.utils.notification.Observer) → None
Attach an observer to the subject; subscribe the observer.

detach(*observer:* so_magic.utils.notification.Observer) → None
Detach an observer from the subject; unsubscribe the observer.

notify() → None
Trigger an update in each subscriber/observer.

property state

class so_magic.utils.Transformer(*args, **kwargs)
Bases: so_magic.utils.transformations.RuntimeTransformer

5.1.2 Submodules

5.1.3 so_magic.so_master module

class so_magic.so_master.SoMaster(data_manager, dataset_constructor, magic_map_manager_constructor)
Bases: object

property command

property commands_decorators

property datapoints

property dataset

load_data(file_path)

property map

5.1.4 Module contents

so_magic.init_so_magic()

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

C

D

E

F

N

O

P

Q

R

S

T

U

V

W