order

Release 1.3.6

Marcel Rieger

Jul 06, 2022

CONTENTS

1 Getting started **3**
- 1.1 Quickstart . . . 3
- 1.2 API Reference . . . 12

2 Installation and dependencies **63**

3 Contributing and testing **65**

4 Development **67**

Python Module Index **69**

Index **71**

If you're designing a high-energy physics analysis (e.g. with data recorded by an LHC experiment at CERN), manual bookkeeping of external data can get complicated quite fast. *order* provides a pythonic class collection that helps you structuring

- analyses,
- MC campaigns,
- datasets,
- physics process and cross sections,
- channels,
- categories,
- variables, and
- systematic shifts.

CHAPTER

ONE

GETTING STARTED

1.1 Quickstart

The sections below explain the most important classes, the concept of mixins and some notes on copying objects. Open the intro.ipynb notebook to see these classes in action or run it interactively on binder:

order follows a very strict API design and naming scheme.

1. Methods start with (or at least contain) verbs. E.g. setters and getters are prefixed with `set_` and `get_`, respectively.
2. Properties implement type checks and, where applicable, conversions! Assigments such as `my_object.some_string_attribute = 123` (can) immediately fail. This way, type ambiguities during analysis execution are mitigated.
3. The code style is PEP 8 compatible (checked via flake8).

Contents

- UniqueObject *and* UniqueObjectIndex
- Analysis, Campaign *and* Config
- Dataset *and* Process
- Channel *and* Category
- Shift
- Variable
- *Mixin classes*
- *Copying objects*

1.1.1 *UniqueObject* and *UniqueObjectIndex*

Before diving into the actual physics-related classes, it is necessary to understand their primary uncerlying base class: *UniqueObject*.

A *UniqueObject* essentially has three attributes: a *name* (string), an *id* (integer) and a uniqueness *context*. The combination (*name*, *context*) as well as (*id*, *context*) are forced to be unique per inheriting class (see example below). This way, *name* and *id* are logically connected and once used, they cannot be associated to any other object. The mechanism unambiguously assigns

1. a human-readable name to objects (e.g.) for expressive use on the command-line, and
2. a unique identifier to objects (e.g.) for encoding information in data structures such as ROOT trees.

Consider this example implementation of a physics process:

```
import order as od

class Process(od.UniqueObject):

    def __init__(self, name, id, xsec, context=None):
        super(Process, self).__init__(name, id, context=context)

        self.xsec = xsec
```

When creating a process via

```
ttbar = Process("ttbar", 1, 831.76)

print(ttbar)
# -> "Process(name=ttbar, id=1, context=process)"
```

both the *name* `"ttbar"` and the *id* `1` are globally reserved and no other instance of the same class can be created with that *name* or *id* within the same *context*. For the `ttbar` process object that we just created, the *context* argument was *None*, which tells *order* to use a default value (the lower-case class name `"process"` in this case). When trying to create another process with e.g. the same id,

```
ttbarH = Process("ttbarH", 1, 0.508)
# -> order.unique.DuplicateIdException: an object with id '1' already exists in the␣
↪uniqueness context 'process'
```

an exception (`DuplicateIdException`) is raised (or a `DuplicateNameException` in case of duplicate names). When the object is placed into an other uniqueness context, however, the instance creation succeeds. This can be achieved by passing a different *context* to the `Process` constructor, or via using the *uniqueness_context()* guard

```
with od.uniqueness_context("other_analysis"):
    ttbarH = Process("ttbarH", 1, 0.508)

# same as
# ttbarH = Process("ttbarH", 1, 0.508, context="other_analysis")

print(ttbarH)
# -> "Process(name=ttbarH, id=1, context=other_analysis)"
```

The class itself can be seen as part of the uniqueness context. For other classes that inherit from *UniqueObject*, the same (*name*, *context*) and (*id*, *context*) combinations can be used again.

Internally, each class maintains its own instance cache to check for duplicate names and ids. The cache functionality is implemented in *UniqueObjectIndex*. Additionaly, this class is (mainly) employed for constructing **has / owns** relations between objects. Let's extend the example above by creating a simple *Dataset* class that is aware of the physics processes it contains:

```
class Dataset(od.UniqueObject):

    def __init__(self, *args, **kwargs):
        super(Dataset, self).__init__(*args, **kwargs)

        self.processes = UniqueObjectIndex(cls=Process)

dataset_ttbar = Dataset("ttbar", 1)
dataset_ttbar.processes.add(ttbar)

print(dataset_ttbar.processes)
# -> "UniqueObjectIndex(cls=Process, len=1)"
```

You will find similar constructs all across *order*, however, with several convenience methods. To reflect the uniqueness rules explained above, a *UniqueObjectIndex* stores objects per *context*. When studying the API reference, you will notice a *context* argument in the signatures of most of its methods (such as *len()*, *names()*, *ids()*, *keys()* or *values()*). For instance, `processes.len()` will return 1, whereas `processes.len(context="other_analysis")` will return 0 as no object with the uniquess context `"other_analysis"` was added yet.

1.1.2 *Analysis*, *Campaign* and *Config*

An instance of the *Analysis* class represents the overarching object containing information of a physics analysis.

Varying requirements across data-taking periods, complex sub measurements, or simply different revisions of the same analysis are typical reasons why the structuring of information is quite demanding over the course of an analysis with sometimes unpredictable incidents and deadlines (code time uncertainty is a thing). For this purpose, *order* introduces two classes: *Campaign* and *Config*.

A *Campaign* describes and contains **analysis-independent** information, such as detector alignment settings, event simulation settings, recorded / simulated datasets, etc. In general, a pre-configured campaign object could be provided centrally by a working group or collaboration.

A *Config* object holds **analysis-dependent** information related to a certain campaign. Thus, a config is unambiguously assigned to both an analysis and a campaign.

Consider, for example, offline triggers that are used to select events specifically in one analysis. By construction, they should not be stored in a campaign object (which is **analysis-independent**), but they might also change between different data-taking periods and therefore should not be stored in the analysis object itself. Instead, a config object is an ideal place for such information.

```
import order as od

# the campaign (could be configured externally)
campaign_2018 = od.Campaign("data_taking_2018", 1, ecm=13)

# create the analysis
analysis = od.Analysis("my_analysis", 1)

# add a config for the 2018 campaign
```

(continues on next page)

(continued from previous page)

```
# when no name or id are passed, it has the same as the campaign
cfg = analysis.add_config(campaign_2018)

# add trigger information as auxiliary data
cfg.set_aux("triggers", ["trigger_ee", "trigger_emu", "trigger_mumu"])
```

An analysis can contain several config objects for the same campaign. Just note that uniqueness rules apply here as well.

See the intro.ipynb notebook for more examples.

1.1.3 *Dataset* and *Process*

Physics processes and simualted / recorded datasets are described by two classes: *Process* and *Dataset*.

Besides a *name* and an *id*, a *Process* object has cross sections for different center-of-mass energies, labels and colors for plotting purposes, and information about whether or not it describes real data or MC (it inherits from the *DataSourceMixin*, see mixin classes). Cross section values are automatically converted to scinum.Number instances, which are able to store multiple uncertainties, provide automatic error propagation and also support NumPy arrays.

Moreover, processes can have subprocesses and *order* provides convenience methods to work with arbitrarily deep process lookup.

```
import order as od
from scinum import Number, UP, DOWN, REL, ABS

ttH = od.Process("ttH", 1,
    xsecs={
        13: Number(0.5071, {
            "scale": (REL, 0.058, 0.092),  # relative scale uncertainty of +5.8/-9.2 %
            "pdf"  : (REL, 0.036),         # relative pdf uncertainty of +-3.6 %
        }),
    },
   label=r"t\bar{t}H",
   color=(255, 0, 0),
)

# print the ttH cross section at 13 TeV with the up-shifted scale uncertainty
print(ttH.get_xsec(13)(UP, "scale"))
# -> 0.5365118

print(ttH.get_xsec(13).__class__)
# -> "scinum.Number"

# add the ttH (H -> bb) subprocess
ttH_bb = ttH.add_process("ttH_bb", 2,
    xsecs={
        13: ttH.get_xsec(13) * 0.5824,  # branching ratio of H -> bb
    },
    label=ttH.label + r", H \rightarrow b\bar{b}",
)

# again, print the cross section for the up-shfited uncertainty, note the correct␣
↪propagation
```

(continues on next page)

(continued from previous page)

```
print(ttH_bb.get_xsec(13)(UP, "scale"))
# -> 0.3124645

# print the label in ROOT-style latex
print(ttH_bb.label_root)
# -> "t#bar{t}H, H #rightarrow b#bar{b}"

# check that the subprocess is really contained in the ttH subprocesses
print("ttH_bb" in ttH.processes)
# -> True
```

Information about datasets is stored in *Dataset* objects. Standard attributes are *name* and *id*, labels, and data/MC information. Optionally, a dataset can be assigned to a *Campaign* and to one or more *Process* objects. Let's extend the above example:

```
dataset_ttH_bb = od.Dataset("ttH_bb", 1,
    campaign=campaign_2018,
    processes=[ttH_bb],
    n_files=1000,
)

# the campaign is now aware of this dataset
print(dataset_ttH_bb in campaign_2018.datasets)
# -> True

# and the dataset knows about the ttH_bb process
print(ttH_bb in dataset_ttH_bb.processes)
# -> True

# as a little exercise, get all ttH subprocesses which are contained in the dataset
# this is, of course, only the ttH_bb process itself
print([p for p in ttH.processes if p in dataset_ttH_bb.processes])
# -> ["<Dataset at 0x1169421d0, name=ttH_bb, id=1, context=dataset>"]

# print the number of files
print(datasaet_ttH_bb.n_files)
# -> 1000
```

The last statement prints the number of files in that dataset. But what happens when systematic variations exist for that dataset? Let's assume there are two variants that were generated with different top quark masses. Do we create two additional datasets? **No**. They are stored in the same dataset object.

A dataset stores *DatasetInfo* objects, containing information that may vary (e.g.) across systematic uncertainties. Examples are the number of files, the number of total events, or arbitrary auxiliary information (see the *AuxDataMixin* in the mixin classes below). In fact, the number of files `n_files` above is already stored in a *DatasetInfo* object, stored as `dataset_ttH_bb.info["nominal"]`. The attributes `n_files` and `n_events` of the *nominal* info object are forwarded to the dataset object itself. Say the dataset with the up variation of the top mass has 300 files. We can extend the dataset above retrospectively

```
dataset_ttH_bb.set_info("mtop_up", od.DatasetInfo(
    n_files=300,
    aux=dict(mtop=173.5),
))
```

or directly in the constructor

```
dataset_ttH_bb = od.Dataset("ttH_bb", 1,
    campaign=campaign_2018,
    processes=[ttH_bb],
    info={
        "nominal": dict(n_files=1000, mtop=172.5),
        "mtop_up": dict(n_files=300, mtop=173.5),
    },
)
```

Note that the dictionaries passed in `info` are automatically converted to a *DatasetInfo* objects, and are accessible on the dataset itself via items (`__getitem__`). Also, the example shows how to use the auxiliary data storage capabilities, that most objects in *order* provide.

```
# print the number of files in the dataset with the up-varied top quark mass
print(dataset_ttH_bb["mtop_up"].n_files)
# -> 300

# also, print the respective top quark mass itself
print(dataset_ttH_bb["mtop_up"].get_aux("mtop"))
# -> 173.5
```

1.1.4 *Channel* and *Category*

The typical nomenclature for distinguishing between phase space regions comprises *channels* and *categories*. The distinction between them is often somewhat arbitrary and may vary from analysis to analysis. A channel often refers to a very distinct event / data signature and when combining analyses, multiple of these channels are usually merged. In this scenario, a category describes a sub phase space of events *within* a channel.

order introduces the *Channel* and *Category* classes. However, as the definition above might not apply to all use cases, they can be used quite independently. They have a simple difference: while categories can have selection strings, channels cannot. This distinction might appear marginal but in some cases it turned out to be very helpful.

Categories can be (optionally) assigned to a channel. Likewise, categories are nestable:

```
import order as od

# create a channel
channel_e = od.Channel("e", 1,
    title="Single electron",
)

# create a category
category_4j = od.Category("4j", 1,
    channel=channel_e,
    selection="nJets == 4",
    title="4 jets",
)

# now, the channel knows about the category and vice versa
print(category_4j in channel_e.categories)
# -> True
```

(continues on next page)

(continued from previous page)

```
# print the full category label
print(category_4j.full_label)
# -> "Single electron, 4 jets"

# now, add a subcategory
category_4j2b = category_4j.add_category("4j2b", 2,
    channel=channel_e,
    selection=od.util.join_root_selection(category_4j.selection, "nBTags == 2"),
    title=category_4j.title + ", 2 b-tags",
)

# print the selection string
print(category_4j2b.selection)
# -> "Single electron, 4 jets"

# print the full category label
print(category_4j2b.full_label)
# -> "Single electron, 4 jets, 2 b-tags"
```

1.1.5 *Shift*

A *Shift* object can be used to describe a systematic uncertainty. Its name must obey a simple naming scheme: either it is `nominal` or it has the format `<source>_<direction>` where *source* can be an arbitrary string and *direction* is either `"up"` or `"down"`. Also, a shift can have a type such as `Shift.RATE`, `Shift.SHAPE`, or `Shift.RATE_SHAPE` to signify exclusive rate- or shape-changing effects, or both. As usual, shifts are unqique objects.

```
import order as od

pdf_up = od.Shift("pdf_up", 1, type=Shift.SHAPE)

# print some properties
print(pdf_up.name)
# -> "pdf_up"

print(pdf_up.source)
# -> "pdf"

print(pdf_up.direction)
# -> "up"

print(pdf_up.is_down)
# -> False

# "nominal" has some special behavior
nom = od.Shift("nominal")
print(nom.name)
print(nom.source)
print(nom.direction)
# 3 x -> "nominal"
```

1.1.6 *Variable*

A *Variable* is supposed to provide convenience for plotting purposes. Essentially, it stores a variable expression, additional selection strings (especially useful in conjunction with categories, see above), binning helpers, axis titles, and unit information. Here are some examples:

```
import order as od

v = od.Variable("myVar",
    expression="myBranchA * myBranchB",
    selection="myBranchC > 0",
    binning=(20, 0., 10.),
    x_title=r"$\mu p_{T}$",
    unit="GeV",
)

# access and print some attributes
print(v.expression)
# -> "myBranchA * myBranchB"

print(v.n_bins)
# -> 20

print(v.even_binning)
# -> True

print(v.x_title_root)
# -> "#mu p_{T}"

print(v.full_title())
# -> "myVar;$\mu p_{T}$" / GeV;Entries / 0.5 GeV'"

# add further selections
v.add_selection("myBranchD < 0", op="&&")
print(v.selection)
# -> "(myBranchC > 0) && (myBranchD < 0)"

v.add_selection("myBranchE < 5", op="||")
print(v.selection)
# -> "((myBranchC > 0) && (myBranchD < 0)) || (myBranchE < 5)"

# change the binning
v.binning = [0., 1., 5., 7., 9., 10.]
print(v.even_binning)
# -> False

print(v.n_bins)
# -> 5
```

Variables are unique objects. However, no *id* was set in the example above. This is because variables make use of the *auto id* mechanism. The default id in the variable constructor is `Variable.AUTO_ID` (or simply "+"), which tells order to automatically use an id that was not used before (usually the maximum of the currently used ids plus one).

1.1.7 Mixin classes

Within *order*, common functionality is implemented in so-called mixin classes in the *order.mixins* module. Examples are the handling of auxiliary data, labels, data sources (data or MC), selection strings, etc. Most classes inherit from one or (often) more mixin classes listed below.

- *CopyMixin*: Adds copy functionality.
- *AuxDataMixin*: Adds storage and access to auxiliary data.
- *TagMixin*: Adds tagging capabilities.
- *DataSourceMixin*: Adds *is_data* and *is_mc* flags.
- *SelectionMixin*: Adds selection string handling.
- *LabelMixin*: Adds labeling.
- *ColorMixin*: Adds attributes for configuring colors.

1.1.8 Copying objects

Most classes inherit from the *CopyMixin*. As a result, instances can be copied with Python's builtin `copy.copy` and `copy.deepcopy` methods as well as with an additional *copy()* method defined on the mixin class itself.

The use of the latter is recommended as it provides better control over the copy behavior, and in fact, both `copy.copy` and `copy.deepcopy` simply call *copy()* without arguments (and thus, have the identical outcome). When using *copy()*, you can pass keyword arguments to configure / overwrite certain attributes of the copied object instead of copying them from the original one.

The copy mechanism can be demonstrated using *Variable*'s.

```
import order as od

jet1_pt = od.Variable("jet1_pt", 1,  # explict id
    expression="jet_pt[0]",
    unit="GeV",
    binning=[40, 0., 400.],
    x_title=r"Leading jet p_{T}",
    x_title_short=r"jet1 p_{T}",
    log_y=True,
    tags={"jet_variable"},
)

# copy the variable and add a selection for high pt regimes
# attention: variables are unique objects, so we explicitely need to change
# both name and id if we place the copied version into the same unqiqueness context
jet1_pt_high = jet1_pt.copy("jet1_pt_high", 2,
   selection="jet_pt[0] > 200",
)

# now, copy the same variable, but this time change the uniqueness context,
# so there is no need to set a different name of id
# additional, skip copying the "tags" attribute
with od.uniqueness_context("untagged"):
    jet1_pt_untagged = jet1_pt.copy(_skip=["tags"])
```

(continues on next page)

(continued from previous page)

```
print(jet1_pt.has_tag("jet_variable"))
# -> True

print(jet1_pt_untagged.has_tag("jet_variable"))
# -> False
```

Checkout the API reference of the specific classes to find detailed notes on their copy behavior.

1.2 API Reference

1.2.1 order.analysis

Definition of the central analysis class.

Contents

- *order.analysis*
  - *Class* Analysis

Class *Analysis*

class Analysis(**args*, ***kwargs*)

Bases: *UniqueObject*, *AuxDataMixin*, *TagMixin*

The analysis class which represents the central object of a physics analysis. Yet, it is quite lightweight as most information is contained in *Config* objects in the scope of a *Campaign*. In addition, it provides some convenience methods to directly access objects in deeper data structures.

Arguments

The configuration objects are initialized with *configs*. *tags* are forwarded to the *TagMixin*, *aux* to the *AuxDataMixin*, and *name*, *id* and *context* to the *UniqueObject* constructor.

Example

For usage examples, see the examples directory.

Members

configs

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of child configs.

get_channels(*config*)

Returns the *UniqueObjectIndex* of channels that belong to a *config* that was previously added. Shorthand for `get_config(config).channels`.

get_categories(*config*)

Returns the *UniqueObjectIndex* of categories that belong to a *config* that was previously added. Shorthand for `get_config(config).categories`.

get_datasets(*config*)

Returns the *UniqueObjectIndex* of datasets that belong to a *config* that was previously added. Shorthand for `get_config(config).datasets`.

get_processes(*config*)

Returns the *UniqueObjectIndex* of processes that belong to a *config* that was previously added. Shorthand for `get_config(config).processes`.

get_variables(*config*)

Returns the *UniqueObjectIndex* of variables that belong to a *config* that was previously added. Shorthand for `get_config(config).variables`.

get_shifts(*config*)

Returns the *UniqueObjectIndex* of shifts that belong to a *config* that was previously added. Shorthand for `get_config(config).shifts`.

add_config(**args*, ***kwargs*)

Adds a child config. See *order.unique.UniqueObjectIndex.add()* for more info. Also sets the analysis of the added config to *this* instance. Returns the added config object.

remove_config(**args*, ***kwargs*)

Removes a child config. See *order.unique.UniqueObjectIndex.remove()* for more info. Also resets the analysis of the added config. Returns the removed config object.

clear_configs(*context=None*)

Removes all child configs from the *configs* index for *context*. When *context* is *None*, the *default_context* of the *configs* index is used.

extend_configs(*objs*, *context=None*)

Adds multiple child configs to the *configs* index for *context* and returns the added objects in a list. When *context* is *None*, the *default_context* of the *configs* index is used.

get_config(*obj*, *default=no_default*, *context=None*)

Returns a child config given by *obj*, which might be a *name*, *id*, or an instance from the *configs* index for *context*. When no config is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the *configs* index is used.

has_config(*obj*, *context=None*)

Checks if the *configs* index for *context* contains an *obj* which might be a *name*, *id*, or an instance. When *context* is *None*, the *default_context* of the *configs* index is used.

property has_configs

Returns *True* when this config has child configs, *False* otherwise.

property is_leaf_config

Returns *True* when this config has no child configs, *False* otherwise.

1.2.2 order.config

Definition of a data-taking campaign and the connection of its information to an analysis within a config.

Contents

- *order.config*
  - *Class* Campaign
  - *Class* Config

Class *Campaign*

class Campaign(**args*, ***kwargs*)

Bases: *UniqueObject*, *CopyMixin*, *AuxDataMixin*, *TagMixin*

Class that provides data that is subject to a campaign, i.e., a well-defined range of data-taking, detector alignment, MC production settings, datasets, etc. Common, generic information is available via dedicated attributes, specialized data can be stored as auxiliary data.

Arguments

ecm is the center-of-mass energy, *bx* the bunch-crossing. *tags* are forwarded to the *TagMixin*, *aux* to the *AuxDataMixin*, *name*, *id* and *context* to the *UniqueObject* constructor.

Copy behavior

All attributes are copied **except** for references to contained datasets. Also note the copy behavior of *UniqueObject*'s.

Example

```
import order as od

c = od.Campaign("2017B", 1,
    ecm=13,
    bx=25,
)

d = c.add_dataset("ttH", 1)

d in c.datasets
# -> True

d.campaign == c
# -> True
```

Members

ecm

type: float

The center-of-mass energy in arbitrary units.

bx

type: float

The bunch crossing in arbitrary units.

datasets

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of child datasets.

add_dataset(**args*, ***kwargs*)

Adds a child dataset and returns it. See `UniqueObjectIndex.add()` for more info. Also sets the *campaign* of the added dataset to *this* instance.

remove_dataset(**args*, ***kwargs*)

Removes a child dataset. See `UniqueObjectIndex.remove()` for more info. Also resets the *campaign* of the added dataset.

clear_datasets(*context=None*)

Removes all child datasets from the *datasets* index for *context*. When *context* is *None*, the *default_context* of the *datasets* index is used.

extend_datasets(*objs*, *context=None*)

Adds multiple child datasets to the *datasets* index for *context* and returns the added objects in a list. When *context* is *None*, the *default_context* of the *datasets* index is used.

get_dataset(*obj*, *default=no_default*, *context=None*)

Returns a child dataset given by *obj*, which might be a *name*, *id*, or an instance from the *datasets* index for *context*. When no dataset is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the *datasets* index is used.

has_dataset(*obj*, *context=None*)

Checks if the *datasets* index for *context* contains an *obj* which might be a *name*, *id*, or an instance. When *context* is *None*, the *default_context* of the *datasets* index is used.

property has_datasets

Returns *True* when this dataset has child datasets, *False* otherwise.

property is_leaf_dataset

Returns *True* when this dataset has no child datasets, *False* otherwise.

Class *Config*

class Config(**args*, ***kwargs*)

Bases: *UniqueObject*, *CopyMixin*, *AuxDataMixin*, *TagMixin*

Class holding analysis information that is related to a *Campaign* instance. Most of the analysis configuration happens here.

Besides references to the *Analysis* and *Campaign* instances it belongs to, it stores analysis *datasets*, *processes*, *channels*, *categories*, *variables*, and *shifts* in *UniqueObjectIndex* instances.

Arguments

datasets, *processes*, *channels*, *categories*, *variables*, and *shifts* are initialized from constructor arguments. *name*, *id* and *context* are forwarded to the *UniqueObject* constructor. *name* and *id* default to the values of the *campaign* instance. Specialized data such as integrated luminosities, triggers, etc, can be stored as auxiliary data *aux*, which are forwarded to the *AuxDataMixin*. *tags* are forwarded to the *TagMixin*.

Copy behavior

Only *name*, *id*, *context*, auxiliary data, and references to the *analysis* and *campaign* instances are copied. Datasets, processes, channels, categories, variables and shifts are not copied! This is due to the fact that order does not try to auto-magically guess wich exact copy behavior is desired by the user. Also note the copy behavior of *UniqueObject*'s.

Example

```
import order as od

analysis = od.Analysis("ttH", 1)
campaign = od.Campaign("data_taking_2018", 1)

# add the campaign to the analysis, which returns a new config
cfg = analysis.add_config(campaign)

cfg.name, cfg.id
# -> "data_taking_2019", 1

# start configuration
cfg.add_dataset(campaign.get_dataset("ttH_bb"))

cfg.add_process("ttH_bb", 1, xsecs={13: 0.5071})

bb = cfg.add_channel("bb", 1)

bb.add_category("eq6j_eq4b")

cfg.add_variable("jet1_px", expression="jet1_pt * cos(jet1_phi)")

cfg.add_shift("pdf_up", type=Shift.SHAPE)
...

# at some point you might want to create a second config
# with other values for that campaign, e.g. for sub-measurements
cfg2 = analysis.add_config(campaign, name="sf_meausurement", id=2)
...
```

Members

campaign

type: Campaign

read-only

The *Campaign* instance this config belongs to.

analysis

type: Analysis

read-only

The *Analysis* instance this config belongs to. When set, *this* config is added to the index of configs of the analysis object.

shifts

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of child shifts.

variables

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of child variables.

categories

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of child categories.

channels

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of child channels.

processes

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of child processes.

datasets

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of child datasets.

add_category(**args*, ***kwargs*)

Adds a child category to the *categories* index and returns it. See `UniqueObjectIndex.add()` for more info.

add_channel(**args*, ***kwargs*)

Adds a child channel to the *channels* index and returns it. See `UniqueObjectIndex.add()` for more info.

add_dataset(**args*, ***kwargs*)

Adds a child dataset to the *datasets* index and returns it. See `UniqueObjectIndex.add()` for more info.

add_process(**args*, ***kwargs*)

Adds a child process to the *processes* index and returns it. See `UniqueObjectIndex.add()` for more info.

add_shift(**args*, ***kwargs*)

Adds a child shift to the *shifts* index and returns it. See `UniqueObjectIndex.add()` for more info.

add_variable(**args*, ***kwargs*)

Adds a child variable to the *variables* index and returns it. See `UniqueObjectIndex.add()` for more info.

clear_categories(*context=None*)

Removes all child categories from the *categories* index for *context*. When *context* is *None*, the *default_context* of the *categories* index is used.

clear_channels(*context=None*)

Removes all child channels from the `channels` index for *context*. When *context* is *None*, the *default_context* of the `channels` index is used.

clear_datasets(*context=None*)

Removes all child datasets from the `datasets` index for *context*. When *context* is *None*, the *default_context* of the `datasets` index is used.

clear_processes(*context=None*)

Removes all child processes from the `processes` index for *context*. When *context* is *None*, the *default_context* of the `processes` index is used.

clear_shifts(*context=None*)

Removes all child shifts from the `shifts` index for *context*. When *context* is *None*, the *default_context* of the `shifts` index is used.

clear_variables(*context=None*)

Removes all child variables from the `variables` index for *context*. When *context* is *None*, the *default_context* of the `variables` index is used.

extend_categories(*objs*, *context=None*)

Adds multiple child categories to the `categories` index for *context* and returns the added objects in a list. When *context* is *None*, the *default_context* of the `categories` index is used.

extend_channels(*objs*, *context=None*)

Adds multiple child channels to the `channels` index for *context* and returns the added objects in a list. When *context* is *None*, the *default_context* of the `channels` index is used.

extend_datasets(*objs*, *context=None*)

Adds multiple child datasets to the `datasets` index for *context* and returns the added objects in a list. When *context* is *None*, the *default_context* of the `datasets` index is used.

extend_processes(*objs*, *context=None*)

Adds multiple child processes to the `processes` index for *context* and returns the added objects in a list. When *context* is *None*, the *default_context* of the `processes` index is used.

extend_shifts(*objs*, *context=None*)

Adds multiple child shifts to the `shifts` index for *context* and returns the added objects in a list. When *context* is *None*, the *default_context* of the `shifts` index is used.

extend_variables(*objs*, *context=None*)

Adds multiple child variables to the `variables` index for *context* and returns the added objects in a list. When *context* is *None*, the *default_context* of the `variables` index is used.

get_category(*obj*, *deep=True*, *default=no_default*, *context=None*)

Returns a child category given by *obj*, which might be a *name*, *id*, or an instance from the `categories` index for *context*. If *deep* is *True*, the lookup is recursive. When no category is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the `categories` index is used.

get_channel(*obj*, *deep=True*, *default=no_default*, *context=None*)

Returns a child channel given by *obj*, which might be a *name*, *id*, or an instance from the `channels` index for *context*. If *deep* is *True*, the lookup is recursive. When no channel is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the `channels` index is used.

get_dataset(*obj*, *default=no_default*, *context=None*)

Returns a child dataset given by *obj*, which might be a *name*, *id*, or an instance from the `datasets` index for *context*. When no dataset is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the `datasets` index is used.

get_leaf_categories(*context=None*)

Returns all child categories from the `categories` index for *context* that have no child categories themselves in a recursive fashion. When *context* is *None*, the *default_context* of the `categories` index is used.

get_leaf_channels(*context=None*)

Returns all child channels from the `channels` index for *context* that have no child channels themselves in a recursive fashion. When *context* is *None*, the *default_context* of the `channels` index is used.

get_leaf_processes(*context=None*)

Returns all child processes from the `processes` index for *context* that have no child processes themselves in a recursive fashion. When *context* is *None*, the *default_context* of the `processes` index is used.

get_process(*obj*, *deep=True*, *default=no_default*, *context=None*)

Returns a child process given by *obj*, which might be a *name*, *id*, or an instance from the `processes` index for *context*. If *deep* is *True*, the lookup is recursive. When no process is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the `processes` index is used.

get_shift(*obj*, *default=no_default*, *context=None*)

Returns a child shift given by *obj*, which might be a *name*, *id*, or an instance from the `shifts` index for *context*. When no shift is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the `shifts` index is used.

get_variable(*obj*, *default=no_default*, *context=None*)

Returns a child variable given by *obj*, which might be a *name*, *id*, or an instance from the `variables` index for *context*. When no variable is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the `variables` index is used.

property has_categories

Returns *True* when this category has child categories, *False* otherwise.

has_category(*obj*, *deep=True*, *context=None*)

Checks if the `categories` index for *context* contains an *obj* which might be a *name*, *id*, or an instance. If *deep* is *True*, the lookup is recursive. When *context* is *None*, the *default_context* of the `categories` index is used.

has_channel(*obj*, *deep=True*, *context=None*)

Checks if the `channels` index for *context* contains an *obj* which might be a *name*, *id*, or an instance. If *deep* is *True*, the lookup is recursive. When *context* is *None*, the *default_context* of the `channels` index is used.

property has_channels

Returns *True* when this channel has child channels, *False* otherwise.

has_dataset(*obj*, *context=None*)

Checks if the `datasets` index for *context* contains an *obj* which might be a *name*, *id*, or an instance. When *context* is *None*, the *default_context* of the `datasets` index is used.

property has_datasets

Returns *True* when this dataset has child datasets, *False* otherwise.

has_process(*obj*, *deep=True*, *context=None*)

Checks if the `processes` index for *context* contains an *obj* which might be a *name*, *id*, or an instance. If *deep* is *True*, the lookup is recursive. When *context* is *None*, the *default_context* of the `processes` index is used.

property has_processes

Returns *True* when this process has child processes, *False* otherwise.

has_shift(*obj*, *context=None*)

Checks if the `shifts` index for *context* contains an *obj* which might be a *name*, *id*, or an instance. When *context* is *None*, the *default_context* of the `shifts` index is used.

property has_shifts

Returns *True* when this shift has child shifts, *False* otherwise.

has_variable(*obj*, *context=None*)

Checks if the `variables` index for *context* contains an *obj* which might be a *name*, *id*, or an instance. When *context* is *None*, the *default_context* of the `variables` index is used.

property has_variables

Returns *True* when this variable has child variables, *False* otherwise.

property is_leaf_category

Returns *True* when this category has no child categories, *False* otherwise.

property is_leaf_channel

Returns *True* when this channel has no child channels, *False* otherwise.

property is_leaf_dataset

Returns *True* when this dataset has no child datasets, *False* otherwise.

property is_leaf_process

Returns *True* when this process has no child processes, *False* otherwise.

property is_leaf_shift

Returns *True* when this shift has no child shifts, *False* otherwise.

property is_leaf_variable

Returns *True* when this variable has no child variables, *False* otherwise.

remove_category(*obj*, *context=None*, *silent=False*)

Removes a child category given by *obj*, which might be a *name*, *id*, or an instance from the `categories` index for *context* and returns the removed object. When *context* is *None*, the *default_context* of the `categories` index is used. Unless *silent* is *True*, an error is raised if the object was not found. See `UniqueObjectIndex.remove()` for more info.

remove_channel(*obj*, *context=None*, *silent=False*)

Removes a child channel given by *obj*, which might be a *name*, *id*, or an instance from the `channels` index for *context* and returns the removed object. When *context* is *None*, the *default_context* of the `channels` index is used. Unless *silent* is *True*, an error is raised if the object was not found. See `UniqueObjectIndex.remove()` for more info.

remove_dataset(*obj*, *context=None*, *silent=False*)

Removes a child dataset given by *obj*, which might be a *name*, *id*, or an instance from the `datasets` index for *context* and returns the removed object. When *context* is *None*, the *default_context* of the `datasets` index is used. Unless *silent* is *True*, an error is raised if the object was not found. See `UniqueObjectIndex.remove()` for more info.

remove_process(*obj*, *context=None*, *silent=False*)

Removes a child process given by *obj*, which might be a *name*, *id*, or an instance from the `processes` index for *context* and returns the removed object. When *context* is *None*, the *default_context* of the `processes` index is used. Unless *silent* is *True*, an error is raised if the object was not found. See `UniqueObjectIndex.remove()` for more info.

remove_shift(*obj*, *context=None*, *silent=False*)

Removes a child shift given by *obj*, which might be a *name*, *id*, or an instance from the `shifts` index for *context* and returns the removed object. When *context* is *None*, the *default_context* of the `shifts` index is used. Unless *silent* is *True*, an error is raised if the object was not found. See `UniqueObjectIndex.remove()` for more info.

remove_variable(*obj*, *context=None*, *silent=False*)

Removes a child variable given by *obj*, which might be a *name*, *id*, or an instance from the `variables` index for *context* and returns the removed object. When *context* is *None*, the *default_context* of the `variables` index is used. Unless *silent* is *True*, an error is raised if the object was not found. See `UniqueObjectIndex.remove()` for more info.

walk_categories(*context=None*, *depth_first=False*, *include_self=False*)

Walks through the `categories` index for *context* and per iteration, yields a child category, its depth relative to *this* category, and its child categories in a list that can be modified to alter the walking. When *context* is *None*, the *default_context* of the `categories` index is used. When *context* is *all*, all indices are traversed. When *depth_first* is *True*, iterate depth-first instead of the default breadth-first. When *include_self* is *True*, also yield this category instance with a depth of 0.

walk_channels(*context=None*, *depth_first=False*, *include_self=False*)

Walks through the `channels` index for *context* and per iteration, yields a child channel, its depth relative to *this* channel, and its child channels in a list that can be modified to alter the walking. When *context* is *None*, the *default_context* of the `channels` index is used. When *context* is *all*, all indices are traversed. When *depth_first* is *True*, iterate depth-first instead of the default breadth-first. When *include_self* is *True*, also yield this channel instance with a depth of 0.

walk_processes(*context=None*, *depth_first=False*, *include_self=False*)

Walks through the `processes` index for *context* and per iteration, yields a child process, its depth relative to *this* process, and its child processes in a list that can be modified to alter the walking. When *context* is *None*, the *default_context* of the `processes` index is used. When *context* is *all*, all indices are traversed. When *depth_first* is *True*, iterate depth-first instead of the default breadth-first. When *include_self* is *True*, also yield this process instance with a depth of 0.

1.2.3 order.dataset

Classes to define datasets.

Contents

- *order.dataset*
  - *Class* Dataset
  - *Class* DatasetInfo

Class *Dataset*

class Dataset(**args*, ***kwargs*)

Bases: *UniqueObject*, *CopyMixin*, *AuxDataMixin*, *TagMixin*, *DataSourceMixin*, *LabelMixin*

Dataset definition providing two kinds of information:

1. (systematic) shift-dependent, and
2. shift-indepent information.

Independent is e.g. whether or not it contains real data, whereas shift-dependent information is e.g. the number of events in the *nominal* or a *shifted* variation. Latter information is contained in *DatasetInfo* objects that are stored in *this* class and mapped to strings. These info objects can be accessed via *get_info()* or via items (*__getitem__*). For convenience, some of the properties of the *nominal* *DatasetInfo* object are accessible on this class via forwarding.

Arguments

A dataset is always measured in (real data) / created for (MC) a dedicated *campaign*, therefore it *belongs* to a *Campaign* object. In addition, physics *processes* can be *linked* to a dataset, therefore it *has* *Process* objects.

When *info* is does not contain a nominal *DatasetInfo* object (mapped to the key *order.shift.Shift.NOMINAL*, i.e., "nominal"), all *kwargs* are used to create one. Otherwise, it should be a dictionary matching the format of the *info* mapping. *label* and *label_short* are forwarded to the *LabelMixin*, *is_data* to the *DataSourceMixin*, *tags* to the *TagMixin*, *aux* to the *AuxDataMixin*, and *name*, *id* and *context* to the *UniqueObject* constructor.

Copy behavior

All attributes are copied **except** for references to linked processes. The *campaign* reference is kept. Also note the copy behavior of *UniqueObject*'s.

Example

```
import order as od

campaign = od.Campaign("2017B", 1, ...)

d = od.Dataset("ttH_bb", 1,
    campaign=campaign,
    keys=["/ttHTobb_M125.../.../..."],
    n_files=123,
    n_events=456789,
)

d.info.keys()
# -> ["nominal"]

d["nominal"].n_files
# -> 123

d.n_files
# -> 123

# similar to above, but set explicit info objects
d = Dataset("ttH_bb", 1,
    campaign=campaign,
```

(continues on next page)

(continued from previous page)

```
    info={
        "nominal": {
            "keys": ["/ttHTobb_M125.../.../..."],
            "n_files": 123,
            "n_events": 456789,
        },
        "scale_up": {
            "keys": ["/ttHTobb_M125_scaleUP.../.../..."],
            "n_files": 100,
            "n_events": 40000,
        },
    },
)

d.info.keys()
# -> ["nominal", "scale_up"]

d["nominal"].n_files
# -> 123

d.n_files
# -> 123

d["scale_up"].n_files
# -> 100
```

Members

campaign

type: Campaign, None

The *Campaign* object this dataset belongs to. When set, *this* dataset is also added to the dataset index of the campaign object.

info

type: dictionary

Mapping of shift names to *DatasetInfo* instances.

keys

type: list

read-only

The dataset keys of the nominal *DatasetInfo* object.

n_files

type: integer

read-only

The number of files of the nominal *DatasetInfo* object.

n_events

type: integer

read-only

The number of events of the nominal *DatasetInfo* object.

processes

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of child processes.

set_info(*shift_name*, *info*)

Sets an *DatasetInfo* object *info* for a given *shift_name*. Returns the object.

get_info(*shift_name*)

Returns the *DatasetInfo* object for a given *shift_name*.

add_process(**args*, ***kwargs*)

Adds a child process to the *processes* index and returns it. See `UniqueObjectIndex.add()` for more info.

clear_processes(*context=None*)

Removes all child processes from the *processes* index for *context*. When *context* is *None*, the *default_context* of the *processes* index is used.

extend_processes(*objs*, *context=None*)

Adds multiple child processes to the *processes* index for *context* and returns the added objects in a list. When *context* is *None*, the *default_context* of the *processes* index is used.

get_leaf_processes(*context=None*)

Returns all child processes from the *processes* index for *context* that have no child processes themselves in a recursive fashion. When *context* is *None*, the *default_context* of the *processes* index is used.

get_process(*obj*, *deep=True*, *default=no_default*, *context=None*)

Returns a child process given by *obj*, which might be a *name*, *id*, or an instance from the *processes* index for *context*. If *deep* is *True*, the lookup is recursive. When no process is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the *processes* index is used.

has_process(*obj*, *deep=True*, *context=None*)

Checks if the *processes* index for *context* contains an *obj* which might be a *name*, *id*, or an instance. If *deep* is *True*, the lookup is recursive. When *context* is *None*, the *default_context* of the *processes* index is used.

property has_processes

Returns *True* when this process has child processes, *False* otherwise.

property is_leaf_process

Returns *True* when this process has no child processes, *False* otherwise.

remove_process(*obj*, *context=None*, *silent=False*)

Removes a child process given by *obj*, which might be a *name*, *id*, or an instance from the *processes* index for *context* and returns the removed object. When *context* is *None*, the *default_context* of the *processes* index is used. Unless *silent* is *True*, an error is raised if the object was not found. See `UniqueObjectIndex.remove()` for more info.

walk_processes(*context=None*, *depth_first=False*, *include_self=False*)

Walks through the *processes* index for *context* and per iteration, yields a child process, its depth relative to *this* process, and its child processes in a list that can be modified to alter the walking. When *context* is *None*, the *default_context* of the *processes* index is used. When *context* is *all*, all indices are traversed. When *depth_first* is *True*, iterate depth-first instead of the default breadth-first. When *include_self* is *True*, also yield this process instance with a depth of 0.

Class *DatasetInfo*

class DatasetInfo(*keys=None*, *n_files=- 1*, *n_events=- 1*, *tags=None*, *aux=None*)

Bases: *CopyMixin*, *AuxDataMixin*, *TagMixin*

Container class holding information on particular dataset variations. Instances of *this* class are typically used in *Dataset* objects to store shift-dependent information, such as the number of files or events for a particular shift (e.g. *nominal*, *scale_up*, etc).

Arguments

keys denote the identifiers or *origins* of a dataset. *n_files* and *n_events* can be used for further bookkeeping. *tags* are forwarded to the *TagMixin*, and *aux* to the *AuxDataMixin*.

Copy behavior

All attributes are copied. Also note the copy behavior of *UniqueObject*'s.

Members

keys

type: list

The dataset keys, e.g. `["/ttHTobb_M125.../.../..."]`.

n_files

type: integer

The number of files.

n_events

type: integer

The number of events.

1.2.4 order.process

Classes to define physics processes.

Contents

- *order.process*
  - *Class* Process

Class *Process*

class Process(**args*, ***kwargs*)

Bases: *UniqueObject*, *CopyMixin*, *AuxDataMixin*, *TagMixin*, *DataSourceMixin*, *LabelMixin*, *ColorMixin*

Definition of a phyiscs process.

Arguments

xsecs should be a mapping of center-of-mass energies to cross sections values (automatically converted to scinum.Number instances).

color is forwarded to the *ColorMixin*, *label* and *label_short* to the *LabelMixin*, *is_data* to the *DataSourceMixin*,*tags* to the *TagMixin*, *aux* to the *AuxDataMixin*, and *name*, *id* and *context* to the *UniqueObject* constructor.

A process can have parent-child relations to other processes. Initial child processes are set to *processes*.

Copy behavior

All attributes are copied **except** for references to child and parent processes. Also note the copy behavior of *UniqueObject*'s.

Example

```
import order as od
from scinum import Number, REL

p = od.Process("ttH", 1,
    xsecs={
        13: Number(0.5071, {"scale": (REL, 0.036)}),  # +-3.6% scale uncertainty
    },
    label=r"$t\bar{t}H$",
    color=(255, 0, 0),
)

p.get_xsec(13).str("%.2f")
# -> "0.51 +- 0.02 (scale)"

p.label_root
# -> "t#bar{t}H"

p2 = p.add_process("ttH_bb", 2,
    xsecs={
        13: p.get_xsec(13) * 0.5824,
    },
    label=p.label + r", $b\bar{b}$",
)

p2 == p.get_process("ttH_bb")
# -> True

p2.label_root
# -> "t#bar{t}H, b#bar{b}"

p2.has_parent_process("ttH")
# -> True
```

Members

xsecs

type: dictionary (float -> scinum.Number)

Cross sections mapped to a center-of-mass energies with arbitrary units.

processes

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of child processes.

parent_processes

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of parent processes.

classmethod pretty_print_all(**args*, *contenxt=None*, ***kwargs*)

Calls *pretty_print()* of all root processes in the instance cache for *context* and forwards all *args* and *kwargs*. When *context* is *all*, root processes of all indices are printed.

get_xsec(*ecm*)

Returns the cross section (a scinum.Number instance) for a center-of-mass energy *ecm*.

set_xsec(*ecm*, *xsec*)

Sets the cross section for a center-of-mass energy *ecm* to *xsec*. When *xsec* is not a scinum.Number instance, it is converted to one. The (probably converted) value is returned.

add_parent_process(**args*, ***kwargs*)

Adds a parent process to the *parent_processes* index and returns it. Also adds *this* process to the *processes* index of the added process. When *context* is *None*, the *default_context* of the *processes* index is used. See `UniqueObjectIndex.add()` for more info.

add_process(**args*, ***kwargs*)

Adds a child process to the *processes* index and returns it. Also adds *this* process to the *parent_processes* index of the added process. See `UniqueObjectIndex.add()` for more info.

clear_parent_processes(*context=None*)

Removes all parent processes from the *parent_processes* index for *context*. Also removes *this* process instance from the *processes* index of all removed process.

clear_processes(*context=None*)

Removes all child processes from the *processes* index for *context*. Also removes *this* process instance from the *parent_processes* index of all removed processes. When *context* is *None*, the *default_context* of the *processes* index is used

extend_parent_processes(*objs*, *context=None*)

Adds multiple parent processes to the *parent_processes* index for *context* and returns the added objects in a list. Also adds *this* process to the *processes* index of the added process. When *context* is *None*, the *default_context* of the *processes* index is used.

extend_processes(*objs*, *context=None*)

Adds multiple child processes to the *processes* index for *context* and returns the added objects in a list. Also adds *this* process to the *parent_processes* index of the added process. When *context* is *None*, the *default_context* of the *processes* index is used.

get_leaf_processes(*context=None*)

Returns all child processes from the *processes* index for *context* that have no child processes themselves in a recursive fashion. When *context* is *None*, the *default_context* of the *processes* index is used.

get_parent_process(*obj*, *deep=True*, *default=no_default*, *context=None*)

Returns a parent process given by *obj*, which might be a *name*, *id*, or an instance from the *parent_processes* index for *context*. If *deep* is *True*, the lookup is recursive. When no process is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the *parent_processes* index is used.

get_process(*obj*, *deep=True*, *default=no_default*, *context=None*)

Returns a child process given by *obj*, which might be a *name*, *id*, or an instance from the `processes` index for *context*. If *deep* is *True*, the lookup is recursive. When no process is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the `processes` index is used.

get_root_processes(*context=None*)

Returns all parent processes from the `parent_processes` index for *context* that have no parent processes themselves in a recursive fashion. When *context* is *None*, the *default_context* of the `parent_processes` index is used.

has_parent_process(*obj*, *deep=True*, *context=None*)

Checks if the `parent_processes` index for *context* contains an *obj*, which might be a *name*, *id*, or an instance. If *deep* is *True*, the lookup is recursive. When *context* is *None*, the *default_context* of the `parent_processes` index is used.

property has_parent_processes

Returns *True* when this process has parent processes, *False* otherwise.

has_process(*obj*, *deep=True*, *context=None*)

Checks if the `processes` index for *context* contains an *obj* which might be a *name*, *id*, or an instance. If *deep* is *True*, the lookup is recursive. When *context* is *None*, the *default_context* of the `processes` index is used.

property has_processes

Returns *True* when this process has child processes, *False* otherwise.

property is_leaf_process

Returns *True* when this process has no child processes, *False* otherwise.

property is_root_process

Returns *True* when this process has no parent processes, *False* otherwise.

pretty_print(*ecm=None*, *offset=40*, *max_depth=- 1*, *stream=None*, ***kwargs*)

Prints this process and potentially its subprocesses down to a maximum depth *max_depth* in a structured fashion. When *ecm* is set, process cross section values are shown as well with a maximum horizontal distance of *offset*. *stream* can be a file object and defaults to *sys.stdout*. All *kwargs* are forwarded to the `Number.str()` methods of the cross section numbers.

remove_parent_process(*obj*, *context=None*, *silent=False*)

Removes a parent process *obj* which might be a *name*, *id*, or an instance from the `parent_processes` index for *context*. Also removes *this* instance from the `processes` index of the removed parent process. Returns the removed object. When *context* is *None*, the *default_context* of the `parent_processes` index is used. Unless *silent* is *True*, an error is raised if the object was not found. See `UniqueObjectIndex.remove()` for more info.

remove_process(*obj*, *context=None*, *silent=False*)

Removes a child process given by *obj*, which might be a *name*, *id*, or an instance from the `processes` index for *context* and returns the removed object. Also removes *this* process from the `parent_processes` index of the removed process. When *context* is *None*, the *default_context* of the `processes` index is used. Unless *silent* is *True*, an error is raised if the object was not found. See `UniqueObjectIndex.remove()` for more info.

walk_parent_processes(*context=None*, *depth_first=False*, *include_self=False*)

Walks through the `parent_processes` index for *context* and per iteration, yields a parent process, its depth relative to *this* process, and its parent processes in a list that can be modified to alter the walking. When *context* is *None*, the *default_context* of the `parent_processes` index is used. When *context* is *all*,

all indices are traversed. When *depth_first* is *True*, iterate depth-first instead of the default breadth-first. When *include_self* is *True*, also yield this process instance with a depth of 0.

walk_processes(*context=None*, *depth_first=False*, *include_self=False*)

Walks through the `processes` index for *context* and per iteration, yields a child process, its depth relative to *this* process, and its child processes in a list that can be modified to alter the walking. When *context* is *None*, the *default_context* of the `processes` index is used. When *context* is *all*, all indices are traversed. When *depth_first* is *True*, iterate depth-first instead of the default breadth-first. When *include_self* is *True*, also yield this process instance with a depth of 0.

1.2.5 order.category

Classes to describe object that help distinguishing events.

Contents

- *order.category*
  - *Class* Channel
  - *Class* Category

Class *Channel*

class Channel(**args*, ***kwargs*)

Bases: `UniqueObject`, `CopyMixin`, `AuxDataMixin`, `TagMixin`, `LabelMixin`

An object that descibes an analysis channel, often defined by a particular decay *channel* that results in distinct final state objects. A channel can have parent-child relations to other channels with one parent per child, and child relations to categories.

Arguments

References to contained categories are initialized with *categories*. *label* and *label_short* are passed to the `LabelMixin`, *tags* to the `TagMixin`, *aux* to the `AuxDataMixin`, and *name*, *id* and *context* to the `UniqueObject` constructor.

Copy behavior

All attributes are copied **except** for references to child channels and the parent channel as well as categories. Also note the copy behavior of `UniqueObject`'s.

Example

```
import order as od

# create a channel
SL_channel = od.Channel("SL", 1, label="lepton+jets")

# add child channels
e_channel  = SL_channel.add_channel("e", 1, label="e+jets")
mu_channel = SL_channel.add_channel("mu", 2)

len(SL_channel.channels)
```

(continues on next page)

(continued from previous page)

```
# -> 2

len(e_channel.parent_channels)
# -> 1

e_channel.parent_channel
# -> SL_channel

# add categories
cat_e_2j = e_channel.add_category("e_2j",
    label="2 jets",
    selection="nJets == 2",
)

# print the category label
cat_e_2j.full_label
# -> "e+jets, 2 jets"
```

Members

categories

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of child categories.

channels

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of child channels.

parent_channels

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of parent channels.

add_category(**args*, ***kwargs*)

Adds a child category. See `UniqueObjectIndex.add()` for more info. Also sets the *channel* of the added category to *this* instance.

add_channel(**args*, ***kwargs*)

Adds a child channel to the *channels* index and returns it. Also adds *this* channel to the *parent_channels* index of the added channel. An exception is raised when the number of allowed parents of a child channel is exceeded. See `UniqueObjectIndex.add()` for more info.

add_parent_channel(**args*, ***kwargs*)

Adds a parent channel to the *parent_channels* index and returns it. Also adds *this* channel to the *channels* index of the added channel. An exception is raised when the number of allowed parents is exceeded. When *context* is *None*, the *default_context* of the *channels* index is used. See `UniqueObjectIndex.add()` for more info.

clear_categories(*context=None*)

Removes all child categories from the *categories* index for *context*. When *context* is *None*, the *default_context* of the *categories* index is used.

clear_channels(*context=None*)

Removes all child channels from the *channels* index for *context*. Also removes *this* channel instance from the *parent_channels* index of all removed channels. When *context* is *None*, the *default_context* of the *channels* index is used

clear_parent_channels(*context=None*)

Removes all parent channels from the *parent_channels* index for *context*. Also removes *this* channel instance from the *channels* index of all removed channel.

extend_categories(*objs*, *context=None*)

Adds multiple child categories to the *categories* index for *context* and returns the added objects in a list. When *context* is *None*, the *default_context* of the *categories* index is used.

extend_channels(*objs*, *context=None*)

Adds multiple child channels to the *channels* index for *context* and returns the added objects in a list. Also adds *this* channel to the *parent_channels* index of the added channel. An exception is raised when the number of allowed parents of a child channel is exceeded. When *context* is *None*, the *default_context* of the *channels* index is used.

extend_parent_channels(*objs*, *context=None*)

Adds multiple parent channels to the *parent_channels* index for *context* and returns the added objects in a list. Also adds *this* channel to the *channels* index of the added channel. An exception is raised when the number of allowed parent channels is exceeded. When *context* is *None*, the *default_context* of the *channels* index is used.

get_category(*obj*, *deep=True*, *default=no_default*, *context=None*)

Returns a child category given by *obj*, which might be a *name*, *id*, or an instance from the *categories* index for *context*. If *deep* is *True*, the lookup is recursive. When no category is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the *categories* index is used.

get_channel(*obj*, *deep=True*, *default=no_default*, *context=None*)

Returns a child channel given by *obj*, which might be a *name*, *id*, or an instance from the *channels* index for *context*. If *deep* is *True*, the lookup is recursive. When no channel is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the *channels* index is used.

get_leaf_categories(*context=None*)

Returns all child categories from the *categories* index for *context* that have no child categories themselves in a recursive fashion. When *context* is *None*, the *default_context* of the *categories* index is used.

get_leaf_channels(*context=None*)

Returns all child channels from the *channels* index for *context* that have no child channels themselves in a recursive fashion. When *context* is *None*, the *default_context* of the *channels* index is used.

get_parent_channel(*obj*, *deep=True*, *default=no_default*, *context=None*)

Returns a parent channel given by *obj*, which might be a *name*, *id*, or an instance from the *parent_channels* index for *context*. If *deep* is *True*, the lookup is recursive. When no channel is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the *parent_channels* index is used.

get_root_channels(*context=None*)

Returns all parent channels from the *parent_channels* index for *context* that have no parent channels themselves in a recursive fashion. When *context* is *None*, the *default_context* of the *parent_channels* index is used.

property has_categories

Returns *True* when this category has child categories, *False* otherwise.

has_category(*obj*, *deep=True*, *context=None*)

Checks if the `categories` index for *context* contains an *obj* which might be a *name*, *id*, or an instance. If *deep* is *True*, the lookup is recursive. When *context* is *None*, the *default_context* of the `categories` index is used.

has_channel(*obj*, *deep=True*, *context=None*)

Checks if the `channels` index for *context* contains an *obj* which might be a *name*, *id*, or an instance. If *deep* is *True*, the lookup is recursive. When *context* is *None*, the *default_context* of the `channels` index is used.

property has_channels

Returns *True* when this channel has child channels, *False* otherwise.

has_parent_channel(*obj*, *deep=True*, *context=None*)

Checks if the `parent_channels` index for *context* contains an *obj*, which might be a *name*, *id*, or an instance. If *deep* is *True*, the lookup is recursive. When *context* is *None*, the *default_context* of the `parent_channels` index is used.

property has_parent_channels

Returns *True* when this channel has parent channels, *False* otherwise.

property is_leaf_category

Returns *True* when this category has no child categories, *False* otherwise.

property is_leaf_channel

Returns *True* when this channel has no child channels, *False* otherwise.

property is_root_channel

Returns *True* when this channel has no parent channels, *False* otherwise.

remove_channel(*obj*, *context=None*, *silent=False*)

Removes a child channel given by *obj*, which might be a *name*, *id*, or an instance from the `channels` index for *context* and returns the removed object. Also removes *this* channel from the `parent_channels` index of the removed channel. When *context* is *None*, the *default_context* of the `channels` index is used. Unless *silent* is *True*, an error is raised if the object was not found. See `UniqueObjectIndex.remove()` for more info.

remove_parent_channel(*obj=None*, *context=None*, *silent=False*)

Removes the parent channel *obj* the `parent_channels` index for *context*. When *obj* is not *None*, it can be a *name*, *id*, or an instance referring to the parent channel for validation purposes. Also removes *this* instance from the `channels` index of the removed parent channel. Returns the removed object. When *context* is *None*, the *default_context* of the `parent_channels` index is used. Unless *silent* is *True*, an error is raised if the object was not found. See `UniqueObjectIndex.remove()` for more info.

walk_categories(*context=None*, *depth_first=False*, *include_self=False*)

Walks through the `categories` index for *context* and per iteration, yields a child category, its depth relative to *this* category, and its child categories in a list that can be modified to alter the walking. When *context* is *None*, the *default_context* of the `categories` index is used. When *context* is *all*, all indices are traversed. When *depth_first* is *True*, iterate depth-first instead of the default breadth-first. When *include_self* is *True*, also yield this category instance with a depth of 0.

walk_channels(*context=None*, *depth_first=False*, *include_self=False*)

Walks through the `channels` index for *context* and per iteration, yields a child channel, its depth relative to *this* channel, and its child channels in a list that can be modified to alter the walking. When *context* is

None, the *default_context* of the `channels` index is used. When *context* is *all*, all indices are traversed. When *depth_first* is *True*, iterate depth-first instead of the default breadth-first. When *include_self* is *True*, also yield this channel instance with a depth of 0.

walk_parent_channels(*context=None*, *depth_first=False*, *include_self=False*)

Walks through the `parent_channels` index for *context* and per iteration, yields a parent channel, its depth relative to *this* channel, and its parent channels in a list that can be modified to alter the walking. When *context* is *None*, the *default_context* of the `parent_channels` index is used. When *context* is *all*, all indices are traversed. When *depth_first* is *True*, iterate depth-first instead of the default breadth-first. When *include_self* is *True*, also yield this channel instance with a depth of 0.

remove_category(**args*, ***kwargs*)

Removes a child category. See `UniqueObjectIndex.remove()` for more info. Also resets the *channel* of the added category.

Class *Category*

class Category(*name*, *id='+'*, *channel=None*, *categories=None*, *label=None*, *label_short=None*, *selection=None*, *selection_mode=None*, *tags=None*, *aux=None*, *context=None*)

Bases: `UniqueObject`, `CopyMixin`, `AuxDataMixin`, `TagMixin`, `SelectionMixin`, `LabelMixin`

Class that describes an analysis category. This is not to be confused with an analysis `Channel`. While the definition of a channel can be understood as being fixed by e.g. the final state of an event, a category describes an arbitrary sub phase-space. Therefore, a category can be (optionally) uniquely assigned to a channel - it *has* a channel.

Also, categories can be nested, i.e., they can have child and parent categories.

Arguments

channel should be a reference to a `Channel` instance or *None*. Child categories are initialized with *categories*.

label and *label_short* are forwarded to the `LabelMixin`, *selection* and *selection_mode* to the `SelectionMixin`, *tags* to the `TagMixin`, *aux* to the `AuxDataMixin`, and *name*, *id* (defaulting to an auto id) and *context* to the `UniqueObject` constructor.

Copy behavior

All attributes are copied **except** for references to child and parent categories. If set, the *channel* reference is kept. Also note the copy behavior of `UniqueObject`'s.

Example

```
import order as od

# toggle the default selection mode to Root-style selection string concatenation
od.Category.default_selection_mode = "root"

cat = od.Category("4j",
    label="4 jets",
    label_short="4j",
    selection="nJets == 4",
)

# note that no id needs to be passed to the Category constructor
# its id is set automatically based on the maximum id of currently existing category
# instances plus one (which is - of course - one in this example)
```

(continues on next page)

(continued from previous page)

```
cat.id
# -> 1

cat.label
# -> "4 jets"

# add a channel
ch = od.Channel("dilepton", 1,
    label="Dilepton",
    label_short="DL"
)
cat.channel = ch

# the category is also assigned to the channel now
cat in ch.categories
# -> True

# and we can create the full category label
cat.full_label
# -> "Dilepton, 4 jets"

# and the short version of it
cat.full_label_short
# -> "DL, 4j"

# add a sub category
cat2 = cat.add_category("4j_2b",
    label=cat.label + ", 2 b-tags",
)

# set the selection string (could also be set in add_category above)
cat2.selection = [cat.selection, "nBTags == 2"]

cat2.selection
# -> "(nJets == 4) && (nBTags == 2)"
```

Members

channel

type: Channel, None

The channel instance of this category, or *None* when not set.

full_label

type: string

read-only

The label of this category, prefix with the channel label if a channel is set.

full_label_short

type: string

read-only

The short label of this category, prefix with the short channel label if a channel is set.

full_label_root

type: string

read-only

The label of this category, prefix with the channel label if a channel is set, converted to ROOT-style latex.

full_label_short_root

type: string

read-only

The short label of this category, prefix with the short channel label if a channel is set, converted to ROOT-style latex.

categories

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of child categories.

parent_categories

type: UniqueObjectIndex

read-only

The *UniqueObjectIndex* of parent categories.

add_category(**args*, ***kwargs*)

Adds a child category to the *categories* index and returns it. Also adds *this* category to the *parent_categories* index of the added category. See `UniqueObjectIndex.add()` for more info.

add_parent_category(**args*, ***kwargs*)

Adds a parent category to the *parent_categories* index and returns it. Also adds *this* category to the *categories* index of the added category. When *context* is *None*, the *default_context* of the *categories* index is used. See `UniqueObjectIndex.add()` for more info.

clear_categories(*context=None*)

Removes all child categories from the *categories* index for *context*. Also removes *this* category instance from the *parent_categories* index of all removed categories. When *context* is *None*, the *default_context* of the *categories* index is used

clear_parent_categories(*context=None*)

Removes all parent categories from the *parent_categories* index for *context*. Also removes *this* category instance from the *categories* index of all removed category.

extend_categories(*objs*, *context=None*)

Adds multiple child categories to the *categories* index for *context* and returns the added objects in a list. Also adds *this* category to the *parent_categories* index of the added category. When *context* is *None*, the *default_context* of the *categories* index is used.

extend_parent_categories(*objs*, *context=None*)

Adds multiple parent categories to the *parent_categories* index for *context* and returns the added objects in a list. Also adds *this* category to the *categories* index of the added category. When *context* is *None*, the *default_context* of the *categories* index is used.

get_category(*obj*, *deep=True*, *default=no_default*, *context=None*)

Returns a child category given by *obj*, which might be a *name*, *id*, or an instance from the *categories* index for *context*. If *deep* is *True*, the lookup is recursive. When no category is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the *categories* index is used.

get_leaf_categories(*context=None*)

Returns all child categories from the `categories` index for *context* that have no child categories themselves in a recursive fashion. When *context* is *None*, the *default_context* of the `categories` index is used.

get_parent_category(*obj*, *deep=True*, *default=no_default*, *context=None*)

Returns a parent category given by *obj*, which might be a *name*, *id*, or an instance from the `parent_categories` index for *context*. If *deep* is *True*, the lookup is recursive. When no category is found, *default* is returned when set. Otherwise, an error is raised. When *context* is *None*, the *default_context* of the `parent_categories` index is used.

get_root_categories(*context=None*)

Returns all parent categories from the `parent_categories` index for *context* that have no parent categories themselves in a recursive fashion. When *context* is *None*, the *default_context* of the `parent_categories` index is used.

property has_categories

Returns *True* when this category has child categories, *False* otherwise.

has_category(*obj*, *deep=True*, *context=None*)

Checks if the `categories` index for *context* contains an *obj* which might be a *name*, *id*, or an instance. If *deep* is *True*, the lookup is recursive. When *context* is *None*, the *default_context* of the `categories` index is used.

property has_parent_categories

Returns *True* when this category has parent categories, *False* otherwise.

has_parent_category(*obj*, *deep=True*, *context=None*)

Checks if the `parent_categories` index for *context* contains an *obj*, which might be a *name*, *id*, or an instance. If *deep* is *True*, the lookup is recursive. When *context* is *None*, the *default_context* of the `parent_categories` index is used.

property is_leaf_category

Returns *True* when this category has no child categories, *False* otherwise.

property is_root_category

Returns *True* when this category has no parent categories, *False* otherwise.

remove_category(*obj*, *context=None*, *silent=False*)

Removes a child category given by *obj*, which might be a *name*, *id*, or an instance from the `categories` index for *context* and returns the removed object. Also removes *this* category from the `parent_categories` index of the removed category. When *context* is *None*, the *default_context* of the `categories` index is used. Unless *silent* is *True*, an error is raised if the object was not found. See `UniqueObjectIndex.remove()` for more info.

remove_parent_category(*obj*, *context=None*, *silent=False*)

Removes a parent category *obj* which might be a *name*, *id*, or an instance from the `parent_categories` index for *context*. Also removes *this* instance from the `categories` index of the removed parent category. Returns the removed object. When *context* is *None*, the *default_context* of the `parent_categories` index is used. Unless *silent* is *True*, an error is raised if the object was not found. See `UniqueObjectIndex.remove()` for more info.

walk_categories(*context=None*, *depth_first=False*, *include_self=False*)

Walks through the `categories` index for *context* and per iteration, yields a child category, its depth relative to *this* category, and its child categories in a list that can be modified to alter the walking. When *context* is *None*, the *default_context* of the `categories` index is used. When *context* is *all*, all indices are traversed. When *depth_first* is *True*, iterate depth-first instead of the default breadth-first. When *include_self* is *True*, also yield this category instance with a depth of 0.

walk_parent_categories(*context=None*, *depth_first=False*, *include_self=False*)

Walks through the `parent_categories` index for *context* and per iteration, yields a parent category, its depth relative to *this* category, and its parent categories in a list that can be modified to alter the walking. When *context* is *None*, the *default_context* of the `parent_categories` index is used. When *context* is *all*, all indices are traversed. When *depth_first* is *True*, iterate depth-first instead of the default breadth-first. When *include_self* is *True*, also yield this category instance with a depth of 0.

1.2.6 order.variable

Tools to work with variables.

Contents

- *order.variable*
  - *Class* Variable

Class *Variable*

class Variable(*name*, *id='+'*, *expression=None*, *binning=(1, 0.0, 1.0)*, *x_title=''*, *x_title_short=None*, *y_title='Entries'*, *y_title_short=None*, *x_labels=None*, *log_x=False*, *log_y=False*, *unit='1'*, *unit_format='{title} / {unit}'*, *null_value=None*, *selection=None*, *selection_mode=None*, *tags=None*, *aux=None*, *context=None*)

Bases: `UniqueObject`, `CopyMixin`, `AuxDataMixin`, `TagMixin`, `SelectionMixin`

Class that provides simplified access to variables for convenience methods for plotting with both ROOT and matplotlib.

Arguments

expression can be a string (for projection statements) or function that defines the variable expression. When empty, it defaults to *name*. *selection* is expected to be a string. Other options that are relevant for plotting are *binning*, *x_title*, *x_title_short*, *y_title*, *y_title_short*, *unit*, *unit_format* and *null_value*. See the attribute listing below for further information.

selection and *selection_mode* are passed to the `SelectionMixin`, *tags* to the `TagMixin`, *aux* to the `AuxDataMixin`, and *name*, *id* (defaulting to an automatically increasing id) and *context* to the `UniqueObject` constructor.

Copy behavior

All attributes are copied. Also note the copy behavior of `UniqueObject`'s.

Example

```
import order as od

v1 = od.Variable("myVar",
    expression="myBranchA * myBranchB",
    selection="myBranchC > 0",
    binning=(20, 0., 10.),
    x_title=r"$\mu p_{T}$",
    unit="GeV",
    null_value=-999.0,
```

(continues on next page)

(continued from previous page)

```
)

v1.x_title_root
# -> "#mu p_{T}"

v1.get_full_title()
# -> "myVar;$\mu p_{T}$" / GeV;Entries / 0.5 GeV'"

v2 = v1.copy(name="copiedVar", id="+",
    binning=[0.0, 0.5, 1.5, 3.0],
)

v2.get_full_title()
# -> "copiedVar;#mu p_{T} / GeV;Entries / GeV"

v2.even_binning
# -> False
```

Members

expression

type: string, callable, None

The expression of this variable. Defaults to name if *None*.

binning

type: tuple, list

Descibes the bin edges when given a list, or the number of bins, minimum value and maximum value when passed a 3-tuple.

even_binning

type: bool

read-only

Whether or not the binning is even.

x_title

type: string

The title of the x-axis in standard LaTeX format.

x_title_root

type: string

read-only

The title of the x-axis, converted to ROOT-style latex.

x_title_short

type: string

Short version for the title of the x-axis in standard LaTeX format. Defaults to *x_title* when not explicitely set.

x_title_short_root

type: string

read-only

The short version of the title of the x-axis, converted to ROOT-style latex.

y_title

type: string

The title of the y-axis in standard LaTeX format.

y_title_root

type: string

read-only

The title of the y-axis, converted to ROOT-style latex.

y_title_short

type: string

Short version for the title of the y-axis in standard LaTeX format. Defaults to *y_title* when not explicitely set.

y_title_short_root

type: string

read-only

The short version of the title of the y-axis, converted to ROOT-style latex.

x_labels

type: list, None

A list of custom bin labels or *None*. When not *None*, its length must be the same as the number of bins.

x_labels_root

type: list, None

read-only

A list of custom bin labels, converted to ROOT-style latex, or *None*.

unit

type: string, None

The unit to be shown on both, x- and y-axis. When *None*, no unit is shown.

unit_format

type: string

The format string for concatenating axis titles and units, e.g. `"{title} / {unit}"`. The format string must contain the fields *title* and *unit*.

null_value

type: int, float, None

A configurable NULL value for this variable, possibly denoting missing values. *None* is considered as "non-configured".

log_x

type: boolean

Whether or not the x-axis should be drawn logarithmically.

log_y

type: boolean

Whether or not the y-axis should be drawn logarithmically.

n_bins

type: int

read-only

The number of bins.

x_min

type: float

read-only

The minimum value of the x-axis.

x_max

type: float

read-only

The maximum value of the x-axis.

bin_width

type: float

read-only

The width of a bin.

bin_edges

type: list

read-only

A list of the *n_bins* + 1 bin edges.

get_full_x_title(*short=False*, *root=False*)

Returns the full title (i.e. with unit string) of the x-axis. When *short* is *True*, the short version is returned. When *root* is *True*, the title is converted to *proper* ROOT latex.

get_full_y_title(*bin_width=None*, *short=False*, *root=False*)

Returns the full title (i.e. with bin width and unit string) of the y-axis. When not *None*, the value *bin_width* instead of the one evaluated from *binning* when even. When *short* is *True*, the short version is returned. When *root* is *True*, the title is converted to ROOT-style latex.

get_full_title(*name=None*, *short=False*, *short_x=None*, *short_y=None*, *root=False*, *bin_width=None*)

Returns the full combined title that is compliant with ROOT's TH1 classes. *short_x* (*short_y*) is passed to `full_x_title()` (`full_y_title()`). Both values fallback to *short* when *None*. *bin_width* is forwarded to `full_y_title()`. When *root* is *False*, the axis titles are not converted to ROOT-style latex.

get_mpl_hist_data(*update=None*, *skip=None*)

Returns a dictionary containing information on *bins*, *range*, *label*, and *log*, that can be passed to matplotlib histograms. When *update* is set, the returned dict is updated with *update*. When *skip* is set, it can be a single key or a sequence of keys that will not be added to the returned dictionary.

1.2.7 order.shift

Classes and helpers to describe and work with systematic shifts.

Contents

- *order.shift*
  - *Class* Shift

Class *Shift*

class Shift(*name*, *id*, *type=None*, *label=None*, *label_short=None*, *tags=None*, *aux=None*, *context=None*)

Bases: *UniqueObject*, *CopyMixin*, *AuxDataMixin*, *TagMixin*, *LabelMixin*

Description of a systematic shift.

Arguments

The shift *name* should either be `"nominal"` or it should have the format `"<source>_<direction>"` where *direction* is either `"up"` or `"down"`. *type* describes the shift's effect, which is either only rate-changing (*RATE*) or also shape-changing (*SHAPE*). When *None*, *UNKNOWN* is used.

label and *label_short* are forwarded to the *LabelMixin*, *tags* to the *TagMixin*, *aux* to the *AuxDataMixin* *name*, *id* (defaulting to an auto id) and *context* to the *UniqueObject* constructor.

Copy behavior

All attributes are copied. Also note the copy behavior of *UniqueObject*'s.

Example

```
import order as od

s = od.Shift("nominal", 1)

s.name
# -> "nominal"

s.is_up
# -> False

s = Shift("pdf_up", 2)

s.source
# -> "pdf"

s.direction
# -> "up"

s.is_up
# -> True
```

Members

classattributeNOMINAL

type: string

Flag denoting a nominal shift (`"nominal"`). Same as scinum.Number.NOMINAL.

classattributeUP

type: string

Flag denoting an up variation (`"up"`). Same as scinum.Number.UP.

classattributeDOWN

type: string

Flag denoting a down variation (`"down"`). Same as scinum.Number.DOWN.

classattributeRATE

type: string

Flag denoting a rate-changing effect ("rate").

classattributeSHAPE

type: string

Flag denoting a shape-changing effect ("shape").

classattributeRATE_SHAPE

type: string

Flag denoting a both rate- and shape-changing effect ("rate_shape").

source

type: string

read-only

The source of this shift, e.g. *NOMINAL*, "pdf", etc.

direction

type: string

read-only

The direction of this shift, either *NOMINAL*, *UP* or *DOWN*.

type

type: string

The type of this shift, either *RATE*, *SHAPE* or *RATE_SHAPE*.

is_nominal

type: bool

read-only

Flag denoting if the shift is nominal.

is_up

type: bool

read-only

Flag denoting if the shift direction is *UP*.

is_down

type: bool

read-only

Flag denoting if the shift direction is *DOWN*.

is_rate

type: bool

read-only

Flag denoting if the shift type is rate-changing only.

is_shape

type: bool

read-only

Flag denoting if the shift type is shape-changing only.

is_rate_shape

type: bool

read-only

Flag denoting if the shift type is rate- and shape-changing.

classmethod split_name(*name*)

Splits a shift *name* into its source and direction. If *name* is *NOMINAL*, both source and direction will be *NOMINAL*. Example:

```
split_name("nominal")  # -> ("nominal", "nominal")
split_name("pdf_up")   # -> ("pdf", "up")
split_name("pdfup")    # -> ValueError: invalid shift name format: pdfup
```

classmethod join_name(*source*, *direction*)

Joins a shift *source* and a shift *direction* to return a shift name. If either *source* or *direction* is *None*, *None* is returned. If *source* is *NOMINAL*, *direction* must be *NOMINAL* as well. Otherwise, *direction* must be either *UP* or *DOWN*. Example:

```
join_name("nominal", "nominal")  # -> "nominal"
join_name("nominal", "up")       # -> ValueError: pointless nominal shift␣
↪direction
join_name("pdf", "up")           # -> "pdf_up"
join_name("pdf", "high")         # -> ValueError: invalid shift direction
```

1.2.8 order.unique

Classes that define unique objects and the index to store them.

Contents

- *order.unique*
  - *Class* UniqueObject
  - *Class* UniqueObjectIndex
  - *Class* UniqueObjectMeta
  - *Function* uniqueness_context
  - *Decorator* unique_tree

Class *UniqueObject*

class UniqueObject(*name*, *id*, *context=None*)

Bases: *UniqueObject*

An unique object defined by a *name* and an *id*. The purpose of this class is to provide a simple interface for objects that

1. are used programatically and should therefore have a unique, human-readable name, and
2. have a unique identifier that can be saved to files, such as (e.g.) ROOT trees.

Both, *name* and *id* should have unique values within a certain *uniqueness context*. This context defaults to either the current one as set by *uniqueness_context()*, the *default_context* of this class, or, whem empty, the lower-case class name. See `get_default_context()` for more info.

Arguments

name, *id* and *context* initialize the same-named attributes.

Copy behavior

When an inheriting class inherits also from *CopyMixin* certain copy rules apply for the *name*, *id* and *context* attributes as duplicate names or ids within the same context would directly cause an exception to be thrown (which is the desired behavior, see examples below). Two ways are in general recommended to copy a unique object:

```
import order as od

class MyClass(od.UniqueObject, od.CopyMixin):
    pass

orig = MyClass("foo", 1)

# 1. use the same context, explicitely change name and id
copy = orig.copy(name="bar", id=2)

# 2. use a different context, optionally set different name or id
with od.unqiueness_context("other"):
    copy = orig.copy()
    copy2 = orig.copy(name="baz")
```

Example

```
import order as od

foo = od.UniqueObject("foo", 1)

print(foo.name)
# -> "foo"
print(foo.id)
# -> 1

# name and id must be strictly string and integer types, respectively
od.UniqueObject(123, 1)
# -> TypeError: invalid name: 123
UniqueObject("foo", "mystring")
# -> TypeError: invalid id: mystring

# the name "foo" and the id 1 can no longer be used
# (at least not within the same uniqueness context, which is the default one when␣
↪not set)
bar = UniqueObject("foo", 2)
# -> DuplicateNameException: 'order.unique.UniqueObject' object with name 'foo' already
#                            exists in the uniqueness context 'uniqueobject'

bar = UniqueObject("bar", 1)
# -> DuplicateIdException: 'order.unique.UniqueObject' object with id '1' already␣
↪exists in
#                          the uniqueness context 'uniqueobject'
```

(continues on next page)

(continued from previous page)

```
# everything is fine when an other context is provided
bar = UniqueObject("bar", 1, context="myNewContext")
# works!

# unique objects can als be compared by name and id
foo == 1
# -> True

bar == "bar"
# -> True

foo == bar
# -> False

# automatically use the next highest possible id
obj = UniqueObject("baz", id=UniqueObject.AUTO_ID)

obj.id
# -> 2  # 1 is the highest used id in the default context, see above
```

Members

classattributedefault_context

type: arbitrary (hashable)

The default context of uniqueness when none is given in the instance constructor. Two instances are only allowed to have the same name *or* the same id if their classes have different contexts. This member is not inherited when creating a sub-class.

classattributecls_name_singular

type: str

The name of the unique object class in singular form, e.g. for producing automatic messages.

classattributecls_name_plural

type: str

The name of the unique object class in plural form, e.g. for producing automatic messages.

context

type: arbitrary (hashable)

The uniqueness context of this instance.

name

type: str

read-only

The unique name.

id

type: int

read-only

The unique id.

classmethod get_instance(*obj*, *default=no_default*, *context=None*)

Returns an object that was instantiated by this class before. *obj* might be a *name*, *id*, or an instance of *cls*.

If *default* is given, it is used as the default return value if no such object was found. Otherwise, an error is raised. *context* defaults to the `default_context` of this class.

classmethod auto_id(*name*, *context*)

Method to create an automatic id for instances that are created with `id="+"`. The default recipe is `max(ids) + 1`.

Class *UniqueObjectIndex*

class UniqueObjectIndex(*cls*, *objects=None*, *default_context=None*)

Bases: `CopyMixin`

Index of `UniqueObject` instances for faster lookup by either name or id. The instances are stored for different uniqueness context, so most methods have a *context* argument which usually defaults to the return value of `UniqueObject.get_default_context()`.

Arguments

cls must be a subclass of `UniqueObject`, which is used for type validation when a new object is added to the index. The *default_context* is used in case no context argument is set in most methods of this index object. It defaults to the return value of the *cls*' `UniqueObject.get_default_context()`.

Example

```
import order as od

idx = od.UniqueObjectIndex()
foo = idx.add("foo", 1)
bar = idx.add("bar", 2)

len(idx)
# -> 2

idx.names()
# -> ["foo", "bar"]

idx.ids()
# -> [1, 2]

idx.get(1) == foo
# -> True

# add objects for an other uniqueness context
# note: for the default context, the redundant use of the id 1 would have caused an␣
↪error!
baz = idx.add("baz", 1, context="other")

len(idx)
# -> 3

# idx.len() (with a context argument) returns the number of objects contained with␣
↪the
# default context
idx.len()
# -> 2
```

(continues on next page)

(continued from previous page)

```
idx.len(context="other")
# -> 1

# get ids of objects for all contexts (which might contain duplicates)
idx.ids(context=all)
# -> [1, 2, 1]
```

Members

classattributeALL

The flag that denotes that all contexts should be tranversed in methods that accept a *context* argument. It defaults to the built-in function `all`.

cls

type: class

read-only

Class of objects hold by this index.

default_context

type: string

The default context that is used when no *context* argument is provided in most methods.

n

type: DotAccessProxy

read-only

An object that provides simple attribute access to contained objects via name in the default context.

ALL()

Return True if bool(x) is True for all values x in the iterable.

If the iterable is empty, return True.

len(*context=None*)

Returns the length of the index stored for *context*. When *None*, the *default_context* is used. When *context* is *all*, the sum of lengths of all indices is returned, which is equivalent to `__len__()`.

contexts()

Returns a list of all contexts for whom indices are stored.

names(*context=None*)

Returns the names of the contained objects in the index stored for *context*. When *None*, the *default_context* is used. When *context* is *all*, a list of tuples (*name*, *context*) are returned with names from all indices. Note that the returned *context* refers to the one the object is stored in, rather than the uniqueness context of the object itself.

ids(*context=None*)

Returns the names of the contained objects in the index stored for *context*. When *None*, the *default_context* is used. When *context* is *all*, a list of tuples (*id*, *context*) are returned with ids from all indices. Note that the returned *context* refers to the one the object is stored in, rather than the uniqueness context of the object itself.

keys(*context=None*)

Returns pairs containing *name* and *id* of the currently contained objects in the index stored for *context*. When *None*, the *default_context* is used. When *context* is *all*, tuples (*name*, *id*, *context*) are returned with objects from all indices. Note that the returned *context* refers to the one the object is stored in, rather than the uniqueness context of the object itself.

values(*context=None*)

Returns all contained objects in the index stored for *context*. When *None*, the *default_context* is used. When *context* is *all*, tuples (*value*, *context*) are returned with objects from all indices. Note that the returned *context* refers to the one the object is stored in, rather than the uniqueness context of the object itself.

items(*context=None*)

Returns a list of (nested) tuples ((*name*, *id*), *value*) of all objects in the index stored for *context*. When *context* is *all*, tuples ((*name*, *id*), *value*, *context*) are returned with objects from all indices. Note that the returned *context* refers to the one the object is stored in, rather than the uniqueness context of the object itself.

add(**args*, ***kwargs*)

Adds a new object to the index for a certain context. When the first *arg* is not an instance of *cls*, all *args* and *kwargs* are passed to the *cls* constructor to create a new object. In this case, the *kwargs* may contain *index_context* to define the *context* if the index in which the newly created object should be stored. When not set, *default_context* is used. Otherwise, when the first *arg* is already an object and to be added, the context is either *index_context* or *contex*. The former has priority for consistency with the case described above. In both cases the added object is returned.

extend(*objs*, *context=None*)

Adds multiple new objects to the index for *context*. All elements of the sequence *objs* are forwarded to *add()* and returns the added objects in a list. When an object is a dictionary or a tuple, it is expanded for the invocation of *add()*. When *context* is *None*, the *default_context* is used.

get(*obj*, *default=no_default*, *context=None*)

Returns an object that is stored in the index for *context*. *obj* might be a *name*, *id*, or an instance of *cls*. If *default* is given, it is used as the default return value if no such object could be found. Otherwise, an error is raised. When *context* is *None*, the *default_context* is used.

get_first(*default=no_default*, *context=None*)

Returns the first object that is stored in the index for *context*. If *default* is given, it is used as the default return value if no object could be found. Otherwise, an error is raised. When *context* is *None*, the *default_context* is used.

get_last(*default=no_default*, *context=None*)

Returns the last object that is stored in the index for *context*. If *default* is given, it is used as the default return value if no object could be found. Otherwise, an error is raised. When *context* is *None*, the *default_context* is used.

has(*obj*, *context=None*)

Checks if an object is contained in the index for *context*. *obj* might be a *name*, *id*, or an instance of the wrapped *cls*. When *context* is *None*, the *default_context* is used.

index(*obj*, *context=None*)

Returns the position of an object in the index for *context*. When *context* is *None*, the *default_context* is used. *obj* might be a *name*, *id*, or an instance of *cls*. When the object is not found in the index, an error is raised.

remove(*obj*, *context=None*, *silent=False*)

Removes an object from the index for *context*. *obj* might be a *name*, *id*, or an instance of *cls*. Returns the

removed object. Unless *silent* is *True*, an error is raised if the object could not be found. When *context* is *None*, the *default_context* is used.

clear(*context=None*)

Clears the index for *context* by removing all elements. When *None*, the *default_context* is used. When *context* is *all*, the indices for all contexts are cleared.

Class *UniqueObjectMeta*

class UniqueObjectMeta(*class_name*, *bases*, *class_dict*)

Bases: `type`

Meta class definition that adds an instance cache to every class inheriting from *UniqueObject*.

Function *uniqueness_context*

uniqueness_context(*context*)

Adds the uniqueness *context* on top of the list of the *current contexts*, which is priotized in the *UniqueObject* constructor when no context is configured.

```
obj = UniqueObject("myObj", 1, context="myContext")

obj.context
# -> "myContext"

with uniqueness_context("otherContext"):
    obj2 = UniqueObject("otherObj", 2)

obj2.context
# -> "otherContext"
```

Decorator *unique_tree*

unique_tree(*cls=None*, *parents=1*, *deep_children=False*, *deep_parents=False*, *skip=None*)

Decorator that adds attributes and methods to the decorated class to provide tree features, i.e., *parent-child* relations. Example:

```
@unique_tree()
class MyNode(UniqueObject):
    cls_name_singular = "node"
    cls_name_plural = "nodes"
    default_context = "myclass"

# now, MyNode has the following attributes and methods:
# nodes,          parent_nodes,
# has_node(),     has_parent_node(),
# add_node(),     add_parent_node(),
# remove_node(),  remove_parent_node(),
# walk_nodes(),   walk_parent_nodes(),
# get_node(),     get_parent_node(),
# has_nodes,      has_parent_nodes,
```

(continues on next page)

(continued from previous page)

```
# is_leaf_node,   is_root_node

c1 = MyNode("nodeA", 1)
c2 = c1.add_node("nodeB", 2)

c1.has_node(2)
# -> True

c2.has_parent_node("nodeA")
# -> True

c2.remove_parent_node(c1)
c2.has_parent_node("nodeA")
# -> False
```

cls denotes the type of instances the tree should hold and defaults to the decorated class itself. When *parents* is *False*, the additional features are reduced to provide only child relations. When *parents* is an integer, it is interpreted as the maximim number of parents a child can have. Negative numbers mean that an unlimited amount of parents is allowed. Additional convenience methods are added when *parents* is *True* or exactly 1. When *deep_children* (*deep_parents*) is *True*, *get_** and *has_** child (parent) methods will have recursive features. When *skip* is a sequence, it can contain names of attributes to skip that would normally be created.

A class can be decorated multiple times. Internally, the objects are stored in a separated *UniqueObjectIndex* instance per added tree functionality.

Doc strings are automatically created.

1.2.9 order.mixins

Mixin classes providing common functionality.

Contents

- *order.mixins*
  - *Class* CopyMixin
  - *Class* CopySpec
  - *Class* AuxDataMixin
  - *Class* TagMixin
  - *Class* DataSourceMixin
  - *Class* SelectionMixin
  - *Class* LabelMixin
  - *Class* ColorMixin

Class *CopyMixin*

class CopyMixin

Bases: `object`

Mixin-class that adds copy features to inheriting classes.

Inheriting classes should define a *copy_specs* class member, which is supposed to be a list containing specifications per attribute to be copied. See *CopySpec* and `CopySpec.new()` for information about possible copy specifications.

Example

```
import order as od

some_object = object()

class MyClass(od.CopyMixin):

    copy_specs = [
        "name",
        {"attr": "obj", "ref": True},
    ]

    def __init__(self, name, obj):
        super(MyClass, self).__init__()
        self.name = name
        self.obj = obj

a = MyClass("foo", some_object)
a.name
# -> "foo"

b = a.copy()
b.name
# -> "foo"

b.obj is a.obj
# -> True

c = a.copy(name="bar")
c.name
# -> "bar"

# one can also use the python copy module
import copy

d = copy.copy(a)

d.name
# -> "foo"

d.obj is a.obj
# -> True
```

(continues on next page)

(continued from previous page)

```
# no distinction is made between copy and deepcopy
e = copy.deepcopy(a)

e.obj is a.obj
# -> True
```

Members

classattributecopy_specs

type: list

List of copy specifications per attribute.

_copy_attribute(*obj*, *spec*)

Copies an object *obj*, taking into account the *CopySpec* speficications *spec*, and returns the copy. Internally, `copy.copy` and `copy.deepcopy` are used to copy objects.

_copy_attribute_manual(*inst*, *obj*, *spec*)

Hook that is called in *copy()* to invoke the manual copying of an object *obj* **after** the copied instance *inst* was created. *spec* is the associated *CopySpec* object. Instead of returning the copied object, the method should directly alter *inst*.

_copy_ref(*kwargs*, *cls*, *specs*)

Hook that is called in *copy()* before an instance is actually copied. When this method returns *True*, no new instance is created but rather a reference will be returned. The default is *False*. This is useful in special situations that require flexible copy decisions. *kwargs* is a dictionary that contains all *args* and *kwargs* passed to *copy()* (*args* are included by mapping them to the target argument names via inspection), *cls* is the target class to instantiate, and *specs* is the full list of *CopySpec* instances.

copy(**args*, ***kwargs*, *_cls=None*, *_specs=None*, *_replace_specs=False*, *_skip=None*)

Creates a copy of this instance and returns it. Internally, an instance if *_cls* is created, defaulting to the class of *this* instance, with all *args* and *kwargs* forwarded to its constructor. Attributes that are not present in *args* or *kwargs* are copied over from *this* instance based in the attribute specifications given in *copy_specs*. They can be extended by *_specs* or even replaced when *_replace_specs* is *True*. *_skip* can be a sequence of destination attribute names that should be skipped.

__copy__()

Shorthand to *copy()* without arguments.

__deepcopy__(*memo*)

Shorthand to *copy()* without arguments.

Class *CopySpec*

class CopySpec(*dst*, *src=None*, *ref=False*, *shallow=False*, *use_setter=False*, *manual=False*)

Bases: `object`

Class holding attribute copy specifications. Instances of this class are used in *CopyMixin* to describe the copy behavior individually per attribute.

Arguments

dst is the destination attribute name, *src* is the source attribute. When *ref* is *True*, the attribute is not copied but a reference is passed to the newly created instance. By default, `copy.deepcopy` is used to copy attributes (if *ref* is *False*). When *shallow* is *True*, `copy.copy` is used instead. The standard way to pass copied attributes to new instances is via constructor arguments. However, you can set *use_setter* to *True* to use a plain setter to add the copied

attribute to the instance. *manual* denotes whether the custom *CopyMixin._copy_attribute_manual()* method should be invoked to copy an attribute. This method must be implemented by inheriting functions.

Members

dst

type: string

The destination attribute.

src

type: string

The source attribute.

ref

type: bool

Whether or not the attribute should be passed as a reference instead of copying.

shallow

type: bool

Whether or not the attribute should be copied shallow or deep.

use_setter

type: bool

Whether or not a setter should be invoked to set the copied attribute. When *False*, it is passed as a constructor argument (the default).

manual

type: bool

Whether or not the attribute should be copied manually by the custom *CopyMixin._copy_attribute_manual()* method.

Class *AuxDataMixin*

class AuxDataMixin(*aux=None*)

Bases: `object`

Mixin-class that provides storage of auxiliary data via a simple interface.

Arguments

aux can be a dictionary or a list of 2-tuples to initialize the auxiliary data container. For convenient access via attributes, each instance of this mixin-class has a proxy object *x* which can be used to obtain auxiliary information. See example below.

Example

```
import order as od

class MyClass(od.AuxDataMixin):
    pass

c = MyClass(aux={"foo": "bar"})

# "foo" in c.aux
# same as c.has_aux("foo")
```

(continues on next page)

(continued from previous page)

```
# -> True

c.set_aux("test", "some_value")
c.get_aux("test")
# -> "some_value"

c.get_aux("notthere")
# -> KeyError

c.get_aux("notthere", default=123)
# -> 123

c.x.test
# -> "some_value"

c.x("test")  # same as c.get_aux
# -> "some_value"

c.x.notthere
# -> AttributeError
```

Members

aux

type: OrderedDict

The dictionary of auxiliary data.

x

type: DotAccessProxy

read-only

An object that provides simple attribute access to auxiliary data.

set_aux(*key*, *value*)

Stores auxiliary *value* for a specific *key*. Returns *value*.

has_aux(*key*)

Returns *True* when an auxiliary data entry for a specific *key* exists, *False* otherwise.

get_aux(*key*[, *default*])

Returns the auxiliary data for a specific *key*. If a *default* is given, it is returned in case *key* is not found.

remove_aux(*key*, *silent=False*)

Removes the auxiliary data for a specific *key*. Unless *silent* is *True*, an exception is raised if the *key* to remove is not found.

clear_aux()

Clears the auxiliary data container.

Class *TagMixin*

class TagMixin(*tags=None*)

Bases: `object`

Mixin-class that allows inheriting objects to be attribute one or more *tags*. See the example below for more infos.

Arguments

tags initializes the internal set of stored tags.

Example

```
import order as od

class MyClass(od.TagMixin):
    pass

c = MyClass(tags={"foo", "bar"})

c.has_tag("foo")
# -> True

c.has_tag("baz")
# -> False

c.has_tag(("foo", "baz"), mode=any)  # the default mode
# -> True

c.has_tag(("foo", "baz"), mode=all)
# -> False

c.has_tag(("foo", "ba*"), mode=all)
# -> True

c.has_tag(("foo", "ba(r|z)"), mode=all, func="re")
# -> True
```

Members

tags

type: set

The set of tags of this object. See *has_tag()* for information about how to evaluate them with patterns or regular expressions.

add_tag(*tag*)

Adds a new *tag* to the set of tags.

remove_tag(*tag*)

Removes a previously added *tag* from the set if tags.

has_tag(*tag*, *mode=any*, ***kwargs*)

Returns *True* when this object is tagged with *tag*, *False* otherwise. When *tag* is a sequence of tags, the behavior is defined by *mode*. For *any*, the object is considered *tagged* when at least one of the provided tags matches. When *all*, all provided tags have to match. Each *tag* can be a *fnmatch* or *re* pattern. All *kwargs* are passed to `util.multi_match()`.

Class *DataSourceMixin*

class DataSourceMixin(*is_data=False*)

Bases: object

Mixin-class that provides convenience attributes for distinguishing between MC and real data.

Arguments

is_data initializes the same-named attribute.

Example

```
import order as od

class MyClass(od.DataSourceMixin):
    pass

c = MyClass(is_data=False)

c.is_data
# -> False

c.data_source
# -> "mc"

c.is_data = True
c.data_source
# -> "data"
```

Members

classattributeDATA_SOURCE_DATA

type: string

The data source string for data ("data").

classattributeDATA_SOURCE_MC

type: string

The data source string for mc ("mc").

is_data

type: boolean

True if this object contains information on real data.

is_mc

type: boolean

True if this object contains information on MC data.

data_source

type: string

Either *DATA_SOURCE_DATA* or *DATA_SOURCE_MC*, depending on the source of contained data.

Class *SelectionMixin*

class SelectionMixin(*selection=None*, *selection_mode=None*)

Bases: `object`

Mixin-class that adds attibutes and methods to describe a selection rule using ROOT- or numexpr-style expression syntax, or a bare callable.

Arguments

selection and *selection_mode* initialize the same-named attributes.

Example

```
import order as od

class MyClass(od.SelectionMixin):
    pass

# ROOT-style expressions
c = MyClass(selection="branchA > 0", selection_mode=MyClass.MODE_ROOT)

c.selection
# -> "branchA > 0"

c.add_selection("myBranchB < 100")
c.selection
# -> "(myBranchA > 0) && (myBranchB < 100)"

c.add_selection("myWeight", op="*")
c.selection
# -> "((myBranchA > 0) && (myBranchB < 100)) * (myWeight)"

# numexpr-style expressions
c = MyClass(selection="branchA > 0", selection_mode=MyClass.MODE_NUMEXPR)

c.add_selection("myBranchB < 100")
c.selection
# -> "(myBranchA > 0) & (myBranchB < 100)"

def my_selection(*args, **kwargs):
    ...

c.selection = my_selection
c.selection
# -> <function my_selection()>
c.selection_mode
# -> None
c.add_selection("myBranchB < 100")
# -> TypeError
```

Members

classattributeMODE_ROOT

type: string

Flag denoting the ROOT-style selection mode (`"root"`).

classattributeMODE_NUMEXP

type: string

Flag denoting the numexpr-style selection mode ("numexpr").

classattributedefault_selection_mode

type: string

The default *selection_mode* when none is given in the instance constructor. It is initially set to *MODE_NUMEXPR* if *order.util.ROOT_DEFAULT* is *false*, or to *MODE_ROOT* otherwise.

selection

type: string, callable

The selection string or a callable. When a string, *selection_mode* decides how the string is treated.

selection_mode

type: string, None

The selection mode. Should either be *MODE_ROOT* or *MODE_NUMEXPR*. Only considered when *selection* is a string.

add_selection(*selection*, ***kwargs*)

Adds a *selection* string to the current one if it is also a string. The new string will be logically connected via *AND* by default, which can be configured through *kwargs*. All *kwargs* are forwarded to `util.join_root_selection()` or `util.join_numexpr_selection()`.

Class *LabelMixin*

class LabelMixin(*label=None*, *label_short=None*)

Bases: `object`

Mixin-class that provides a label, a short version of that label, and some convenience attributes.

Arguments

label and *label_short* initialize the same-named attributes.

Example

```
import order as od

l = od.LabelMixin(label="Muon")

l.label
# -> "Muon"

# when not set, the short label returns the normal label
l.label_short
# -> "Muon"

l.label_short = r"$\mu$"
l.label_short
# -> "$\mu$"

# conversion to ROOT-style latex
l.label_short_root
# -> "#mu"
```

Members

label

type: string

The label. When this object has a *name* (configurable via *_label_fallback_attr*) attribute, the label defaults to that value when not set.

label_root

type: string

read-only

The label, converted to ROOT-style latex.

label_short

type: string

A short label, which defaults to the normal label when not set.

label_short_root

type: string

read-only

Short version of the label, converted to ROOT-style latex.

Class *ColorMixin*

class ColorMixin(*color=None*)

Bases: `object`

Mixin-class that provides a color in terms of RGB values as well as some convenience methods.

Arguments

color can be a tuple of 3 or 4 numbers that are interpreted as red, green and blue color values, and optinally an alpha value. Floats are stored internally. When integers are passed, they are divided by 255.

Example

```
import order as od

c = od.ColorMixin(color=(255, 0.5, 100))

c.color
# -> (1.0, 0.5, 0.392..)

c.color_int
# -> (255, 128, 100)

c.color_alpha
# -> 1.0
```

Members

color_r

type: float

Red component.

color_g

type: float

Green component.

color_b

type: float

Blue component.

color_r_int

type: int

Red component, converted to an integer in the [0, 255] range.

color_g_int

type: int

Green component, converted to an integer in the [0, 255] range.

color_b_int

type: int

Blue component, converted to an integer in the [0, 255] range.

color_alpha

type: float

The alpha value, defaults to 1.

color

type: tuple (float)

The RGB color values in a 3-tuple.

color_int

type: tuple (int)

The RGB int color values in a 3-tuple.

1.2.10 order.util

Helpful utilities.

ROOT_DEFAULT = False

Boolean value that denotes if ROOT-style selection strings, latex labels, etc, are used by default. The value defaults to *False* and can be altered by setting

```
os.environ["ORDER_ROOT_DEFAULT"] = 1|"1"|"yes"|"true"
```

before importing order. This rather peculiar mechanism is favored over e.g. using a setter since also default values of various methods across the order package depend on this flag.

class typed(*fparse=None*, *setter=True*, *deleter=True*, *name=None*)

Shorthand for the most common property definition. Can be used as a decorator to wrap around a single function. Example:

```
class MyClass(object):

    def __init__(self):
        self._foo = None
```

(continues on next page)

(continued from previous page)

```
    @typed
    def foo(self, foo):
        if not isinstance(foo, str):
            raise TypeError("not a string: '%s'" % foo)
        return foo

myInstance = MyClass()
myInstance.foo = 123   -> TypeError
myInstance.foo = "bar" -> ok
print(myInstance.foo)  -> prints "bar"
```

In the exampe above, set/get calls target the instance member `_foo`, i.e. "_<function_name>". The member name can be configured by setting *name*. If *setter* (*deleter*) is *True* (the default), a setter (deleter) method is booked as well. Prior to updating the member when the setter is called, *fparse* is invoked which may implement sanity checks.

make_list(*obj*, *cast=True*)

Converts an object *obj* to a list and returns it. Objects of types *tuple* and *set* are converted if *cast* is *True*. Otherwise, and for all other types, *obj* is put in a new list.

multi_match(*name*, *patterns*, *mode=<built-in function any>*, *func='fnmatch'*, **args*, ***kwargs*)

Compares *name* to multiple *patterns* and returns *True* in case of at least one match (*mode = any*, the default), or in case all patterns matched (*mode = all*). Otherwise, *False* is returned. *func* determines the matching function and accepts `"fnmatch"`, `"fnmatchcase"`, and `"re"`. All *args* and *kwargs* are passed to the actual matching function.

flatten(*struct*, *depth=- 1*)

Flattens and returns a complex structured object *struct* up to a certain *depth*. When *depth* is negative, *struct* is flattened entirely.

to_root_latex(*s*)

Converts latex expressions in a string *s* to ROOT-compatible latex.

join_root_selection(**selection*, *op='&&'*, *bracket=False*)

Returns a concatenation of root *selection* strings, which is done by default via logical *AND*. (*op*). When *bracket* is *True*, the final selection string is placed into brackets.

join_numexpr_selection(**selection*, *op='&'*, *bracket=False*)

Returns a concatenation of numexpr *selection* strings, which is done by default via logical *AND*. (*op*). When *bracket* is *True*, the final selection string is placed into brackets.

class_id(*cls*)

Returns the full id of a *class*, i.e., the id of the module it is defined in, extended by the name of the class. Example:

```
# module a.b

class MyClass(object):
    ...

class_id(MyClass)
# "a.b.MyClass"
```

args_to_kwargs(*func*, *args*)

Converts arguments *args* passed to a function *func* to a dictionary that can be used as keyword arguments. Internally, `inspect.getargspec` is used to get the names of arguments in the function signature. Example:

```
def func(a, b, c=1):
    ...

def wrapper(*args, **kwargs):
    kwargs.update(args_to_kwargs(func, args))
    # kwargs now contains the initial args and kwargs for easy parsing
    ...
    return func(**kwargs)
```

class DotAccessProxy(*getter*, *setter=None*)

Proxy object that provides simple attribute access to values that are retrieved by a *getter* and optionally set through a *setter*. Example:

```
my_dict = {"foo": 123}

proxy = DotAccessProxy(my_dict.__getattr__)
proxy.foo
# -> 123
proxy.bar
# -> AttributeError

proxy = DotAccessProxy(my_dict.get)
proxy.foo
# -> 123
proxy.bar
# -> None

proxy = DotAccessProxy(my_dict.get, my_dict.__setitem__)
proxy.foo
# -> 123
proxy.bar
# -> None
proxy.bar = 99
proxy.bar
# -> 99
```

See the intro.ipynb notebook for an introduction to the most important classes and an example setup of a small analysis. You can also run the notebook interactively on binder:

CHAPTER

TWO

INSTALLATION AND DEPENDENCIES

Install *order* via pip:

```
pip install order
```

The only dependencies are scinum and six, which are installed with the above command.

CHAPTER

THREE

CONTRIBUTING AND TESTING

If you like to contribute, I'm happy to receive pull requests. Just make sure to add new test cases and run them via:

```
python -m unittest tests
```

In general, tests should be run for Python 2.7, 3.6, 3.7 and 3.8. To run tests in a docker container, do

```
# run the tests
./tests/docker.sh python:3.8

# or interactively by adding a flag "1" to the command
./tests/docker.sh python:3.8 1
> pip install -r requirements.txt
> python -m unittest tests
```

In addition, PEP 8 compatibility should be checked with flake8:

```
flake8 order tests setup.py
```

CHAPTER

FOUR

DEVELOPMENT

- Source hosted at GitHub
- Report issues, questions, feature requests on GitHub Issues

PYTHON MODULE INDEX

O

order.analysis, 12
order.category, 29
order.config, 14
order.dataset, 21
order.mixins, 50
order.process, 25
order.shift, 40
order.unique, 43
order.util, 60
order.variable, 37

INDEX

Symbols

__copy__() (*CopyMixin method*), 52
__deepcopy__() (*CopyMixin method*), 52
_copy_attribute() (*CopyMixin method*), 52
_copy_attribute_manual() (*CopyMixin method*), 52
_copy_ref() (*CopyMixin method*), 52

A

add() (*UniqueObjectIndex method*), 48
add_category() (*Category method*), 35
add_category() (*Channel method*), 30
add_category() (*Config method*), 17
add_channel() (*Channel method*), 30
add_channel() (*Config method*), 17
add_config() (*Analysis method*), 13
add_dataset() (*Campaign method*), 15
add_dataset() (*Config method*), 17
add_parent_category() (*Category method*), 35
add_parent_channel() (*Channel method*), 30
add_parent_process() (*Process method*), 27
add_process() (*Config method*), 17
add_process() (*Dataset method*), 24
add_process() (*Process method*), 27
add_selection() (*SelectionMixin method*), 58
add_shift() (*Config method*), 17
add_tag() (*TagMixin method*), 55
add_variable() (*Config method*), 17
ALL (*UniqueObjectIndex class attribute*), 47
ALL() (*UniqueObjectIndex method*), 47
Analysis (*class in order.analysis*), 12
analysis (*Config attribute*), 16
args_to_kwargs() (*in module order.util*), 61
auto_id() (*UniqueObject class method*), 46
aux (*AuxDataMixin attribute*), 54
AuxDataMixin (*class in order.mixins*), 53

B

bin_edges (*Variable attribute*), 40
bin_width (*Variable attribute*), 40
binning (*Variable attribute*), 38
bx (*Campaign attribute*), 14

C

Campaign (*class in order.config*), 14
campaign (*Config attribute*), 16
campaign (*Dataset attribute*), 23
categories (*Category attribute*), 35
categories (*Channel attribute*), 30
categories (*Config attribute*), 17
Category (*class in order.category*), 33
channel (*Category attribute*), 34
Channel (*class in order.category*), 29
channels (*Channel attribute*), 30
channels (*Config attribute*), 17
class_id() (*in module order.util*), 61
clear() (*UniqueObjectIndex method*), 49
clear_aux() (*AuxDataMixin method*), 54
clear_categories() (*Category method*), 35
clear_categories() (*Channel method*), 30
clear_categories() (*Config method*), 17
clear_channels() (*Channel method*), 30
clear_channels() (*Config method*), 17
clear_configs() (*Analysis method*), 13
clear_datasets() (*Campaign method*), 15
clear_datasets() (*Config method*), 18
clear_parent_categories() (*Category method*), 35
clear_parent_channels() (*Channel method*), 31
clear_parent_processes() (*Process method*), 27
clear_processes() (*Config method*), 18
clear_processes() (*Dataset method*), 24
clear_processes() (*Process method*), 27
clear_shifts() (*Config method*), 18
clear_variables() (*Config method*), 18
cls (*UniqueObjectIndex attribute*), 47
cls_name_plural (*UniqueObject class attribute*), 45
cls_name_singular (*UniqueObject class attribute*), 45
color (*ColorMixin attribute*), 60
color_alpha (*ColorMixin attribute*), 60
color_b (*ColorMixin attribute*), 60
color_b_int (*ColorMixin attribute*), 60
color_g (*ColorMixin attribute*), 59
color_g_int (*ColorMixin attribute*), 60
color_int (*ColorMixin attribute*), 60
color_r (*ColorMixin attribute*), 59

color_r_int (*ColorMixin attribute*), 60
ColorMixin (*class in order.mixins*), 59
Config (*class in order.config*), 15
configs (*Analysis attribute*), 12
context (*UniqueObject attribute*), 45
contexts() (*UniqueObjectIndex method*), 47
copy() (*CopyMixin method*), 52
copy_specs (*CopyMixin class attribute*), 52
CopyMixin (*class in order.mixins*), 51
CopySpec (*class in order.mixins*), 52

D

data_source (*DataSourceMixin attribute*), 56
DATA_SOURCE_DATA (*DataSourceMixin class attribute*), 56
DATA_SOURCE_MC (*DataSourceMixin class attribute*), 56
Dataset (*class in order.dataset*), 22
DatasetInfo (*class in order.dataset*), 25
datasets (*Campaign attribute*), 15
datasets (*Config attribute*), 17
DataSourceMixin (*class in order.mixins*), 56
default_context (*UniqueObject class attribute*), 45
default_context (*UniqueObjectIndex attribute*), 47
default_selection_mode (*SelectionMixin class attribute*), 58
direction (*Shift attribute*), 42
DotAccessProxy (*class in order.util*), 62
DOWN (*Shift class attribute*), 41
dst (*CopySpec attribute*), 53

E

ecm (*Campaign attribute*), 14
even_binning (*Variable attribute*), 38
expression (*Variable attribute*), 38
extend() (*UniqueObjectIndex method*), 48
extend_categories() (*Category method*), 35
extend_categories() (*Channel method*), 31
extend_categories() (*Config method*), 18
extend_channels() (*Channel method*), 31
extend_channels() (*Config method*), 18
extend_configs() (*Analysis method*), 13
extend_datasets() (*Campaign method*), 15
extend_datasets() (*Config method*), 18
extend_parent_categories() (*Category method*), 35
extend_parent_channels() (*Channel method*), 31
extend_parent_processes() (*Process method*), 27
extend_processes() (*Config method*), 18
extend_processes() (*Dataset method*), 24
extend_processes() (*Process method*), 27
extend_shifts() (*Config method*), 18
extend_variables() (*Config method*), 18

F

flatten() (*in module order.util*), 61
full_label (*Category attribute*), 34
full_label_root (*Category attribute*), 34
full_label_short (*Category attribute*), 34
full_label_short_root (*Category attribute*), 35

G

get() (*UniqueObjectIndex method*), 48
get_aux() (*AuxDataMixin method*), 54
get_categories() (*Analysis method*), 12
get_category() (*Category method*), 35
get_category() (*Channel method*), 31
get_category() (*Config method*), 18
get_channel() (*Channel method*), 31
get_channel() (*Config method*), 18
get_channels() (*Analysis method*), 12
get_config() (*Analysis method*), 13
get_dataset() (*Campaign method*), 15
get_dataset() (*Config method*), 18
get_datasets() (*Analysis method*), 12
get_first() (*UniqueObjectIndex method*), 48
get_full_title() (*Variable method*), 40
get_full_x_title() (*Variable method*), 40
get_full_y_title() (*Variable method*), 40
get_info() (*Dataset method*), 24
get_instance() (*UniqueObject class method*), 45
get_last() (*UniqueObjectIndex method*), 48
get_leaf_categories() (*Category method*), 35
get_leaf_categories() (*Channel method*), 31
get_leaf_categories() (*Config method*), 19
get_leaf_channels() (*Channel method*), 31
get_leaf_channels() (*Config method*), 19
get_leaf_processes() (*Config method*), 19
get_leaf_processes() (*Dataset method*), 24
get_leaf_processes() (*Process method*), 27
get_mpl_hist_data() (*Variable method*), 40
get_parent_category() (*Category method*), 36
get_parent_channel() (*Channel method*), 31
get_parent_process() (*Process method*), 27
get_process() (*Config method*), 19
get_process() (*Dataset method*), 24
get_process() (*Process method*), 27
get_processes() (*Analysis method*), 13
get_root_categories() (*Category method*), 36
get_root_channels() (*Channel method*), 31
get_root_processes() (*Process method*), 28
get_shift() (*Config method*), 19
get_shifts() (*Analysis method*), 13
get_variable() (*Config method*), 19
get_variables() (*Analysis method*), 13
get_xsec() (*Process method*), 27

H

has() (*UniqueObjectIndex method*), 48
has_aux() (*AuxDataMixin method*), 54

has_categories (*Category property*), 36
has_categories (*Channel property*), 31
has_categories (*Config property*), 19
has_category() (*Category method*), 36
has_category() (*Channel method*), 32
has_category() (*Config method*), 19
has_channel() (*Channel method*), 32
has_channel() (*Config method*), 19
has_channels (*Channel property*), 32
has_channels (*Config property*), 19
has_config() (*Analysis method*), 13
has_configs (*Analysis property*), 13
has_dataset() (*Campaign method*), 15
has_dataset() (*Config method*), 19
has_datasets (*Campaign property*), 15
has_datasets (*Config property*), 19
has_parent_categories (*Category property*), 36
has_parent_category() (*Category method*), 36
has_parent_channel() (*Channel method*), 32
has_parent_channels (*Channel property*), 32
has_parent_process() (*Process method*), 28
has_parent_processes (*Process property*), 28
has_process() (*Config method*), 19
has_process() (*Dataset method*), 24
has_process() (*Process method*), 28
has_processes (*Config property*), 20
has_processes (*Dataset property*), 24
has_processes (*Process property*), 28
has_shift() (*Config method*), 20
has_shifts (*Config property*), 20
has_tag() (*TagMixin method*), 55
has_variable() (*Config method*), 20
has_variables (*Config property*), 20

I

id (*UniqueObject attribute*), 45
ids() (*UniqueObjectIndex method*), 47
index() (*UniqueObjectIndex method*), 48
info (*Dataset attribute*), 23
is_data (*DataSourceMixin attribute*), 56
is_down (*Shift attribute*), 42
is_leaf_category (*Category property*), 36
is_leaf_category (*Channel property*), 32
is_leaf_category (*Config property*), 20
is_leaf_channel (*Channel property*), 32
is_leaf_channel (*Config property*), 20
is_leaf_config (*Analysis property*), 13
is_leaf_dataset (*Campaign property*), 15
is_leaf_dataset (*Config property*), 20
is_leaf_process (*Config property*), 20
is_leaf_process (*Dataset property*), 24
is_leaf_process (*Process property*), 28
is_leaf_shift (*Config property*), 20
is_leaf_variable (*Config property*), 20
is_mc (*DataSourceMixin attribute*), 56
is_nominal (*Shift attribute*), 42
is_rate (*Shift attribute*), 42
is_rate_shape (*Shift attribute*), 42
is_root_category (*Category property*), 36
is_root_channel (*Channel property*), 32
is_root_process (*Process property*), 28
is_shape (*Shift attribute*), 42
is_up (*Shift attribute*), 42
items() (*UniqueObjectIndex method*), 48

J

join_name() (*Shift class method*), 43
join_numexpr_selection() (*in module order.util*), 61
join_root_selection() (*in module order.util*), 61

K

keys (*Dataset attribute*), 23
keys (*DatasetInfo attribute*), 25
keys() (*UniqueObjectIndex method*), 47

L

label (*LabelMixin attribute*), 59
label_root (*LabelMixin attribute*), 59
label_short (*LabelMixin attribute*), 59
label_short_root (*LabelMixin attribute*), 59
LabelMixin (*class in order.mixins*), 58
len() (*UniqueObjectIndex method*), 47
log_x (*Variable attribute*), 39
log_y (*Variable attribute*), 39

M

make_list() (*in module order.util*), 61
manual (*CopySpec attribute*), 53
MODE_NUMEXP (*SelectionMixin class attribute*), 57
MODE_ROOT (*SelectionMixin class attribute*), 57
module
    order.analysis, 12
    order.category, 29
    order.config, 14
    order.dataset, 21
    order.mixins, 50
    order.process, 25
    order.shift, 40
    order.unique, 43
    order.util, 60
    order.variable, 37
multi_match() (*in module order.util*), 61

N

n (*UniqueObjectIndex attribute*), 47
n_bins (*Variable attribute*), 39
n_events (*Dataset attribute*), 23

n_events (*DatasetInfo attribute*), 25
n_files (*Dataset attribute*), 23
n_files (*DatasetInfo attribute*), 25
name (*UniqueObject attribute*), 45
names() (*UniqueObjectIndex method*), 47
NOMINAL (*Shift class attribute*), 41
null_value (*Variable attribute*), 39

O

order.analysis
    module, 12
order.category
    module, 29
order.config
    module, 14
order.dataset
    module, 21
order.mixins
    module, 50
order.process
    module, 25
order.shift
    module, 40
order.unique
    module, 43
order.util
    module, 60
order.variable
    module, 37

P

parent_categories (*Category attribute*), 35
parent_channels (*Channel attribute*), 30
parent_processes (*Process attribute*), 26
pretty_print() (*Process method*), 28
pretty_print_all() (*Process class method*), 27
Process (*class in order.process*), 25
processes (*Config attribute*), 17
processes (*Dataset attribute*), 23
processes (*Process attribute*), 26

R

RATE (*Shift class attribute*), 41
RATE_SHAPE (*Shift class attribute*), 42
ref (*CopySpec attribute*), 53
remove() (*UniqueObjectIndex method*), 48
remove_aux() (*AuxDataMixin method*), 54
remove_category() (*Category method*), 36
remove_category() (*Channel method*), 33
remove_category() (*Config method*), 20
remove_channel() (*Channel method*), 32
remove_channel() (*Config method*), 20
remove_config() (*Analysis method*), 13
remove_dataset() (*Campaign method*), 15
remove_dataset() (*Config method*), 20
remove_parent_category() (*Category method*), 36
remove_parent_channel() (*Channel method*), 32
remove_parent_process() (*Process method*), 28
remove_process() (*Config method*), 20
remove_process() (*Dataset method*), 24
remove_process() (*Process method*), 28
remove_shift() (*Config method*), 21
remove_tag() (*TagMixin method*), 55
remove_variable() (*Config method*), 21
ROOT_DEFAULT (*in module order.util*), 60

S

selection (*SelectionMixin attribute*), 58
selection_mode (*SelectionMixin attribute*), 58
SelectionMixin (*class in order.mixins*), 57
set_aux() (*AuxDataMixin method*), 54
set_info() (*Dataset method*), 24
set_xsec() (*Process method*), 27
shallow (*CopySpec attribute*), 53
SHAPE (*Shift class attribute*), 42
Shift (*class in order.shift*), 41
shifts (*Config attribute*), 16
source (*Shift attribute*), 42
split_name() (*Shift class method*), 43
src (*CopySpec attribute*), 53

T

TagMixin (*class in order.mixins*), 55
tags (*TagMixin attribute*), 55
to_root_latex() (*in module order.util*), 61
type (*Shift attribute*), 42
typed (*class in order.util*), 60

U

unique_tree() (*in module order.unique*), 49
uniqueness_context() (*in module order.unique*), 49
UniqueObject (*class in order.unique*), 43
UniqueObjectIndex (*class in order.unique*), 46
UniqueObjectMeta (*class in order.unique*), 49
unit (*Variable attribute*), 39
unit_format (*Variable attribute*), 39
UP (*Shift class attribute*), 41
use_setter (*CopySpec attribute*), 53

V

values() (*UniqueObjectIndex method*), 48
Variable (*class in order.variable*), 37
variables (*Config attribute*), 17

W

walk_categories() (*Category method*), 36
walk_categories() (*Channel method*), 32

walk_categories() (*Config method*), 21
walk_channels() (*Channel method*), 32
walk_channels() (*Config method*), 21
walk_parent_categories() (*Category method*), 37
walk_parent_channels() (*Channel method*), 33
walk_parent_processes() (*Process method*), 28
walk_processes() (*Config method*), 21
walk_processes() (*Dataset method*), 24
walk_processes() (*Process method*), 29

X

x (*AuxDataMixin attribute*), 54
x_labels (*Variable attribute*), 39
x_labels_root (*Variable attribute*), 39
x_max (*Variable attribute*), 40
x_min (*Variable attribute*), 40
x_title (*Variable attribute*), 38
x_title_root (*Variable attribute*), 38
x_title_short (*Variable attribute*), 38
x_title_short_root (*Variable attribute*), 38
xsecs (*Process attribute*), 26

Y

y_title (*Variable attribute*), 38
y_title_root (*Variable attribute*), 39
y_title_short (*Variable attribute*), 39
y_title_short_root (*Variable attribute*), 39