hepdata_lib Documentation

Release 0.15.0

Andreas Albert, Clemens Lange

Apr 18, 2024

CONTENTS:

CHAPTER

ONE

HEPDATA_LIB

Library for getting your data into HEPData

- Documentation: https://hepdata-lib.readthedocs.io

This code works with Python 3.6, 3.7, 3.8, 3.9, 3.10, 3.11 or 3.12.

1.1 Installation

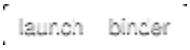
It is highly recommended you install `hepdata_lib` into a virtual environment.

```
pip install hepdata_lib
```

If you are not sure about your Python environment, please also see below how to use `hepdata_lib` in a Docker or Apptainer container.

1.2 Getting started

For using `hepdata_lib`, you don't even need to install it, but can use the binder or SWAN (CERN-only) services using one of the buttons below:



You can also use the Docker image:

```
docker run --rm -it -p 8888:8888 -v ${PWD}:/home/hepdata ghcr.io/hepdata/hepdata_
↪lib:latest
```

And then point your browser to http://localhost:8888 and use the token that is printed out. The output will end up in your current working directory (${PWD}).

If you prefer a shell, instead run:

```
docker run --rm -it -p 8888:8888 -v ${PWD}:/home/hepdata ghcr.io/hepdata/hepdata_
↪lib:latest bash
```

If on CERN LXPLUS or anywhere else where there is Apptainer available but not Docker, you can still use the docker image:

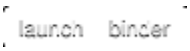
```
export APPTAINER_CACHEDIR="/tmp/$(whoami)/apptainer"
apptainer shell -B /afs -B /eos docker://ghcr.io/hepdata/hepdata_lib:latest bash
```

Unpacking the image can take a few minutes the first time you use it. Please be patient. Both EOS and AFS should be available and the output will be in your current working directory.

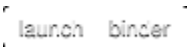
1.3 Further examples

There are a few more examples available that can directly be run using the binder links below or using SWAN (CERN-only, please use LCG release LCG_94 or later) and selecting the corresponding notebook manually:

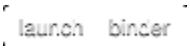
- Reading in text files



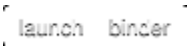
- Reading in a CMS combine ntuple



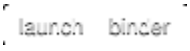
- Reading in ROOT histograms



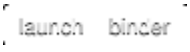
- Reading a correlation matrix



- Reading TGraph and TGraphError from '.C' files



- Preparing scikit-hep histograms



1.4 External dependencies

- ROOT
- ImageMagick

Make sure that you have `ROOT` in your `$PYTHONPATH` and that the `convert` command is available by adding its location to your `$PATH` if needed.

A ROOT installation is not strictly required if your input data is not in a ROOT format, for example, if your input data is provided as text files or `scikit-hep/hist` histograms. Most of the `hepdata_lib` functionality can be used without a ROOT installation, other than the `RootFileReader` and `CFileReader` classes, and other functions of the `hepdata_lib.root_utils` module.

CHAPTER

TWO

SETUP

2.1 Setup for users

2.1.1 With Python 3 locally available

The library is available for installation as a PyPI package and can be installed from the terminal with `pip`:

```
pip install hepdata_lib (--user)
```

The `--user` flag lets you install the package in a user dependent location, and thus avoids writing to the location of your system's main python installation. This is useful because in many cases you, the user, do not have writing permissions to wherever the system keeps its python.

If you are running on your own computer or laptop, it's up to you to decide where you want to install the package. However, we recommended to install it inside a virtual environment (see *Setting up a virtual environment*)

This setup naturally restricts you to use the latest stable version of the package in pypi. If you would like to use the code as it is the github repository, please follow the instructions in *Setup for developers*.

2.1.2 Using Apptainer

On LXPLUS and many other local computing sites with CVMFS and Apptainer (previously called Singularity) available, you can use the library without having to install anything:

```
apptainer run /cvmfs/unpacked.cern.ch/ghcr.io/hepdata/hepdata_lib:latest /bin/bash
```

This opens a new shell with `hepdata_lib`, ROOT, and Python 3 available. Your home directory and most other user directories on the machine on which you execute Apptainer will also be accessible from within this shell.

2.1.3 Using SWAN

SWAN requires a CERN account. `hepdata_lib` should already be installed in most recent LCG Releases used by SWAN. The latest LCG Release might not contain the latest `hepdata_lib` version. The LCG Nightly, possibly containing a more recent `hepdata_lib` version, can be used by selecting the "Bleeding Edge" software stack in the SWAN configuration. Alternatively, you can upgrade `hepdata_lib` by adding a local installation path to the `$PYTHONPATH` in a startup script specified as the "Environment script" in the SWAN configuration (see Install packages in CERNBox). Then execute `!pip install hepdata_lib --user --upgrade` in your Jupyter notebook to upgrade `hepdata_lib` to the latest version.

2.2 Setup for developers

The general comments about installing a python package (see *Setup for users*) apply here, too. Use a virtual environment (see *Setting up a virtual environment*)!

If you would like to develop the code, you need to install the package from the up-to-date GitHub repository rather than the stable release in PyPI. To do this, you can use the pip `-e` syntax. The GitHub repository can be cloned using either an HTTPS URL (`git clone https://github.com/HEPData/hepdata_lib.git`) or an SSH URL:

```
cd $SOMEPATH
git clone git@github.com:HEPData/hepdata_lib.git
cd $SOMEPATH/hepdata_lib

python3 -m venv myhepdata
source myhepdata/bin/activate  # activate virtual environment!
pip install -e $SOMEPATH/hepdata_lib
```

Any changes you now make in your local copy of the repository (editing files, checking out different branches...) will be picked up by the python installation in your virtual environment.

2.3 Setting up a virtual environment

The goal of a virtual environment is to have a clean python copy to work with for each of possibly many projects you work on. This makes it easy to keep track of which python packages you installed for what purpose and gives you a way of installing different versions of the package.

For documentation on how to set up and work with virtual environments, please check out Installing packages using pip and virtual environments. The venv module has been provided in the standard Python library since Python 3.3, avoiding the need to install separate virtualenv and virtualenvwrapper packages as was the case with earlier Python versions.

You can create a virtual environment to work in:

```
python3 -m venv hepdata_pypi
source hepdata_pypi/bin/activate
pip install hepdata_lib
```

You can then have a second virtual environment for installing the development branch:

```
python3 -m venv hepdata_git
source hepdata_git/bin/activate
pip install -e $SOMEPATH/hepdata_lib
```

You can always activate the virtual environment in another shell by sourcing the relevant `activate` script, which also allows you to easily switch between the two instances:

```
source hepdata_pypi/bin/activate
python myscript.py  # Execute script using pypi package
```

2.4 Setup on lxplus with CMSSW

The `hepdata_lib` library is shipped with CMSSW via cmsdist. However, please make sure that you are using a recent CMSSW release, since otherwise you might be using an outdated version of the library. After running `cmsenv`, you can check the installed version as follows:

```
python3 -m pip list | grep hepdata-lib
```

(mind the use of `hepdata-lib` above, when importing, the package is still called `hepdata_lib`). If the version is significantly older than the one on PyPI, please use the Apptainer container as described at *Setup for users* above.

CHAPTER

THREE

USAGE

The library aims to offer tools for two main operations:

- *Reading data* from the usual formats (ROOT, text files, etc.) into python lists.
- *Writing data* from python lists to the HEPData YAML-based format

All of this happens in a user-friendly python interface. *Reading data* is helpful if you need help getting your data as a python list. If you already have your data accessible in python, great! Skip right ahead to *Writing data*.

In the following sections, there are

3.1 HEPData and its data format

The HEPData data model revolves around **Tables** and **Variables**. At its core, a Variable is a one-dimensional array of numbers with some additional (meta-)data, such as uncertainties, units, etc. assigned to it. A Table is simply a set of multiple Variables. This definition will immediately make sense to you when you think of a general table, which has multiple columns representing different variables.

3.2 Reading data

3.2.1 Reading from plain text

If you save your data in a text file, a simple-to-use tool is the `numpy.loadtxt` function, which loads column-wise data from plain-text files and returns it as a `numpy.array`.

```
import numpy as np
my_array = np.loadtxt("some_file.txt")
```

A detailed example is available here. For documentation on the *loadtxt* function, please refer the numpy documentation.

3.2.2 Reading from ROOT files

In many cases, data in the experiments is available as one of various ROOT data types, such as `TGraphs`, `TH1`, `TH2`, etc, which are saved in `*.root` files.

To facilitate reading these objects, the `RootFileReader` class is provided. The reader is instantiated by passing a path to the ROOT file to read from:

```
from hepdata_lib import RootFileReader
reader = RootFileReader("/path/to/myfile.root")
```

After initialization, individual methods are provided for access to different types of objects stored in the file.

- Reading TGraph, TGraphErrors, TGraphAsymmErrors: `RootFileReader.read_graph`
- Reading TH1: `RootFileReader.read_hist_1d`
- Reading TH2: `RootFileReader.read_hist_2d`

While the details of each function are adapted to their respective use cases, they follow a common input/output logic. The methods are called by providing the path to the object inside the ROOT file. They return a dictionary containing lists of all relevant numbers that can be extracted from the object, such as x values, y values, uncertainties, etc.

As an example, if a TGraph is saved as with name `mygraph` in the directory `topdir/subdir` inside the ROOT file, it can be retrieved as:

```
data = reader.read_graph("topdir/subdir/mygraph")
```

Since a graph is simply a set of (x,y) pairs for each point, the `data` dictionary will have two key/value pairs:

- key “x” -> list of x values.
- key “y” -> list of y values.

More complex information will be returned for `TGraphErrors`, etc, which can also be read in this manner. For detailed descriptions of the extraction logic and returned data, please refer to the documentation of the individual methods.

An example notebook shows how to read histograms from a ROOT file.

3.3 Writing data

Following the HEPData data model, the hepdata_lib implements four main classes for writing data:

- **Submission**
- **Table**
- **Variable**
- **Uncertainty**

3.3.1 The Submission object

The Submission object is the central object where all threads come together. It represents the whole HEPData entry and thus carries the top-level meta data that is equally valid for all the tables and variables you may want to enter. The object is also used to create the physical submission files you will upload to the HEPData web interface.

When using hepdata_lib to make an entry, you **always need to create a Submission object**. The most bare-bone submission consists of only a Submission object with no data in it:

```
from hepdata_lib import Submission
sub = Submission()
outdir="./output"
sub.create_files(outdir)
```

The `create_files` function writes all the YAML output files you need and packs them up in a `tar.gz` file ready to be uploaded.

Please note: creating the output files also creates a `submission` folder containing the individual files going into the tarball. This folder exists merely for convenience, in order to make it easy to inspect each individual file. It is not recommended to attempt to manually manage or edit the files in the folder, and there is no guarantee that `hepdata_lib` will handle any of the changes you make in a graceful manner. As far as we are aware, there is no use case where manual editing of the files is necessary. If you have such a use case, please report it in a Github issue.

Adding resource links or files

Additional resources, hosted either externally or locally, can be linked with the `add_additional_resource` function of the Submission object.

```
sub.add_additional_resource("Web page with auxiliary material", "https://atlas.web.cern.
↪ch/Atlas/GROUPS/PHYSICS/PAPERS/STDM-2012-02/")
sub.add_additional_resource("Some file", "root_file.root", copy_file=True)
sub.add_additional_resource("Some file", "root_file.root", copy_file=True, resource_
↪license={"name": "CC BY 4.0", "url": "https://creativecommons.org/licenses/by/4.0/",
↪"description": "This license enables reusers to distribute, remix, adapt, and build␣
↪upon the material in any medium or format, so long as attribution is given to the␣
↪creator."})
sub.add_additional_resource("Archive of full likelihoods in the HistFactory JSON format",
↪ "Likelihoods.tar.gz", copy_file=True, file_type="HistFactory")
```

The first argument is a `description` and the second is the `location` of the external link or local resource file. The optional argument `copy_file=True` (default value of `False`) will copy a local file into the output directory. The optional argument `resource_license` can be used to define a data license for an additional resource. The `resource_license` is in the form of a dictionary with mandatory string `name` and `url` values, and an optional `description`. The optional argument `file_type="HistFactory"` (default value of `None`) can be used to identify statistical models provided in the HistFactory JSON format rather than relying on certain trigger words in the `description` (see pyhf section of submission documentation).

Please note: The default license applied to all data uploaded to HEPData is CC0. You do not need to specify a license for a resource file unless it differs from CC0.

The `add_link` function can alternatively be used to add a link to an external resource:

```
sub.add_link("Web page with auxiliary material", "https://atlas.web.cern.ch/Atlas/GROUPS/
↪PHYSICS/PAPERS/STDM-2012-02/")
```

Again, the first argument is a `description` and the second is the `location` of the external link.

Adding links to related records

To add a link to a related record object, you can use the `add_related_recid` function of the Submission object.

Please note: values must be entered as integers.

```
sub.add_related_recid(1)
sub.add_related_recid(2)
sub.add_related_recid(3)
```

In the last example, we are adding a link to the submission with the record ID value of 3.

Please note: This field should not be used for self-referencing, the IDs inserted should be for OTHER related records.

The documentation for this feature can be found here: Linking records.

3.3.2 Tables and Variables

The real data is stored in Variables and Tables. Variables come in two flavors: *independent* and *dependent*. Whether a variable is independent or dependent may change with context, but the general idea is that the independent variable is what you put in, the dependent variable is what comes out. Example: if you calculate a cross-section limit as a function of the mass of a hypothetical new particles, the mass would be independent, the limit dependent. The number of either type of variables is not limited, so if you have a scenario where you give N results as a function of M model parameters, you can have N dependent and M independent variables. All the variables are then bundled up and added into a Table object.

Let's see what this looks like in code:

```
from hepdata_lib import Variable

mass = Variable("Graviton mass",
                is_independent=True,
                is_binned=False,
                units="GeV")
mass.values = [ 1, 2, 3 ]

limit = Variable("Cross-section limit",
                is_independent=False,
                is_binned=False,
                units="fb")
limit.values = [ 10, 5, 2 ]

table = Table("Graviton limits")
table.add_variable(mass)
table.add_variable(limit)
```

That's it! We have successfully created the Table and Variables and stored our results in them. The only task left is to tell the Submission object about our new Table:

```
sub.add_table(table)
```

After we have done this, the table will be included in the output files the `Submission.create_files` function writes (see *The Submission object*).

Binned Variables

The above example uses unbinned Variables, which means that every point is simply a single number reflecting a localized value. In many cases, it is useful to use binned Variables, e.g. to represent the x axis of a histogram. In this case, everything works the same way as in the unbinned case, except that we have to specify `is_binned=True` in the Variable constructor, and change how we format the list of values:

```
mass_binned = Variable("Same mass as before, but this time it's binned",
                       is_binned=True,
                       is_independent=True)
mass_binned.values = [ (0.5, 1.5), (1.5, 2.5), (2.5, 3.5) ]
```

The list of values has an entry for each bin of the Variable. The entry is a tuple, where the first entry represents the lower edge of the bin, while the second entry represents the upper edge of the bin. You can simply plug this definition into the code snippet of the unbinned case above to go from an unbinned mass to a binned value. Note that binning a Variable only really makes sense for independent variables.

Two-dimensional plots

In some cases, you may want to define information based on multiple parameters, e.g. in the case of a two-dimensional histogram (TH2 in ROOT). This can be easily accomplished by defining two independent Variables in the same Table:

```
table = Table()

x = Variable("Variable on the x axis",
             is_independent=True,
             is_binned=True)
# x.values = [ ... ]

y = Variable("Variable on the y axis",
             is_independent=True,
             is_binned=True)
# y.values = [ ... ]

v1 = Variable("A variable depending on x and y",
              is_independent=False,
              is_binned=False)
# v1.values = [ ... ]

v2 = Variable("Another variable depending on x and y",
              is_independent=False,
              is_binned=False)
# v2.values = [ ... ]

table.add_variable(x)
table.add_variable(y)
table.add_variable(v1)
table.add_variable(v2)
```

Note that you can add as many dependent Variables as you would like, and that you can also make the independent variables unbinned.

One common use case with more than one independent Variable is that of correlation matrices. A detailed example implementation of this case is available here.

Adding a plot thumb nail to a table

HEPData supports the addition of thumb nail images to each table. This makes it easier for the consumer of your entry to find what they are looking for, since they can simply look for the table that has the thumb nail of the plot they are interested in. If you have the full-size plot available on your drive, you can add it to your entry very easily:

```
table.add_image("path/to/image.pdf")
```

The library code then takes care of all the necessary steps, like converting the image to the right format and size, and copying it into your submission folder. The conversion relies on the ImageMagick library, and will only work if the `convert` command is available on your machine.

Adding resource links or files

In the same way as for the Submission object, additional resources, hosted either externally or locally, can be linked with the `add_additional_resource` function of the Table object.

```
table.add_additional_resource("Web page with auxiliary material", "https://atlas.web.
↪cern.ch/Atlas/GROUPS/PHYSICS/PAPERS/STDM-2012-02/")
table.add_additional_resource("Some file", "root_file.root", copy_file=True)
```

For a description of the arguments, see *Adding resource links or files* for the Submission object. A possible use case is to attach the data for the table in its original format before it was transformed into the HEPData YAML format.

Adding keywords to a table

To make HEPData entries more searchable, keywords should be used to define what information is shown in a table. HEPData keeps track of keywords separately from the rest of the information in an entry, and provides dedicated functionalities to search for and filter by a given set of keywords. If a user is e.g. interested in finding all tables relevant to graviton production, they can do so quite easily if the tables are labelled properly. This procedure becomes much harder, or even impossible, if no keywords are used. It is therefore considered good practice to add a number of sensible keywords to your tables.

The keywords are stored as a simple dictionary for each table:

```
table.keywords["observables"] = ["ACC", "EFF"]
table.keywords["reactions"] = ["P P --> GRAVITON --> W+ W-", "P P --> WPRIME --> W+/W- Z0
↪"]
```

In this example, we specify that the observables shown in a table are acceptance (“ACC”) and efficiency (“EFF”). We also specify the reaction we are talking about, in this case graviton or W’ production with decays to SM gauge bosons. This code snippet is taken from one of our examples.

Lists of recognized keywords are available from the hepdata documentation for Observables, Phrases, and Particles.

Adding links to related tables

To add a link to a related table object, you can use the `add_related_doi` function of the Table class.

Please note: your DOIs must match the format: `10.17182/hepdata.[RecordID].v[Version]/t[Table]`.

```
table.add_related_doi("10.17182/hepdata.72886.v2/t3")
table.add_related_doi("10.17182/hepdata.12882.v1/t2")
```

In the second example, we are adding a link to the table with a DOI value of 10.17182/hepdata.12882.v1/t2.

Please note: This field should not be used for self-referencing, the DOIs inserted should be for OTHER related tables.

The documentation for this feature can be found here: Linking tables.

Adding a data license

You can add data license information to a table using the `add_data_license` function of the Table class. This function takes mandatory `name` and `url` string arguments, as well as an optional `description`.

Please note: The default license applied to all data uploaded to HEPData is CC0. You do not need to specify a license for a data table unless it differs from CC0.

```
table.add_data_license("CC BY 4.0", "https://creativecommons.org/licenses/by/4.0/")
table.add_data_license("CC BY 4.0", "https://creativecommons.org/licenses/by/4.0/",
↪"This license enables reusers to distribute, remix, adapt, and build upon the material␣
↪in any medium or format, so long as attribution is given to the creator.")
```

3.3.3 Uncertainties

In many cases, you will want to give uncertainties on the central values provided in the Variable objects. Uncertainties can be *symmetric* or *asymmetric* (up and down variations of the central value either have the same or different magnitudes). For symmetric uncertainties, the values of the uncertainties are simply stored as a one-dimensional list. For asymmetric uncertainties, the up- and downward variations are stored as a list of two-component tuples:

```
from hepdata_lib import Uncertainty
unc1 = Uncertainty("A symmetric uncertainty", is_symmetric=True)
unc1.values = [ 0.1, 0.3, 0.5]

unc2 = Uncertainty("An asymmetric uncertainty", is_symmetric=False)
unc2.values = [ (-0.08, +0.15), (-0.13, +0.20), (-0.18,+0.27) ]
```

After creating the Uncertainty objects, the only additional step is to attach them to the Variable:

```
variable.add_uncertainty(unc1)
variable.add_uncertainty(unc2)
```

See Uncertainties for more guidance. In particular, note that `hepdata_lib` will omit the `errors` key from the YAML output if all uncertainties are zero for a particular bin, printing a warning message "Note that bins with zero content should preferably be omitted completely from the HEPData table". A legitimate use case is where there are multiple dependent variables and a (different) subset of the bins has missing content for some dependent variables. In this case the uncertainties should be set to zero for the missing bins with a non-numeric central value like `'-'`. The warning message can be suppressed by passing an optional argument `zero_uncertainties_warning=False` when defining an instance of the `Variable` class.

CHAPTER

FOUR

DEVELOPER INFORMATION

4.1 The testing system

While further developing the code base, we want to ensure that any changes we make do not accidentally break existing functionality. For this purpose, we use continuous integration via GitHub Actions. Practically, this means that we define a set of test code that is automatically run in a predefined environment every time we push to a branch or create a pull request. If the test code runs successfully, we know that everything works as expected.

To run the tests, move into the `hepdata_lib` directory while in your virtual environment and run

```
pip install -e ".[test]"
pytest tests
```

It is a good idea to run the tests manually to ensure that your changes do not cause any issues.

If you don't have a working ROOT installation, a subset of the tests can still be run without ROOT:

```
pytest tests -m "not needs_root"
```

4.1.1 Definition of test cases

Test cases are defined in the `test` subfolder of the repository. Independent sets of test code are defined in individual files in that directory, with each files being named like `test_*.py`.

Inside each file, a test class is defined that inherits from the `TestCase` class from the `unittest` package. The actual code to be run is then implemented as functions of the test object, and all functions named `test_*` will be run automatically. Example:

```
from unittest import TestCase

class TestForFeatureX(TestCase):
    """Test case for feature X."""

    def test_aspect_y(self):
        """Test that aspect Y works."""
        # Do something with feature X and sub-feature Y."""

    def test_aspect_z(self):
        """Test that aspect Z works."""
        # Do something with feature X and sub-feature Z."""
```

If all functions run without raising exceptions, the test is considered to be passed. Therefore, you should ensure that the test functions only run through if everything is really as expected.

When to test

Tests should be added any time functionality is added. If functionality is modified, so should the tests.

What to test

Generally, the tests should ensure that the code behaves as expected, whatever that may mean. They should be sufficiently rigorous to make sure that nothing can break silently, and that outputs are correct.

Broad inspiration for aspects to check:

- Are all inputs handled gracefully? What happens if an inconsistent input type is provided?
- Is the output correct for a variety of plausible inputs?
- If the tested code can raise exceptions: Is the exceptions really raised if and only if the expected criteria are met?

How to test

The `TestCase` base class provides functionality to help implement simple checks to verify the behavior. A test can immediately be made to fail by calling `self.fail()`. For convenience, additional functions like `assertTrue`, `assertFalse`, etc. are provided, which work like normal python assertions. If the assertion fails, the test is considered to be failed. Additionally, the `assertRaises` method can be used as a context to ensure that exceptions are raised as expected. Here's a simple example:

```
class TestForArithmetic(TestCase):
    """Test case for python arithmetic"""

    def test_addition(self):
        """Test that the addition operator works."""

        # This will fail the test if 1+1 is not equal to 2
        self.assertTrue(1+1 == 2)

        # Equivalent implementation using the explicit fail method:
        if 1+1 != 2:
            self.fail()

    def test_addition_exception(self):
        """Test that the addition operator raises the right exceptions."""

        # Check that it raises an expected TypeError for bad input
        with self.assertRaises(TypeError):
            val = None + 5

        # Check that no TypeError is raised for good input
        try:
            val = 1 + 5
        except TypeError:
            self.fail("The addition operator raised an unexpected TypeError.")
```

Note that this is an overly simple example case: In a real testing case, you should try to cover all possible and impossible input types for a thorough test coverage.

Check out the unittest package documentation for details of the available testing functionality.

4.2 Building the documentation

After installing the `hepdata_lib` package, move into the `hepdata_lib/docs` directory and install the additional necessary packages into your virtual environment:

```
pip install -r requirements.txt
```

Then you can build the documentation locally with Sphinx using `make html` and view the output by opening a web browser at `_build/html/index.html`. In addition, please also test whether building the LateX (`make latexpdf`) and epub (`make epub`) versions works.

4.3 Analysing the code

```
pylint hepdata_lib/*.py
pylint tests/*.py
```

These commands are run by GitHub Actions (for Python 3.8 or later), so you should first check locally that no issues are flagged.

4.4 Making a release

After making a new release available on PyPI, a JIRA issue (example) should be opened to request that `hepdata_lib` is upgraded in future LCG Releases used by SWAN.

CHAPTER

FIVE

CODE DOCUMENTATION

5.1 hepdata_lib package

5.1.1 Module contents

hepdata_lib main.

class hepdata_lib.**AdditionalResourceMixin**

Bases: `object`

Functionality related to additional materials.

add_additional_resource(*description*, *location*, *copy_file=False*, *file_type=None*, *resource_license=None*)

Add any kind of additional resource. If copy_file is set to False, the location and description will be added as-is. This is useful e.g. for the case of providing a URL to a web-based resource.

If copy_file is set to True, we will try to copy the file from the location you have given into the output directory. This only works if the location is a local file. If the location you gave does not exist or points to a file larger than 100 MB, a RuntimeError will be raised. While the file checks are performed immediately (i.e. the file must exist when this function is called), the actual copying only happens once create_files function of the submission object is called.

Parameters

- **description** (*string*) – Description of what the resource is.
- **location** (*string*) – Can be either a URL pointing to a web-based resource or a local file path.
- **copy_file** (*bool*) – If set to true, will attempt to copy a local file to the tar ball.
- **file_type** (*string*) – Type of the resource file. Currently, only "HistFactory" has any effect.
- **resource_license** (*dict*) – License information comprising name, url and optional description.

copy_files(*outdir*)

Copy the files in the files_to_copy list to the output directory.

Parameters

outdir (*string*) – Output directory path to copy to.

class hepdata_lib.**Submission**

Bases: *AdditionalResourceMixin*

Top-level object of a HEPData submission.

Holds all the lower-level objects and controls writing.

add_link(*description*, *location*)

Append link to additional_resources list.

Parameters

- **description** (*string.*) – Description of what the link refers to.
- **location** (*string*) – URL to link to.

add_record_id(*r_id*, *r_type*)

Append record_id to record_ids list.

add_related_recid(*r_id*)

Appends a record ID to the related_records list. :param r_id: The record's ID :type r_id: integer

add_table(*table*)

Append table to tables list.

Parameters
table (*Table.*) – The table to be added.

create_files(*outdir='.'*, *validate=True*, *remove_old=False*)

Create the output files.

Implicitly triggers file creation for all tables that have been added to the submission, all variables associated to the tables and all uncertainties associated to the variables.

If *validate* is True, the hepdata-validator package will be used to validate the output tar ball.

If *remove_old* is True, the output directory will be deleted before recreation.

files_to_copy_nested()

List files-to-copy for this Submission and nested daughters

static get_license()

Return the default license.

read_abstract(*filepath*)

Read in the abstracts file.

Parameters
filepath (*string.*) – Path to text file containing abstract.

class hepdata_lib.**Table**(*name*)

Bases: *AdditionalResourceMixin*

A table is a collection of variables.

It also holds meta-data such as a general description, the location within the paper, etc.

add_data_license(*name*, *url*, *description=None*)

Verify and store the given license data.

Parameters

- **name** (*string*) – The license name

- **url** (*string*) – The license URL
- **description** (*string*) – The (optional) license description

add_image(*file_path*, *outdir=None*)

Add an image file to the table.

This function only stores the path to the image. Any additional processing will be done later (see write_images function).

Parameters

- **file_path** (*string*) – Path to the image file.
- **outdir** – Deprecated.

add_related_doi(*doi*)

Appends a DOI string to the related_tables list.

Parameters
doi (*string*) – The table DOI.

add_variable(*variable*)

Add a variable to the table

Parameters
variable (*Variable.*) – Variable to add.

property name

Name getter.

write_images(*outdir*)

Write image files and thumbnails into the output directory.

Parameters
outdir (*string*) – Path to output directory. Will be created if it doesn't exist.

write_output(*outdir*)

Write the table files into the output directory.

Parameters
outdir (*string*) – Path to output directory. Will be created if it doesn't exist.

write_yaml(*outdir='.'*)

Write the table (and all its variables) to a YAML file.

This function is intended to be called internally by the Submission object. Except for debugging purposes, no user should have to call this function.

class hepdata_lib.**Uncertainty**(*label*, *is_symmetric=True*)

Bases: object

Store information about an uncertainty on a variable

Uncertainties can be symmetric or asymmetric. The main information is stored as one (two) lists in the symmetric (asymmetric) case. The list entries are the uncertainty for each of the list entries in the corresponding Variable.

scale_values(*factor*)

Multiply each value by constant factor.

Parameters
factor (*float*) – Value to multiply by.

set_values_from_intervals(*intervals*, *nominal*)

Set values relative to set of nominal values. Useful if you do not have the actual uncertainty available, but the upper and lower boundaries of an interval.

Parameters

- **intervals** (*List of tuples of two floats*) – Lower and upper interval boundaries
- **nominal** (*List of floats*) – Interval centers

property values

Value getter.

Returns

list – values, either as a direct list of values if uncertainty is symmetric, or list of tuples if it is asymmetric.

class hepdata_lib.**Variable**(*name*, *is_independent=True*, *is_binned=True*, *units=''*, *values=None*, *zero_uncertainties_warning=True*)

Bases: object

A Variable is a wrapper for a list of values + some meta data.

add_qualifier(*name*, *value*, *units=''*)

Add a qualifier.

add_uncertainty(*uncertainty*)

Add an uncertainty.

If the Variable object already has values assigned to it, it is required that the value list of the Uncertainty object has the same length as the list of Variable values.

If the list of values of the Variable is empty, no requirement is applied on the length of the list of Uncertainty values.

make_dict()

Return all data in this Variable as a dictionary.

The dictionary structure follows the hepdata conventions, so that dumping this dictionary to YAML will give a file that hepdata can read.

Uncertainties associated to this Variable are also written into the dictionary.

This function is intended to be called internally by the Submission object. Except for debugging purposes, no user should have to call this function.

scale_values(*factor*)

Multiply each value by constant factor. Also applies to uncertainties.

property values

Value getter.

hepdata_lib.**dict_constructor**(*loader*, *node*)

construct dict.

hepdata_lib.**dict_representer**(*dumper*, *data*)

represent dict.

.C file reader

class hepdata_lib.c_file_reader.**CFileReader**(*cfile*)

Bases: object

Reads ROOT Objects from .C files

property cfile

The .C file this reader reads from.

check_for_comments(*line*)

Check line for comment

create_tgraph(*x_value*, *y_value*)

Function to create pyroot TGraph object

create_tgraph_dict(*graph_list*, *list_of_tgraphs*)

Function to create pyroot TGraph dict

create_tgrapherrors(*x_value*, *y_value*, *dx_value*, *dy_value*)

Function to create pyroot TGraphErrors object

create_tgrapherrors_dict(*graph_list*)

Function to create pyroot TGraphErrors dict

find_graphs()

Find all TGraphs in .C file

get_graphs()

Parse the .C file trying to find TGraph objects

read_graph(*graphname*)

Function to read values of a graph

hepdata_lib helper functions.

hepdata_lib.helpers.**any_uncertainties_nonzero**(*uncertainties*, *size*)

Return a mask of bins where any of the uncertainties is nonzero.

hepdata_lib.helpers.**check_file_existence**(*path_to_file*)

Check that the given file path exists. If not, raise RuntimeError.

Parameters

path_to_file (*string*) – File path to check.

hepdata_lib.helpers.**check_file_size**(*path_to_file*, *upper_limit=None*, *lower_limit=None*)

Check that the file size is between the upper and lower limits. If not, raise RuntimeError.

Parameters

- **path_to_file** (*string*) – File path to check.
- **upper_limit** (*float*) – Upper size limit in MB.
- **lower_limit** (*float*) – Lower size limit in MB.

hepdata_lib.helpers.**convert_pdf_to_png**(*source*, *target*)

Wrapper for the ImageMagick convert utility.

Parameters

- **source** (*str*) – Source file in PDF format.
- **target** (*str*) – Output file in PNG format.

hepdata_lib.helpers.**convert_png_to_thumbnail**(*source*, *target*)

Wrapper for the ImageMagick convert utility in thumbnail mode.

Parameters

- **source** (*str*) – Source file in PNG format.
- **target** (*str*) – Output thumbnailfile in PNG format.

hepdata_lib.helpers.**execute_command**(*command*)

Execute shell command using subprocess. If executable does not exist, return False. For other errors raise RuntimeError. Else return True on success.

Parameters

command (*string*) – Command to execute.

hepdata_lib.helpers.**file_is_outdated**(*file_path*, *reference_file_path*)

Check if the given file is outdated compared to the reference file.

Also returns true if the reference file does not exist.

Parameters

- **file_path** (*str*) – Path to the file to check.
- **reference_file_path** (*str*) – Path to the reference file.

hepdata_lib.helpers.**find_all_matching**(*path*, *pattern*)

Utility function that works like 'find' in bash.

hepdata_lib.helpers.**get_number_precision**(*value*)

Get precision of an input value. Exact integer powers of 10 are assigned same precision of smaller numbers For example get_number_precision(10.0) = 1 get_number_precision(10.001) = 2 get_number_precision(9.999) = 1

hepdata_lib.helpers.**get_value_precision_wrt_reference**(*value*, *reference*)

relative precision of first argument with respect to the second one value and reference are both float and/or int value can be float when reference is an int and viceversa

: param value: first value : type value: float, int

: param reference: reference value (usually the uncertainty on value) : type reference: float, int

hepdata_lib.helpers.**relative_round**(*value*, *relative_digits*)

Rounds to a given relative precision

hepdata_lib.helpers.**round_value_and_uncertainty**(*cont*, *val_key='y'*, *unc_key='dy'*, *sig_digits_unc=2*)

round values and uncertainty according to the precision of the uncertainty, and also round uncertainty to a given number of significant digits Typical usage:

reader = RootFileReader("rootfile.root") data = reader.read_hist_1d("histogramName") round_value_and_uncertainty(data,"y","dy",2)

will round data["y"] to match the precision of data["dy"] for each element, after rounding each element of data["dy"] to 2 significant digits e.g. 26.5345 +/- 1.3456 –> 26.5 +/- 1.3

: param cont : dictionary as returned e.g. by RootFileReader::read_hist_1d() : type cont : dictionary

: param sig_digits_unc: how many significant digits used to round the uncertainty : type sig_digits_unc: integer

hepdata_lib.helpers.**round_value_and_uncertainty_to_decimals**(*cont*, *val_key='y'*, *unc_key='dy'*, *decimals=3*)

round values and uncertainty to some decimals default round to 3 digits after period possible use case: correlations where typical values are within -1,1

: param cont : dictionary as returned e.g. by RootFileReader::read_hist_1d() : type cont : dictionary

: param decimals: how many decimals for the rounding : type decimals: integer

hepdata_lib.helpers.**round_value_to_decimals**(*cont*, *key='y'*, *decimals=3*)

round all values in a dictionary to some decimals in one go default round to 3 digits after period possible use case: correlations where typical values are within -1,1

: param cont : dictionary as returned e.g. by RootFileReader::read_hist_1d() : type cont : dictionary

: param decimals: how many decimals for the rounding : type decimals: integer

hepdata_lib.helpers.**sanitize_value**(*value*)

Handle conversion of input types for internal storage.

Parameters
value (*string, int, or castable to float*) – User-side input value to sanitize.

Strings and integers are left alone, everything else is converted to float.

hepdata_lib utilities to interact with ROOT data formats.

class hepdata_lib.root_utils.**RootFileReader**(*tfile*)

Bases: object

Easily extract information from ROOT histograms, graphs, etc

read_graph(*path_to_graph*)

Extract lists of X and Y values from a TGraph.

Parameters
path_to_graph (*str*) – Absolute path in the current TFile.

Returns
dict – For a description of the contents, check the documentation of the get_graph_points function.

read_hist_1d(*path_to_hist*, ***kwargs*)

Read in a TH1.

Parameters

- **path_to_hist** (*str*) – Absolute path in the current TFile.
- ****kwargs** – See below

Keyword Arguments

- ***xlim*** **(tuple) –**
  limit x-axis range to consider (xmin, xmax)
- ***force_symmetric_errors*** **–**
  Force readout of symmetric errors instead of determining type automatically

Returns
dict – For a description of the contents, check the documentation of the get_hist_1d_points function

read_hist_2d(*path_to_hist*, ***kwargs*)

Read in a TH2.

Parameters

- **path_to_hist** (*str*) – Absolute path in the current TFile.
- ****kwargs** – See below

Keyword Arguments

- ***xlim*** **(tuple) –**
  limit x-axis range to consider (xmin, xmax)
- ***ylim*** **(tuple) –**
  limit y-axis range to consider (ymin, ymax)
- ***force_symmetric_errors* –**
  Force readout of symmetric errors instead of determining type automatically

Returns

dict – For a description of the contents, check the documentation of the get_hist_2d_points function

read_limit_tree(*path_to_tree='limit'*, *branchname_x='mh'*, *branchname_y='limit'*)

Read in CMS combine limit tree.

Parameters

- **path_to_tree** (*str*) – Absolute path in the current TFile
- **branchname_x** (*str*) – Name of the branch that identifies each of the toys/parameter points.
- **branchname_y** (*str*) – Name of the branch that contains the limit values.

Returns

list – Lists with 1+5 entries per toy/parameter point in the file. The entries correspond to the one number in the x branch and the five numbers in the y branch.

read_tree(*path_to_tree*, *branch_name*)

Extract a list of values from a tree branch.

Parameters

- **path_to_tree** (*str*) – Absolute path in the current TFile.
- **branch_name** (*str*) – Name of branch to read.

Returns

list – The values saved in the tree branch.

retrieve_object(*path_to_object*)

Generalized function to retrieve a TObject from a file.

There are three use cases: 1) The object is saved under the exact path given. In this case, the function behaves identically to TFile.Get. 2) The object is saved as a primitive in a TCanvas. In this case, the path has to be formatted as PATH_TO_CANVAS/NAME_OF_PRIMITIVE 3) The object is saved as a primitive in a TPad that is nested in a TCanvas. In this case, the path has to be formatted as CANVAS/PAD1/PAD2…/NAME_OF_PRIMITIVE

Parameters

path_to_object (*str.*) – Absolute path in current TFile.

Returns
TObject – The object corresponding to the given path.

property tfile

The TFile this reader reads from.

hepdata_lib.root_utils.**get_graph_points**(*graph*)

Extract lists of X and Y values from a TGraph.

Parameters
graph (*TGraph, TGraphErrors, TGraphAsymmErrors*) – The graph to extract values from.

Returns
dict – Lists of x, y values saved in dictionary (keys are "x" and "y"). If the input graph is a TGraphErrors (TGraphAsymmErrors), the dictionary also contains the errors (keys "dx" and "dy"). For symmetric errors, the errors are simply given as a list of values. For asymmetric errors, a list of tuples of (down,up) values is given.

hepdata_lib.root_utils.**get_hist_1d_points**(*hist*, ***kwargs*)

Get points from a TH1.

Parameters

- **hist** (*TH1D*) – Histogram to extract points from
- ****kwargs** – See below

Keyword Arguments

- ***xlim*** **(tuple) –**
  limit x-axis range to consider (xmin, xmax)
- ***force_symmetric_errors*** **–**
  Force readout of symmetric errors instead of determining type automatically

Returns
dict – Lists of x/y values saved in dictionary. Corresponding keys are "x" for the value of the bin center. The bin edges may be found under "x_edges" as a list of tuples (lower_edge, upper_edge). The bin contents are stored under the "y" key. Bin content errors are stored under the "dy" key as either a list of floats (symmetric case) or a list of down/up tuples (asymmetric). Symmetric errors are returned if the histogram error option TH1::GetBinErrorOption() returns TH1::kNormal.

hepdata_lib.root_utils.**get_hist_2d_points**(*hist*, ***kwargs*)

Get points from a TH2.

Parameters

- **hist** (*TH2D*) – Histogram to extract points from
- ****kwargs** – See below

Keyword Arguments

- ***xlim*** **(tuple) –**
  limit x-axis range to consider (xmin, xmax)
- ***ylim*** **(tuple) –**
  limit y-axis range to consider (ymin, ymax)
- ***force_symmetric_errors*** **–**
  Force readout of symmetric errors instead of determining type automatically

Returns

dict – Lists of x/y/z values saved in dictionary. Corresponding keys are “x”/”y” for the values of the bin center on the respective axis. The bin edges may be found under “x_edges” and “y_edges” as a list of tuples (lower_edge, upper_edge). The bin contents and errors are stored under the “z” key. Bin content errors are stored under the “dz” key as either a list of floats (symmetric case) or a list of down/up tuples (asymmetric). Symmetric errors are returned if the histogram error option TH1::GetBinErrorOption() returns TH1::kNormal.

CHAPTER

SIX

CONTRIBUTING

Here are a couple of details regarding the development of this library. Contributions are more than welcome!

Please see *Setup for developers* on how to set up the library for development.

- *Using bumpversion* to tag a new release
- *Uploading to PyPI*

6.1 Using bumpversion

bumpversion allows to update the library version consistently over all files. Please do not change the version manually, but use the following steps instead after a pull request has been merged into the `main` branch. Depending on the amount of changes, choose accordingly from:

- `patch` = `+0.0.1`
- `minor` = `+0.1.0`
- `major` = `+1.0.0`

Execute the following commands:

```
pip install --upgrade bumpversion
git checkout main
git pull
bumpversion patch # adjust accordingly
git push origin main --tags
```

The files in which the versions are updated as well as the current version can be found in the .bumpversion.cfg. You need appropriate rights for the repository to be able to push the tag.

6.2 Uploading to PyPI

Once a new version has been tagged, the package should be uploaded to the Python Package Index (PyPI). For the markdown formatting to work, `twine>=1.11.0` is required. Execute the following commands to create a source distribution and upload it:

```
pip install -U wheel
python setup.py sdist bdist_wheel
pip install -U twine
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

This uploads to the PyPI test server. Mind that you need to have an account for both the test and the production servers.

Install the package for testing:

```
pip install --index-url https://test.pypi.org/simple/ hepdata_lib
```

If everything is fine, upload to the production server:

```
twine upload dist/*
```

You should then find the new version at this location. You need to be a maintainer of the project for this to work. For more details please see the python packaging documentation.

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

H

M

N

R

S

T

U

V

W