PyHEP Tutorial

Alex Samuel

February 21, 2005

Abstract

This manual describes how to use the PyHEP Python tools for High Energy Physics. Rather than documenting all APIs exhaustively, it provides examples of common use cases, and describes APIs and object protocols in general terms. See the automatically-generated API documentation, doc comments, or the source code for detailed information about the APIs.

Warning: This document, and the software it describes, are incomplete. In particular, the APIs and protocols described here and implemented in the software may change, as the author gains experience with how the software is best used.

CONTENTS

CHAPTER

ONE

Introduction

This tutorial presents a tour of PyHEP's most important features, and shows how to perform some common tasks. We assume you are familiar with modern versions of the Python language, at least version 2.2.

PyHEP makes use of several newer Python features, such as true division and list comprehensions. While you don't strictly need to be familiar with these features to use PyHEP, some of them will help you use PyHEP to its greatest advantage. In particular, PyHEP is designed to support a functional programming style, which is well-suited for many common HEP computing tasks. Many advanced Python features support functional programming: the `map`, `reduce`, and `filter` built-in functions; iterators; `lambda` expressions; generators; and list comprehensions.

CHAPTER

TWO

Histograms

A histogram is a tool for measuring an *N*-dimensional statistical distribution by summarizing samples drawn from it.

The histogram divides a rectangular region of the *N*-dimensional samples space into rectangular sub-regions, called *bins*. Each bin records the number of recorded samples whose values fall in that sub-region; when a sample is added into the histogram, or *accumulated*, the bin's value is incremented.

The histogram designates an *axis* for each dimension of the sample space, which specifies the range of coordinate values in that dimension which the histogram will accept. Coordinate values that fall below or above that range are called *underflows* and *overflows*, respectively.

Each axis also specifies the edges of the bins in the corresponding dimension, which forms rectangular bins. For *evenly-binned* axes, the positions of the bin edges are evenly spaced along the range of the axis. For *unevenly-binned* axes, the bin edges may be positioned arbitrarily; this allows smaller bins to be used in regions of the coordinate value where samples are more likely to fall more densely.

In some cases, samples are not drawn from the distribution which is being measured, but from a modified distribution (for example, when importance sampling). The modified distribution can be transformed back to the original distribution by assigning each sample a *weight*. The value of the corresponding bin will then be increased by the sample weight. When no weight is specified, unit weight is assumed.

If the samples are statistically independent, the contents of the bins will provide an approximation of the probability density function from which the samples are drawn. This approximation is subject to statistical uncertainty or *error*. The error is drawn from a Poisson distribution; for large statistics, a Gaussian distribution is a good approximation.

2.1 Introductory example

This example creates a one-dimensional histogram with five evenly-spaced bins between zero and ten, fills some sample values into it, and dumps the resulting contents.

Import the histogram module.

```
>>> import hep.hist
```

Create a histogram object.

```
>>> histogram = hep.hist.Histogram1D(5, (0, 10))
```

Show the histogram object we just created.

```
>>> print histogram
Histogram(EvenlyBinnedAxis(5, (0, 10)), bin_type=int, error_model='poisson')
```

Accumulate samples into the histogram.

```
>>> for sample in (0, 4, 5, 6, 7, 7, 9, 10, 11):
```

```
...   histogram << sample
...
```

Dump the histogram contents.

```
>>> hep.hist.dump(histogram)
Histogram, 1 dimensions
int bins, ’poisson’ error model

axes: EvenlyBinnedAxis(5, (0, 10))

      axis 0             bin value / error
---------------------------------------------
[   None,      0)         0 +  1.148 -      0
[      0,      2)         1 +  1.360 -  1.000
[      2,      4)         0 +  1.148 -      0
[      4,      6)         2 +  1.519 -  2.000
[      6,      8)         3 +  1.724 -  2.143
[      8,     10)         1 +  1.360 -  1.000
[     10,   None)         2 +  1.519 -  2.000
```

2.2 Creating histograms

The class `hep.hist.Histogram` provides an arbitrary-dimensional histogram with binned axes. Optionally, a histogram can store errors (uncertainties) of the contents of each bin, or can compute these automatically from the bin contents.

To construct a `Histogram`, provide a specification of each axis, one for each dimension. You can specify an `Axis` object for each axis (described later), but the easiest way to specify an evenly-binned axis is with a tuple containing the number of bins, and a ’(lo, hi)’ pair specifying the low edge of the first bin and the high edge of the last bin. For example,

```
>>> import hep.hist
>>> one_d_histogram = hep.hist.Histogram((20, (0, 100)))
```

creates a one-dimensional histogram with 20 bins between values 0 and 100. Why all the parentheses? The outermost pair are for the function call; the next pair group parameters of the (single) axis; the innermost group the low and high boundaries of the axis range.

Similarly,

```
>>> two_d_histogram = hep.hist.Histogram((20, (-1.0, 1.0)), (100, (0.0, 10.0)))
```

creates a two-dimensional histograms with 20 bins on the first axis and 100 bins on the second axis.

To create a one-dimensional histogram you may use `Histogram1D`, which is equivalent except the axis’s parameters are specified directly as arguments instead of wrapped in a sequence, so that one pair of parentheses may be omitted. Thus, the first example above may be written,

```
>>> one_d_histogram = hep.hist.Histogram1D(20, (0, 100))
```

You can access the sequence of axes for a histogram from its `axes` attribute. You’ll see that the tuple arguments you specified have been converted into `EvenlyBinnedAxis` objects.

```
>>> two_d_histogram.axes
(EvenlyBinnedAxis(20, (-1.0, 1.0)), EvenlyBinnedAxis(100, (0.0, 10.0)))
```

For one-dimensional histograms, you may also use the `axis` attribute.

```
>>> one_d_histogram.axis
EvenlyBinnedAxis(20, (0, 100))
```

For evenly-binned axes such as these, the `range` and `number_of_bins` attributes contain the axis parameters. The `dimensions` attribute contains the number of dimensions of the histogram. This is always equal to the length of the `axes` attribute.

```
>>> one_d_histogram.axis.number_of_bins
20
>>> one_d_histogram.axis.range
(0, 100)
>>> one_d_histogram.dimensions
1
```

If you specify additional keyword arguments when creating the histogram, they are added as attributes to the new histogram object. For example,

```
>>> histogram = hep.hist.Histogram(
...   (40, (0.0, 10.0), "momentum", "GeV/c"),
...   (32, (0, 32), "track hits"),
...   title="drift chamber tracks")
>>> histogram.title
'drift chamber tracks'
```

This creates a two-dimensional histogram of 40 bins of track energy between 0 and 10 GeV/c, and 32 integer bins counting track hits. Note that this histogram has $(40 + 2) \times (32 + 2) = 1428$ bins, including overflow and underflow bins on each axis. The histogram has an additional attribute `title` whose value is `"drift chamber tracks"`. You can, of course, set or modify additional attributes like `title` after creating the histogram.

2.2.1 Axis type

By specifying the range for each axis, you also implicitly specify the numeric type for values along the axis. (If the low and high values you specify for the range are of different types, a common type is obtained by Python's standard `coerce` mechanism.) So, for instance, if you specify two integer values for the low and high boundaries for the axis, the axis will expect integer values, but if you specify one integer and one floating-point value, the axis will expect floating-point values.

You can find out the type of an axis by consulting its `type` attribute. For example,

```
>>> histogram = hep.hist.Histogram((20, (-1.0, 1.0)), (20, (-10, 10)))
>>> histogram.axes[0].type
<type 'float'>
>>> histogram.axes[1].type
<type 'int'>
```

If the type for an axis is `int` or `long` (because you specified integer or long values for the axis range), the difference between the high and low ends of the range must be an even multiple of the number of bins. If this is not desired, use `float` values for the axis. For example, this construction raises an exception:

```
>>> histogram = hep.hist.Histogram1D(8, (-50, 50))
[... stack trace ...]
ValueError: number of bins must be a divisor of axis range
```

because 50 - (-50) = 100 is not a multiple of 8. However, any of these will work:

```
>>> histogram = hep.hist.Histogram1D(10, (-50, 50))
>>> histogram = hep.hist.Histogram1D(8, (-60, 60))
>>> histogram = hep.hist.Histogram1D(8, (-50.0, 50.0))
```

2.2.2 Additional axis parameters

In addition to the number of bins and axis range, you may optionally provide additional elements to a tuple specifying an axis (or as the arguments to `Histogram1D`). The third argument is the string name of the axis quantity, and the fourth argument is a string describing the units of this quantity. For example,

```
h = hep.hist.Histogram1D(30, (0.05, 0.20), "mass", "GeV/$c^2$")
```

creates a histogram of 30 bins of mass between 0.05 and 0.2 GeV/c^2.

LaTeXmarkup may be used in the axis name and units; these will be rendered appropriately when plotting the histogram. See the section on LaTeXmarkup for details of the syntax.

2.2.3 Bin type and error model

A histogram stores for each bin the total number of samples, or the sum of weights of samples, that have fallen in this bin. The keyword argument *bin_type* specifies the numerical type used to store these totals. The histogram's `bin_type` attribute contains this type.

```
>>> histogram = hep.hist.Histogram1D(10, (-50, 50))
>>> histogram.bin_type
<type 'int'>
>>> histogram = hep.hist.Histogram1D(10, (-50, 50), bin_type=float)
>>> histogram.bin_type
<type 'float'>
```

By default, bin contents are stored as `int` values. *Therefore, if you plan to use non-integer weights, you must specify* `bin_type=float` *when creating the histogram!* Otherwise, weights will be truncated to integers when filling. If your weights are all less than one, you fill find your histogram to be empty.

A histogram can also estimate the statistical counting uncertainty on each bin. This error is represented by a 68.2% confidence interval around the bin value. The interval is represented by a pair of values specifying the sizes of the low and high "error bars", *e.g.* if the bin value is `20` and the errors are `(5.5, 4.5)`, the 68.2% confidence interval is the range (14.5, 24.5).

Several models are available are available that control how the errors are stored or computed:

- `"none"`: Each bin is assumed to have zero uncertainty.
- `"gaussian"`: The errors are computed assuming symmetrical Gaussian counting errors. The bin value is interpreted as a number of counts, and the low and high errors are both the square root of the bin content.
- `"poisson"`: The errors are computed assuming Poisson counting errors. The bin value is interpreted as a number of counts, and the low and high errors are chosen to cover 68.2% of the Poisson cumulative distribution around the bin value. The confidence interval is chosen to cover 34.1% on either side of the central value where possible. However, for a central value of zero, one, or two counts, this would produce a confidence interval with a lower edge below zero, so the lower edge is fixed at zero and the upper edge is chosen to capture 68.2%.
- `"symmetric"`: For each bin, the histogram stores a single value representing both the lower and upper errors. The error value may be specified explicitly for each bin with the `setBinError` method.

When a sample is accumulated into the histogram, the sample weight is added in quadrature to the bin error. The error value on a bin (used as both the upper and lower error) are is given by $\sigma = \sqrt{\sum wi^2}$ where wi are the sample weights accumulated into the bin.

- `"asymmetric"`: For each bin, the histogram stores two values representing the lower and upper errors. The two error values may be specified explicitly for each bin with the `setBinError` method.

  Errors are computed from weights as in the `"symmetric"` error model. The only difference is that you may specify manually different values for the lower and upper errors using the `setBinError` method.

Specify the error model for a histogram with the *error_model* keyword argument. The default is `"poisson"` if the bin type is `int` or `long`, or `"gaussian"` otherwise. If you will specify bin errors explicitly using the `setBinError` method, you must specify the `"symmetric"` or the `"asymmetric"` error model. The `error_model` attribute contains a histogram's error model.

For example,

```
>>> histogram = hep.hist.Histogram1D(10, (-50, 50))
>>> histogram.error_model
'poisson'
>>> histogram = hep.hist.Histogram1D(10, (-50, 50), bin_type=float)
>>> histogram.error_model
'gaussian'
>>> histogram = hep.hist.Histogram1D(10, (-50, 50), error_model="symmetric")
>>> histogram.error_model
'symmetric'
```

2.3 Other kinds of axes

The histogram axes we've used up to now have been evenly binned, *i.e.* all the bins on each axis are the same size. It is also possible to create a histogram with unevenly-binned axes, using the `hep.hist.UnevenlyBinnedAxis` class.

Create an `UnevenlyBinnedAxis` instance, specifying a list of bin edges as its argument. You can then specify this axis when creating a histogram. Note that each histogram axis is independently specified, so that one, several, or all may be unevenly binned.

For example, to create a 2-D histogram with an unevenly-binned x axis and one evenly-binned y axis,

```
>>> import hep.hist.axis
>>> x_axis = hep.hist.UnevenlyBinnedAxis(
...     [0, 2, 4, 5, 6, 8, 10, 15, 20], name="count")
>>> histogram = hep.hist.Histogram(x_axis,
...     (10, (0., 5.), "time", "sec"))
```

2.4 Filling histograms

To "fill" a histogram is to accumulate samples from the sample distribution into it. A sample is represented by a sequence of coordinate values, where the length of the sequence is equal to the number of dimensions of the histogram. Each item in the sequence is the coordinate value along the corresponding axis of the histogram.

For each bin, the histogram keeps track of the number of accumulated samples whose coordinates fall within the bin. Optionally, samples may be specified a "weight"; in that case, the histogram keeps track of the sum of weights of samples. The numeric type used to store the number of samples or sum of weights for each bin is given by the `bin_type` attribute. The histogram also tracks the total number of accumulations (the “number of entries”) that you have made.

The `accumulate` method accumulates an event into the histogram. Specify the coordinates of the sample, and optionally the event weight (which otherwise is taken to be unity). The left-shift operator `<<` is shorthand for `accumulate` with unit weight.

If any of the coordinate values passed to `accumulate` is `None`, the sample is not accumulated into the histogram and the number of entries is not changed.

For example, to accumulate a the sample whose coordinates are `x` and `y` into a two-dimensional histogram `histogram`,

```
>>> histogram.accumulate((x, y))
```

or

```
>>> histogram << (x, y)
```

To accumulate the same sample with weight `weight`,

```
>>> histogram.accumulate((x, y), weight)
```

For one-dimensional histograms, you may specify the sample coordinate by itself, instead of as a one-element sequence. So, to accumulate the sample with coordinate `x` into one-dimensional `histogram`, you may use any of these:

```
>>> histogram.accumulate((x, ))
>>> histogram.accumulate(x)
>>> histogram << (x, )
>>> histogram << x
```

Plotting histograms is described later in the plotting chapter. A quick way of displaying histogram contents in text format is with the `hep.hist.dump` function.

This script below creates a one-dimensional histogram with eleven integer bins between 2 and 12, inclusive, and fills the histogram with the result of simulating 1000 rolls of two dice.

```
import hep.hist
import random

histogram = hep.hist.Histogram1D(11, (2, 13), "rolls", bin_type=int)
for count in xrange(0, 1000):
    roll = random.randint(1, 6) + random.randint(1, 6)
    histogram << roll

hep.hist.dump(histogram)
```

2.5 Accessing histogram contents

You may specify a particular bin of a histogram with a “bin number”, which is a sequence of bin positions along successive axes. The length of the sequence is equal to the histogram’s number of dimensions. Each item is the index of the bin along the corresponding axis.

Along each axis, the coordinate in the bin number ranges between zero and one less than the number of bins on the axis. It may also take the values `"underflow"` and `"overflow"`, which denote the underflow and overflow bins, respectively

For example, consider this histogram:

```
>>> histogram = hep.hist.Histogram((20, (-1.0, 1.0)), (24, (0, 24)))
```

The corner bin numbers are for this histogram `(0, 0)`, `(9, 0)`, `(0, 23)`, and `(9, 23)`. The bin whose number is `(12, "underflow")` is for any samples whose first coordinate is between 0.2 and 0.3, and whose second coordinate is less than 0. The bin whose number is `("underflow", "overflow")` is for any sample whose first coordinate is less than -1 and whose second coordinate is greater than 24.

To get the bin number corresponding to a sample point, use the `map` method, passing the sample coordinates.

Just as with sample coordinates, for a one-dimensional histogram you may specify either the bin number as a one-element sequence, or simply the bin number along the (only) axis.

To obtain the contents of a bin, use the `getBinContent` method, passing the bin number. To obtain the 68.2% confidence interval on a bin, use the `getBinError` method, which returns two values specifying how far the interval extends below and above the central value.

For example,

```
>>> histogram = hep.hist.Histogram1D(10, (0.0, 1.0), error_model="gaussian")
>>> histogram.accumulate(0.64, 17)
>>> histogram.map(0.64)
(6,)
>>> histogram.getBinContent((6, ))
17
>>> histogram.getBinError((6, ))
(4.1231056256176606, 4.1231056256176606)
```

In the `"gaussian"` error model, the errors on the bin are the square root of the bin contents, here, $\sqrt{17}$=4.123106. Since the histogram is one-dimensional, we just as easily could have used,

```
>>> histogram.getBinContent(6)
```

To set the contents of a bin, use the `setBinContent` method, specifying the new value as the second argument. To set the error estimate on a bin, use the `setBinError` method and specify a pair for the sizes of lower and upper errors. Note that you may only call the `setBinError` method of a histogram with `"asymmetric"` or `"symmetric"` error model, and in the latter case, the average of the lower and upper error values you specify is used as the single symmetric error estimate.

To obtain the range of coordinate values spanned by a single bin, use the `getBinRange` method, passing the bin number. The return value is a sequence, each of whose items is a `(lo, hi)` pair of coordinate values along on axis spanned by the bin. For example, to print the bin range and value for bins in a one-dimensional histogram,

```
>>> for bin in range(histogram.axis.number_of_bins):
...     (lo, hi), = histogram.getBinRange(bin)
...     content = histogram.getBinContent(bin)
...     print "bin (%f,%f): %f" % (lo, hi, content)
```

Note that the return value from `getBinRange` is a one-element sequence, since the histogram is one-dimensional. The single element is the pair `(lo, hi)` of the bin's range along the histogram's axis.

A histogram's `number_of_samples` attribute contains the number of times the `accumulate` method (or the `<<` operator) was invoked.

2.6 More histogram operations

The following functions are provided in `hep.hist`. Invoke `help(hep.hist.`*function*`)` for a description of a function's parameters.

- The function `scale` produces a copy of a histogram with its contents scaled by a constant factor.

- The function `integrate` returns returns the sum over all bins of a histogram. Specify `overflows=True` to include underflow and overflow bins in the integral.
- The function `normalize` returns a copy of a histogram, scaled such that its integral is unity (or another constant value specified as the optional second argument).
- The functions `add` and `divide` create a new histogram by adding or dividing, respectively, the corresponding bins in two histograms. The histograms must have the same axis ranges and binning.
- The function `getMoment` computes the Nth moments of a histogram in each of its dimensions.
- The `mean` and `variance` functions compute those two statistics.
- The function `slice` produces an (N-1)-dimensional histogram by slicing or projecting out one dimension of an N-dimensional histogram.
- The function `rebin` produces a copy of a histogram with groups of adjacent bins combined together.
- The function `transform` transforms a histogram axis by an arbitrary monotonically increasing function by creating a copy of the histogram with adjusted bin boundaries.
- The function `getRange` determines the range of bin values in a histogram. Optionally, the range can accommodate the error intervals on each bin.
- The function `dump` prints the contents of a histogram to standard output or another file.
- The function `project` accumulates samples simultaneously into several histograms from an array of sample events. This function is described later in the chapter on tables.

You may also add or divide two histogram with the ordinary addition and division operators, respectively, and you may scale a histogram with the ordinary multiplication operator. For example,

```
>>> combined_histogram = 3 * histogram1 + histogram2
```

To iterate over all bins in a histogram, use these functions. Each takes an optional second argument `overflow`; if true, underflow and underflow bins are included (by default false).

- `AxisIterator` takes an `Axis`. It returns an iterator that yields the bin numbers for that axis.
- `AxesIterator` takes a sequence of `Axis` objects, such as the value of a histogram's `axes` attribute. It return s an iterator that yields the bin numbers for the multidimensional bin space specified by the axes.
- `BinValueIterator` takes a histogram. It returns an iterator that yields the contents of each bin in the histogram.
- `BinErrorIterator` takes a histogram. It returns an iterator that yields the error estimate on each bin in the histogram.
- `BinIterator` takes a histogram. It returns an iterator that yields triplets `(bin_number, contents, error)` for each bin in the histogram.

For example, the code below shows how you might find the largest bin value in a histogram, including underflow and overflow bins. (You could also use the `getRange` function.)

```
>>> max(hep.hist.BinValueIterator(histogram, overflow=True))
```

2.7 Scatter plots

A `hep.hist.Scatter` object collects sample points from a bivariate distribution. The samples are typically displayed as a “scatter plot”.

The sample points are not binned in any way; rather, the two coordinates of each samples is stored. Unlike with a histogram, the memory usage of a `Scatter` object increases as additional samples are accumulated. However, the resulting object contains the full covariance information for the two coordinate variables, and can be plotted precisely.

To create a `Scatter`, specify information about the two axes. Each may be an instance of `hep.hist.Axis`, or more simply a sequence ’(type, name, units)’, where ’type’ is the Python type of coordinate values along this axis, and ’name’ and ’units’ are strings describing the coordinate values (which may be omitted). If no axes are specified, they are taken to have ’float’ coordinates

For example, this code creates a `Scatter` object.

```
>>> scatter = hep.hist.Scatter(
...     (float, "energy", "GeV"), (int, "number of hits"),
...     title="candidate tracks")
```

Use the `accumulate` method to add a sample to the `Scatter`. The argument must be a two-element sequence containing the two coordinates. The left-shift operator `<<` is a synonym for `accumulate`. The sample values are stored in a sequence attribute `points`.

For example,

```
>>> for track in tracks:
...     scatter << (track.energy, track.number_of_hits)
...
```

CHAPTER

THREE

Tables

PyHEP provides a flat-format database facility, similar to “n-tuples” in other HEP analysis systems.

A PyHEP table is similar to a table in a typical database system. A table is a sequence of rows, each of which has the same number of columns and values for each column of the same type. The number and types of columns constitutes the table’s schema.

PyHEP’s tables are subject to the following restrictions:

- A table’s schema is set when it is created, and cannot subsequently be changed. (Creating another table with a different schema using the same data is straightforward.)
- The size in bytes of each table row is a constant. This limits the types of table columns, but allows the implementation of fast seeks in a large table.
- Rows are appended to the table. An entire row must be appended at one time. Rows may not be inserted into the middle of a table, modified, or removed.

3.1 Table implementations

Several table implementations are provided. Other than for creating new tables or opening existing tables, the interfaces of these implementations are identical.

1. A extension-class implementation written in C++ in the `hep.table` module. The table resides in a disk file, and rows are loaded into memory as needed. The implementation is simple and efficient. Other programs may write and access these tables via a simple C++ interface.
2. An implementation which uses HBOOK n-tuples (column-wise or row-wise) stored in HBOOK files. Not all features of HBOOK n-tuples are supported.
3. An implementation which uses ROOT tress stored in ROOT files. Not all features of ROOT trees are supported.

The extension-class implementation in `hep.table` is used in this tutorial.

3.2 Creating and filling tables

A table schema is represented by an instance of `hep.table.Schema`. The schema collects together the definitions of the columns in the table. Each column is identified by a name, which is a string composed of letters, digits, and underscores.

A new instance of `Schema` has no columns. Add columns using the `addColumn` method, specifying the column name and type. For instance,

```
>>> import hep.table
>>> schema = hep.table.Schema()
>>> schema.addColumn("energy", "float64")
>>> schema.addColumn("momentum", "float64")
>>> schema.addColumn("hits", "int32")
```

The second argument to `addColumn` specifies the storage format used for values in that column. These column types are supported in `hep.table` (note that other table implementations may not support all of these):

- three signed integer types: `"int8"`, `"int16"`, and `"int32"`
- two floating-point types, `"float32"` and `"float64"`
- two floating-point complex types, `"complex64"` and `"complex128"`

More concisely, you may specify columns as keyword arguments, so the above schema may be constructed with,

```
>>> schema = hep.table.Schema(energy="float64", momentum="float64", hits="int32")
```

You can also load or save schemas in an XML file format using `hep.table.loadSchema` and `hep.table.saveSchema`. The schema above would be represented by the XML file

```
<?xml version="1.0" ?>
<schema>
 <column name="energy" type="float64"/>
 <column name="momentum" type="float64"/>
 <column name="hits" type="int32"/>
</schema>
```

Call the `create` function to create a new table. The parameters will vary for each implementation; for the `hep.table` implementation, the parameters are the file name for the new table, and the table's schema.

```
>>> table = hep.table.create("test.table", schema)
```

The return value is a *connection* to the newly-created table, which resides on disk.

To add a row to the table, call the table's `append` method. You may pass it a mapping argument (such as a dictionary) that associates column values with column names, and/or you may provide column values as keyword arguments. One way or another, you must specify values for all columns in the table. The return value of `append` is the index of the newly-appended row.

For instance, to add a row to the table created above,

```
>>> row = {
...   "energy": 2.7746,
...   "momentum": 1.8725,
...   "hits": 17,
>>> }
>>> table.append(row)
```

or equivalently,

```
>>> table.append(energy=2.7746, momentum=1.8725, hits=17)
```

3.2.1 Example: creating a table from a text file

The script below converts a table of values in a text file into a PyHEP table. The script assumes that the file contains floating-point values only, except that the first line of the text file contains headings that will be used as the names of the columns in the table.

```
import hep.table

def textFileToTable(input_file_name, table_file_name):
    """Convert a text file containing tabular values to a table.

    'input_file_name' -- The file name of a text file containing a table
    of floating-point values.  The first line is assumed to contain the
    column names.  All additional lines are assumed to contain values
    for each column.

    'table_file_name' -- The file name of the table to create."""

    lines = iter(file(input_file_name))

    # Read the first line in the file, and split it into column names.
    heading_row = lines.next()
    column_names = heading_row.split()

    # Construct the schema from these column names.
    schema = hep.table.Schema()
    for column_name in column_names:
        schema.addColumn(column_name, "float32")
    # Create the table.
    table = hep.table.create(table_file_name, schema)

    # Scan over remaining lines in the file.
    for line in lines:
        # Split the line into values and convert them to numbers.
        values = map(float, line.split())
        # Make sure the line contains the right number of values.
        if len(values) != len(column_names):
            raise RuntimeError, "format error"
        # Construct a dictionary mapping column names to values.
        row = dict(zip(column_names, values))
        # Add the row to the table.
        table.append(row)

    del row, table


if __name__ == "__main__":
    # This file was invoked as a script.  Convert a text file as
    # specified on the command line.
    import sys
    textFileToTable(sys.argv[1], sys.argv[2])
```

Here is a table containing parameters for tracks measured in a detector. The first column is the track's energy; the other three are the x, y, and z components of momentum.

```
energy          p_x             p_y             p_z
1.180304e+00    -4.751910e-01   -7.889777e-01   -7.305207e-01
2.080963e+00    1.452333e+00    -4.622423e-01   1.412906e+00
1.743646e+00    1.421417e+00    -3.871017e-01   9.267482e-01
1.541235e+00    -1.059296e-01   -1.395949e+00   -6.358849e-01
1.617172e+00    1.401164e+00    -8.951803e-03   8.004645e-01
2.035846e+00    -4.445600e-02   1.965886e+00    5.165461e-01
1.202767e+00    -1.163040e+00   1.832017e-01    -2.219467e-01
```

```
2.567302e+00   -2.095504e+00  8.135615e-02   1.477200e+00
2.603921e+00   -9.120621e-01  -1.848932e+00  1.587083e+00
2.523255e+00   2.622356e-01   -1.991126e+00  1.523911e+00
2.233162e+00   -1.252426e+00  1.779797e+00   4.894894e-01
2.276338e+00   -1.044715e+00  5.443484e-01   1.944943e+00
2.208343e+00   1.969946e+00   -3.294630e-01  9.361521e-01
1.679185e+00   1.460185e+00   7.744986e-01   2.766050e-01
1.097246e+00   2.343778e-01   -1.065456e+00  -5.151948e-02
1.192835e+00   -9.905712e-01  5.303552e-01   3.862431e-01
3.180005e+00   -7.034522e-01  1.516622e+00   2.703014e+00
2.583735e+00   7.747896e-01   -1.303546e+00  2.089256e+00
1.953654e+00   1.482216e-01   1.877128e+00   5.099221e-01
2.217959e+00   5.964227e-01   1.990123e+00   7.693304e-01
2.307544e+00   -1.608457e+00  5.832562e-01   1.544757e+00
1.705716e+00   -4.596864e-01  1.543895e+00   5.508014e-01
2.343682e+00   1.468491e+00   -4.265070e-01  1.772938e+00
2.371860e+00   -1.794360e+00  1.057422e+00   1.129905e+00
1.370261e+00   4.528853e-01   1.226287e+00   -3.969452e-01
2.294908e+00   2.271728e-02   -2.027759e+00  1.069166e+00
2.875299e+00   1.594167e+00   1.849570e+00   1.514564e+00
2.917657e+00   1.757843e+00   5.340980e-01   2.264131e+00
2.766238e+00   -1.591074e+00  1.628313e+00   1.567799e+00
2.030203e+00   -1.662516e+00  -4.085008e-01  1.086154e+00
2.698105e+00   -5.955992e-01  -1.737854e+00  1.973255e+00
2.095454e+00   5.345570e-01   1.171157e+00   1.649971e+00
2.118398e+00   1.290140e+00   1.337788e+00   1.011093e+00
1.129607e+00   -6.164498e-01  7.138193e-01   -6.126172e-01
1.987723e+00   7.570301e-01   -1.814894e+00  2.700781e-01
2.018107e+00   -6.774516e-02  -1.672229e+00  1.122789e+00
```

If you save the script as ‘txt2table.py’ and the table as ‘tracks.table’, you would invoke this command to convert it to a table:

```
> python txt2table.py tracks.txt tracks.table
```

3.3 Using tables

To open an existing table, use the `open` function. The first argument is the table file name. The second argument is the mode in which to open the table, similar to the built-in `open` function: `"r"` (the default) for read-only, `"w"` for write.

For instance, to open the table we created in the last section,

```
>>> tracks = hep.table.open("tracks.table")
```

The table object is a sequence whose elements are the rows. Each row is has a row index, which is equal to its position in the table. Thus, the `len` function returns the number of rows in the table.

```
>>> number_of_tracks = len(tracks)
```

To access a row by its row index, use the normal sequence index notation. This returns a `Row` object, which is a mapping from column names to values.

```
>>> track = tracks[19]
>>> print track["energy"]
```

The table's `schema` attribute contains the table's schema; the schema's `column` attribute is a sequence of the columns in the schema (in unspecified order). You can examine these directly, or use the `dumpSchema` function to print out the schema.

```
>>> hep.table.dumpSchema(tracks.schema)
name             type
----------------------------
energy           float32
p_x              float32
p_y              float32
p_z              float32
```

3.4 Iterating over rows

For most HEP applications, the rows of a table represent independent measurements, and are processed sequentially. An *iterator* represents this sequential processing of rows. Using iterators instead of indexed looping constructs simplifies code, opens up powerful functional-programming methods, and enables automatic optimization of independent operations on rows.

Since a table satisfies the Python sequence protocol, you can produce an iterator over its elements (*i.e.* rows) with the `iter` function. The Python `for` construction does even this automatically. The simplest idiom for processing rows in a table sequentially looks like this:

```
>>> total_energy = 0
>>> for track in tracks:
...     total_energy += track["energy"]
...
>>> print total_energy
74.7496351004
```

Since `iter(tracks)` is an iterator rather than a sequence of all rows, each row is loaded from disk into memory only when needed in the loop, and is subsequently deleted. This is critical for scanning over large tables. (Note that after the loop completes, the last row remains loaded, until the variable *track* is deleted or goes out of scope. Also, the table object itself is deleted only after any variables that refer to it, as well as any variables that refer to any of its rows, are deleted.)

Table iterators may be used to iterate over a subset of rows in a sequence. Most obviously, you could implement this by using a conditional in the loop. For instance, to print the energy of each track with an energy greater than 2.5,

```
>>> for track in tracks:
...     if track["energy"] > 2.5:
...         print track["energy"]
...
```

While this is straightforward, it forces PyHEP to examine each row every time the program is run. By using the selection in the iterator, PyHEP can optimize the selection process, often significantly. The selection criterion can be any boolean-valued expression involving the values in the row. The selection expression is evaluated for each row, and if the result is true, the iterator yields that row; otherwise, that row is skipped. (This is semantically similar to the first argument of the built-in `filter` function.) The expression can be a string containing an ordinary Python expression using column names as if they were variable name (with certain limitations and special features), or may be specified in other ways. Expressions are discussed in greater detail later.

For instance, the same high-energy tracks can be obtained using the selection expression `"energy > 2.5"`. Notice that *energy* appears in this expression as if it were a variable defined when the expression is evaluated. The `select` method returns an iterator which yields only rows for which the selection is true.

```
>>> for track in tracks.select("energy > 2.5"):
...     print track["energy"]
...
```

Python's list comprehensions provide a handy method for collecting values from a table. For instance, to enumerate all values of *energy* above 2.5 instead of merely printing them,

```
>>> energies = [ track["energy"] for track in tracks.select("energy > 2.5") ]
```

Note here that `tracks.select("energy > 2.5")` returns an iterator object, so it may only be used (i.e. iterated over) once. However, if you really want a sequence of `Row` objects, you can use the `list` or `tuple` functions to expand an iterator into an actual sequence, as in

```
>>> high_energy_tracks = list(tracks.select("energy > 2.5"))
```

Such a sequence consumes more resources than an iterator. You can iterate over such a sequence repeatedly, or perform other sequence operations.

3.5 Projecting histograms from tables

If you have a sequence or iterator of values, you can use the built-in `map` function to accumulate them into a histogram. For instance, if *energies* is a sequence of energy values, you could fill them into a histogram using,

```
>>> energy_hist = hep.hist.Histogram1D(20, (0.0, 5.0))
>>> map(energy_hist.accumulate, energies)
```

Generally, though, you will want to project many histograms at one time from a table. Use the `hep.hist.project` function to project multiple histograms in a single scan over a table. Pass an iterable over the table rows to project—the table itself, or an iterator constructed with the `select` method—and a sequence of histogram objects. Each histogram should have an attribute `expression`, which is the expression whose value is accumulated into the histogram for each table row. The expression are similar to those used with the `select` method, except that their values should be numerical instead of boolean.

For example, this script projects three histograms – energy, transverse momentum, and invariant mass, from the table of tracks we constructed previously. Only high-energy tracks (those with energy above 2.5) are included. The dictionary containing the histograms is stored in a standard pickle file.

```
from   hep.hist import Histogram, Histogram1D, project
import hep.table
import pickle

histograms = {
    "energy":   Histogram1D(20, (0.0, 5.0), "energy", "GeV",
                expression="energy"),

    "pt":       Histogram1D(20, (1.0, 3.0), "p_T", "GeV/c",
                expression="hypot(p_x, p_y)"),

    "mass":     Histogram1D(20, (0.0, 0.2), "mass", "GeV/c^2",
                expression="sqrt(energy**2 - p_x**2 - p_y**2 - p_z**2)"),

    "px_vs_py": Histogram((8, (-2.0, 2.0), "p_x", "GeV/c"),
                          (8, (-2.0, 2.0), "p_y", "GeV/c"),
                expression="p_x, p_y"),
    }
```

```
tracks = hep.table.open("tracks.table")
project(tracks.select("energy > 2.5"), histograms.values())
pickle.dump(histograms, file("histograms.pickle", "w"))
```

Observe that the last histogram is two-dimensional, and its expression specifies the two coordinates of the sample using a comma expression.

With this scheme, you can determine later how the values in a histogram were computed, by checking its `expression` attribute.

3.6 Using expressions with tables

Expressions are described in more detail in a later chapter. This section presents techniques for optimizing expressions used with tables.

As described above, you can use Python expressions encoded in character strings with `select` method and `hep.hist.project` function. When such an expressions are evaluated, unbound symbols (*i.e.* variables) are bound to the value of the corresponding column.

Since a table row satisfies the Python mapping protocol, you can pass a table row directly to an expression's `evaluate` method. For example, this constructs an expression object to compute the transverse momentum of a track in the tracks table constructed above.

```
>>> import hep.expr
>>> p_t = hep.expr.asExpression("hypot(p_x, p_y)")
```

Its `evaluate` method computes transverse momentum for a track by binding *p_x* and *p_y* to the corresponding values of the table row.

```
>>> print p_t.evaluate(tracks[0])
0.92102783203
```

To find the largest transverse momentum in the track table,

```
>>> print max(map(p_t.evaluate, tracks))
2.44177757441
```

The chapter on expressions, below, describes how to compile an expression into a format for faster evaluation. When the expression is used with a table, you can produce an even faster version by compiling it with the table's `compile` method. This sets the symbol types correctly based on the table's schema, and applies additional optimizations specific to tables. For example,

```
>>> p_t = tracks.compile("hypot(p_x, p_y)")
>>> print p_t.evaluate(tracks[0])
0.92102783203
```

When you evaluate an expression on many rows of a large table, performance will be substantially better if you compile the expression first for that table. Note that an expression compiled for a table should only be used with that table.

3.6.1 Caching expressions in tables

You can also configure a table to cache the results of evaluating a boolean expression on the rows. The first time you evaluate the expression on a row, the row is loaded and the expression is evaluated as usual. On subsequent times, the table reuses the cached value of the expression, instead of reloading the row and re-evaluating the expression.

To instruct a table to cache the value of an expression, pass the expression to the table's `cache` method. Do not pass a compiled expression; pass the uncompiled form instead. You may pass a string or function here as well. Once you add the expression to the table's cache, the expression cache will automatically be used when you compile the expression or use it with `select` and other table functions.

For example, suppose your table file 'tracks.table' was extremely large, and you expect to select repeatedly all tracks with energy above 2.5. You could cache this selection expression like this:

```
>>> cut = "energy > 2.5"
>>> tracks.cache(cut)
```

```
>>> cut = tracks.compile(cut)
```

Optionally, compile the expression and evaluate it on each row once, to fill the cache.

```
>>> cut = table.compile(cut)
>>> for track in tracks:
...   cut.evaluate(track)
...
```

Once you have added a cached expression to the table, the cache will be used automatically whenever you compile the expression for that table: the compiled `cut` above uses the cache. The cache will also be used for subexpressions, so if you were to compile the expression `"mass < 1 and energy > 2.5"` would use the cache, too.

3.6.2 Row types

As we have seen above, the object representing one row of a table satisfies Python's read-only mapping protocol: it maps the name of a column to the corresponding value in the row. While in many ways, they behave like ordinary Python dictionaries (for instance, they support the `keys` and `items` methods), they actually instances of the `hep.table.RowDict` class.

```
>>> track = tracks[0]
>>> print type(track)
<type 'RowDict'>
```

If you ever need an actual dictionary object containing the values in a row, Python will produce that for you:

```
>>> dict(track)
{'energy': 1.1803040504455566, 'p_z': -0.73052072525024414,
 'p_x': -0.47519099712371826, 'p_y': -0.78897768259048462}
```

For some applications, however, it is more convenient to use a different interface to access row data. You can specify another type to use for row objects by setting the table's `row_type` attribute (or with the *row_type* keyword argument to `hep.table.open`). The default value, as you have seen, is `hep.table.RowDict`.

PyHEP includes a second row implementation, `hep.table.RowObject`, which provides access to row values as object attributes instead of items in a map. The row has an attribute named for each column in the table, and the value of the attribute is the corresponding value in the row.

For example,

```
>>> tracks.row_type = hep.table.RowObject
>>> track = tracks[0]
>>> print track.p_x, track.p_y, track.p_z
-0.475190997124 -0.78897768259 -0.73052072525
```

You may also derive a subclass from `RowObject` and use that as your table's row type. This is very handy for adding additional methods, get-set attributes, etc. to the row, for instance to compute derived values.

For example, you could create a `Track` class that provides the mass and scalar momentum as "attributes" that are computed dynamically from the row's contents.

```
>>> from hep.num import hypot
>>> from math import sqrt
>>> class Track(hep.table.RowObject):
...   momentum = property(lambda self: hypot(self.p_x, self.p_y, self.p_z))
...   mass = property(lambda self: sqrt(self.energy**2 - self.momentum**2))
...
```

Now set this class as the row type.

```
>>> tracks.row_type = Track
>>> track = tracks[0]
>>> print type(track)
<class '__main__.Track'>
```

You can now access the members of `Track`:

```
>>> print track.momentum
1.17556488438
>>> print track.mass
0.105663873224
```

Setting a table's row type to `RowObject` or a subclass will not break evaluation of compiled expressions on row objects. Expressions look for a `get` method, which is provided by both `RowDict` and `RowObject`.

Note that computing complicated derived values in this way is less efficient than using compiled expressions, as described above. However, you can create methods or get-set members that evaluate compiled expressions.

Only subclasses of `RowDict` and `RowObject` may be used as a table's `row_type`.

3.7 More table functions

The function `hep.table.project` is a generalization of `hep.hist.project`. Like the latter, it iterates over table rows and evaluates a set of expressions on each row. Instead of accumulating the expression values into histograms, passes them to arbitrary functions. Invoke `help(hep.table.project)` for usage information.

The `hep.table.Chain` class concatenates multiple tables into one. Simply pass the tables as arguments. Of course, any columns accessed in the chain must appear in all the included tables. For example, this code chains together all table files (files with the ".table" extension) in the current working directory.

```
>>> import os
>>> tables = [ hep.table.open(path)
...            for path in os.listdir(".")
...            if path.endswith(".table") ]
>>> chain = hep.table.Chain(*tables)
```

CHAPTER

FOUR

Expressions

We have already seen expressions used as predicates in a table's `select` method, and as formulas for computing the values to accumulate into histograms. Generally, operations using expressions execute much faster than the same logic coded directly as Python, since expressions are compiled for the specific table into a special format from which they are evaluated very quickly.

4.1 Building and evaluating expressions

To parse an expression into a Python expression object representing its parse tree, use `hep.expr.asExpression`.

```
>>> import hep.expr
>>> ex = hep.expr.asExpression("p_x ** 2 + p_y ** 2")
```

The expression object's string representation looks similar to the original formula:

```
>>> print ex
(p_x ** 2) + (p_y ** 2)
```

The expression object's `repr` shows its tree structure:

```
>>> print repr(ex)
Add(Power(Symbol('p_x', None), Constant(2)), Power(Symbol('p_y', None), Constant(2)))
```

To evaluate an expression, you must provide the values of all symbols. For the expression above, the symbols *p_x* and *p_y* must be specified. You may either call the expression object directly, passing symbol values as keyword arguments:

```
>>> ex(p_x=1, p_y=2)
5.0
```

or you may call its `evaluate` method with a map providing values of the symbols mentioned in the expression:

```
>>> ex.evaluate({"p_x": 1, "p_y": 2})
5.0
```

4.2 Compiling expressions

Expression objects, as well as expression coded directly in Python, execute quite a bit slower than similar mathematical expressions implemented in a compiled language like C or C++. However, PyHEP can often execute an expressions much faster, by compiling it to an internal binary format and then evaluating it with an optimized, stack-based evaluator implemented in C++.

To compile an expression to this optimized form, use `hep.expr.compile`. It returns an expression object that can be used the same way as the original expression, but which executes faster.

```
>>> cex = hep.expr.compile(ex)
>>> cex(p_x=1, p_y=2)
5.0
```

There is no need to use `asExpression` before `compile`; just pass it the expression formula directly.

```
>>> cex = hep.expr.compile("p_x ** 2 + p_y ** 2")
>>> cex(p_x=1, p_y=2)
5.0
```

4.2.1 Expression types

PyHEP attempts to determine the numeric type of the result of an expression. To do this, it needs to know the types of the symbol values in the expression. For constants, this is obvious, but since Python is an untyped language, the expression compiler cannot automatically determine the types of symbolic names in an expression, and treats them as generic objects.

PyHEP expressions understand the types `int`, `float`, and `complex`. The value `None` indicates that the type is not known, so the value should be treated as a generic Python object.

For example, in these expressions, PyHEP can infer the type of the expression value entirely from the types of constants.

```
>>> print hep.expr.asExpression("10 + 12.5").type
<type ’float’>
>>> print hep.expr.asExpression("3 ** 4").type
<type ’int’>
```

However, a symbol’s type is assumed to be generic, so the whole expression’s type cannot be inferred.

```
>>> print hep.expr.asExpression("2 * c + 10").type
None
```

The expression compiler can do a much better job if it knows the numerical type of the expression’s symbols. When you call `compile`, you can specify the types of symbols as keyword arguments. For example,

```
>>> cex = hep.expr.compile("2 * c + 10", c=int)
>>> print cex.type
<type ’int’>
```

If you provide a second non-keyword argument, this type is used as the default for all symbols in the expression.

```
>>> cex = hep.expr.compile("a**2 + b**2 + c**2", float)
>>> print cex.type
<type ’float’>
```

By specifying symbol types, you can construct compiled expressions that execute much faster than expressions with generic types.

You can also construct an uncompiled expression with types specified for symbols using the `hep.expr.setTypes`, `hep.expr.setTypesFrom`, and `hep.expr.setTypesFixed` functions.

4.3 Using expressions with tables

Since a table row object is a map from column names to values, you may specify a row object as the argument to `evaluate`. In the expression, the name of a column in the table is replaced by the corresponding value in that row. Using the 'tracks.table' table we created earlier,

```
>>> import hep.table
>>> tracks = hep.table.open("tracks.table")
>>> print ex.evaluate(tracks[0])
0.848292267373
```

This works well for small numbers of rows. However, if the expression is to be evaluated on a large number of rows in the same table, it should be compiled. Use the table's `compile` method, which sets the symbol types according to the table's schema and performs other necessary expansions before compiling the expression. The compiled expression object behaves just like the original expression object, except that it runs faster.

```
>>> cm = tracks.compile(ex)
>>> print cm.evaluate(tracks[0])
0.848292267373
```

You may also pass an expression as a string to `compile`. Functions provided by PyHEP which work with expressions, such as a table's `select` method or `hep.hist.project` will compile expressions automatically, where possible.

4.4 Expression syntax

An expression is specified using Python's ordinary expression syntax, with the following assumptions:

- Arbitrary names may be used in expressions as variable quantities, functions, etc. Other than the built-in names listed below, all names must be resolved when the expression is evaluated.
- The forward-slash operator for integers is true division, i.e. it produces a `float` quotient. Use the double forward-slash operator (e.g. "x // 3") to obtain the C-style truncated integer division.

The following names are recognized in expressions:

- Built-in Python constants `True`, `False`, and `None`.
- Built-in Python types `int`, `float`, `complex`, and `bool`.
- Built-in Python functions `abs`, `min`, and `max`.
- Constants from the `math` module: `e` and `pi`.
- Functions from the `math` module: `acos`, `asin`, `atan`, `atan2`, `ceil`, `cos`, `cosh`, `exp`, `floor`, `log`, `sin`, `sinh`, `sqrt`, `tan`, and `tanh`.
- From the `hep.lorentz` module, `Frame` and `lab`.

In addition, expressions may use these numerical convenience functions. (They are also available in Python programs in the `hep.num` module.)

`gaussian`(*mu, sigma, x*)
: Returns the probability density at *x* from a gaussian PDF with mean *mu* and standard deviation *sigma*.

`get_bit`(*value, bit*)
: Returns true iff. bit *bit* in *value* is set.

hypot(**terms*)
A generalization of `math.hypot` to arbitrary number of arguments. Returns the square root of the sum of the squares of its arguments.

if_then(*condition*, *value_if_true*, *value_if_false*)
Returns *value_if_true* if *condition* is true, *value_if_false* otherwise. Note that in a compiled expression (only), the second and third arguments are evaluated lazily, so that if *condition* is true, *value_if_false* is not evaluated, and otherwise *value_if_true* is not evaluated.

in_range(*min_value*, *value*, *max_value*)
Returns true if *min_value* is less than or equal to *value* and *value* is less than *max_value*.

near(*central_value*, *half_interval*, *value*)
Returns true if the absolute difference between *central_value* and *value* is less than *half_interval*.

4.5 Other ways to make expressions

You may specify a constant instead of a string when constructing an expression with `asExpression` or `compile`. The resulting expression simply returns the constant.

```
>>> ex = hep.expr.asExpression(15)
>>> print ex
15
>>> print ex.type
<type 'int'>
```

You may also specify a function that takes only positional arguments. The resulting expression calls this function, using symbols for the function arguments matching the parameter names in the function definition. If the function has a parameters `xyz`, the expression will evaluate the symbol `xyz` and call the function with this value. For example,

```
>>> def foo(x, a):
...   return x ** a
...
>>> ex = hep.expr.asExpression(foo)
>>> print ex
foo(x, a)
>>> ex(a=8, x=2)
256
```

The type of the value returned from such a function is not known.

```
>>> ex = hep.expr.asExpression(foo)
>>> print ex.type
None
```

You may specify it by attaching an attribute `type` to the function, containing the expected type of the function's return value.

```
>>> foo.type = float
>>> ex = hep.expr.asExpression(foo)
>>> print ex.type
<type 'float'>
```

You may also construct expressions pragmatically from the classes used by PyHEP to represent expressions internally. Each class represents a single operation. Invoke `help(hep.expr.classes)` for a list of these classes and their interfaces.

Here's an example to give you an idea.

```
>>> from hep.expr import Add, Divide, Constant, Symbol
>>> mean = Divide(Add(Symbol("a"), Symbol("b")), Constant(2))
>>> print mean
(a + b) / 2
>>> mean(a=16, b=20)
18.0
```

CHAPTER

FIVE

Files and directories

PyHEP provides a uniform interface for accessing and manipulating directories and their contents. The same interface can be used not only with file system directories, but directories in Root and HBOOK files as well.

Directory objects satisfy Python's map protocol, very similar to built-in `dict` objects. The keys in a directory are the names of items in the directory. To find the names in a directory, simply call the `keys` method. For supported file types (discussed below), the corresponding values are the contents of the files. To load a file, simply get the value using the subscript operator (square brackets) or `get` method.

Here are some examples of using PyHEP's directory objects. Suppose we are in a directory with these contents, which are various types of files containing

```
> ls -l
total 16
-rw-r-----  1 samuel samuel 1257 Dec  6 15:39 histogram.pickle
-rw-r-----  1 samuel samuel 3980 Dec  6 15:39 plot.pickle
-rw-r-----  1 samuel samuel   50 Dec  7 19:11 readme.txt
drwxr-x---  2 samuel samuel 4096 Dec  6 15:43 recodata
> ls -l recodata/
total 12
-rw-r-----  1 samuel samuel 4384 Dec  6 15:43 reco.root
-rw-r-----  1 samuel samuel  683 Dec  6 15:34 tracks.table
```

Now, let's start Python. First, we import the file system directory module, `hep.fs`, and construct a directory object, an instance of `FileSystemDirectory`, for the current working directory.

```
>>> import hep.fs
>>> cwd = hep.fs.getdir(".")
>>> cwd
FileSystemDirectory('/home/samuel/data', writable=True)
```

Using the `keys` method, we can get the names of files in the directory.

```
>>> print cwd.keys()
['readme.txt', 'plot.pickle', 'recodata', 'histogram.pickle']
```

Directory objects have an additional `list` method, which prints out directory entries and their types.

```
>>> cwd.list()
histogram.pickle    : pickle
plot.pickle         : pickle
readme.txt          : text
recodata            : directory
```

Directory methods that produce or operate on the keys in the directory can be given extra arguments. For instance, the `recursive` option will show keys in subdirectories as well.

```
>>> cwd.list(recursive=True)
histogram.pickle            : pickle
plot.pickle                 : pickle
readme.txt                  : text
recodata                    : directory
recodata/reco.root          : Root file
recodata/reco.root/hist1    : 1D histogram
recodata/reco.root/hist2    : 1D histogram
recodata/tracks.table       : table
```

Notice here that `list` now lists the contents of the 'recodata' subdirectory. One of the files in that directory is a Root file named 'reco.root'. Since a Root file has an internal directory structure, PyHEP treats it as a directory itself and the listing descends into that file too.

Retrieving an object is as simple as looking up a key in a dictionary. PyHEP determines the file type from its extension, and loads the object into memory, creating a Python object of the appropriate type to represent it. For instance, a file with the extension '.pickle' is assumed to contain a Python pickle (see the documentation for the standard `pickle` and `cPickle` for details). For example, to retrieve the contents of 'histogram.pickle',

```
>>> histogram = cwd["histogram.pickle"]
>>> histogram
Histogram(EvenlyBinnedAxis(30, (0.0, 30.0), name='energy', units='GeV'), bin_type=float, error_
```

Obtain an object representing a subdirectory, whether an actual file system subdirectory or a virtual subdirectory in a Root or HBOOK file, in the same way.

```
>>> subdir = cwd["recodata"]
>>> print subdir.keys()
['reco.root', 'tracks.table']
>>> rootfile = subdir["reco.root"]
>>> print rootfile.keys()
['hist1', 'hist2']
```

Likewise for retrieving an object from inside a Root file.

```
>>> histogram = rootfile["hist2"]
```

You don't have to hang on to the intermediate directory objects if you only want one object deep in a hierarchy of subdirectories, Root, and HBOOK files.

```
>>> histogram = cwd["recodata"]["reco.root"]["hist2"]
```

Even simpler, you can specify multiple levels with a single indexing operation by separating keys with forward slashes.

```
>>> histogram = cwd["recodata/reco.root/hist2"]
```

Note that you *can not* use '..' to move up in the directory tree, or specify absolute paths, as these operations break the model of nested map objects.

To store an object to disk, simply assign a new item to the directory object as you would with a dictionary.

```
>>> cwd["reco-hist.pickle"] = histogram
>>> print cwd.keys()
['readme.txt', 'plot.pickle', 'reco-hist.pickle', 'recodata', 'histogram.pickle']
```

Of course, the file type inferred from the key's extension must support storing the type of the value object you provide. Python pickles can store most Python objects, including collections such as tuples and dictionaries, and most PyHEP objects.

Generally, the Python object is not associated with the file once it's loaded. If you change the Python object and want the changes reflected in the file, you must store it back. Directories are an exception to this: if you add a new item to the directory, it goes immediately on disk. Tables are also an exception to this.

5.1 File types

When you retrieve a key from a file system directory (*i.e.* a directory object corresponding to an actual directory in the file system, not a directory in a Root or HBOOK file), PyHEP examines the key's extension to determines how to handle the value.

Each type has an associated type name. When retrieving or storing items in a directory, you may override the determination of the file type with the keyword argument `type`, specifying the type name. For example, if you have a Root file named 'histograms.dat' in a directory, you may retrieve it using

```
>>> histograms = directory.get("histograms.dat", type="Root file")
```

The file types understood by PyHEP, and their corresponding extensions, are listed below.

- `"directory"` (no extension): A directory. The value is a directory object.
- `"HBOOK file"` (extension '.hbook'): An HBOOK file. The value is a directory object representing the root of the RZ directory tree inside the file. Additional information about HBOOK files is presented below.
- `"pickle"` (extension '.pickle'): A file containing a Python pickle. The value is whatever Python object was pickled.
- `"Root file"` (extension '.root'): A Root file. The value is a directory object representing the root of the directory tree inside the file. Additional information about Root files is presented below.
- `"symlink"` (no extension): A symbolic link. The value is a string containing the target of the link. Since this type has no associated extension, it can only be used with the `type` keyword argument described above.
- `"table"` (extension '.table'): A PyHEP table. The value is a handle to the open table. In contrast to how other file types are handled, the table is not loaded into memory. Changes to the table are reflected in the table file.
- `"text"` (extension '.txt'): A text file. The value is a character string of the file's contents.
- `"unknown"` (all other extensions): Represents all file types not recognized by PyHEP.

Keys in Root and HBOOK directories are handled differently. The file types corresponding to these keys is determined from metadata stored in the files themselves, not from an extension. Additional file types are support in these directories, including `"1D histogram"` and `"2D histogram"`.

To determine the file type of a key, use the directory's `getinfo`, method described below.

5.2 Methods for accessing keys and values

As we say above, the `keys` method lists all keys in the directory. Directory objects also support the standard `values` and `items` methods, as well as `iterkeys`, `itervalues`, and `iteritems`.

For all of these, you may use these keyword arguments to restrict the keys, values, or items that are returned:

- `recursive`: If true, include recursively the contents of subdirectories.
- `not_dir`: If true, don't include items that are directories (either in the file system or inside Root or HBOOK files).
- `pattern`: Only include items whose keys match the specified regular expression. See the `re` module for a description of Python's regular expression syntax.
- `glob`: Only include items whose keys match the specified glob pattern. See the `glob` module for a description of Python's glob syntax.
- `only_type`: Only include an item if its type is as specified. Item types are specified by strings; see below for more information.
- `known_types`: True by default, which specifies that only items of known types are included. If you set this to false, all items are included. Note that PyHEP will raise an exception if you try to access the value corresponding to a key of unknown type.

These options can also be used with `list`, which prints a listing of keys in a directory and their types.

For example,

```
>>> print cwd.keys(only_type="pickle")
['plot.pickle', 'reco-hist.pickle', 'histogram.pickle']
>>> cwd.list(glob="hist*", recursive=True)
histogram.pickle            : pickle
recodata/reco.root/hist1    : 1D histogram
recodata/reco.root/hist2    : 1D histogram
```

As with a dictionary, the `len` function returns the number of keys in the directory.

```
>>> print len(cwd)
5
```

Iterating over a directory object iterates over its keys.

```
>>> for key in cwd:
...   print key
...
readme.txt
plot.pickle
reco-hist.pickle
recodata
histogram.pickle
```

The `has_key` method and `in` operator return true if the specified key is in the directory.

```
>>> print cwd.has_key("readme.txt")
True
>>> print "missing.pickle" in cwd
False
```

To retrieve an object, use the Python indexing notation (square brackets) or the `get` method. If the key is not found in the dictionary, indexing will raise `KeyError`; the `get` method will return a default value, which you can specify as the second argument (the default is `None`).

```
>>> print cwd["missing.pickle"]
Traceback (most recent call last):
```

```
...
KeyError: ’missing.pickle’
>>> print cwd.get("missing.pickle")
None
>>> print cwd.get("missing.pickle", 42)
42
```

Use the `hep.fs.getdir` function to return a directory object for an arbitrary path. The path may be absolute or relative to the current working directory. The path may descend into Root and HBOOK files. For example,

```
>>> data_dir = hep.fs.getdir("/nfs/data")
>>> histograms = hep.fs.getir("histograms.root/reco")
```

You can also use `hep.fs.get` to load files of other types by specifying the path to the file.

5.3 Accessing file information

A directory object’s `getinfo` method returns an `Info` object containing information about the file. The `Info` object is constructed without loading the file.

All `Info` objects have an attribute `type`, which is the file type for that file. See above for a discussion of file types. An `Info` object may have additional attributes, depending on the type of directory object it was obtained from.

File system `Info` objects also contain attributes `path`, `file_size`, `user_id`, `group_id`, `access_mode`, and `modification_time`.

The code below demonstrates the use of `getinfo` to print a listing of a file system directory.

```
>>> def listFSDir(directory):
...   for key in directory.keys(known_types=False):
...     info = directory.getinfo(key)
...     print ("%-24s (%-12s)  %8d bytes, mode %06o, owner %4d"
...            % (key, info.type, info.file_size, info.access_mode, info.user_id))
...
>>> listFSDir(cwd)
readme.txt               (text        )        50 bytes, mode 000640, owner  500
plot.pickle              (pickle      )      3980 bytes, mode 000640, owner  500
reco-hist.pickle         (pickle      )      1252 bytes, mode 000640, owner  500
recodata                 (directory   )      4096 bytes, mode 000750, owner  500
histogram.pickle         (pickle      )      1257 bytes, mode 000640, owner  500
```

5.4 Storing data

To store data in a file, simply set a key in the dictionary object. For file system directories, the file type is determined from the key’s extension. The file type must support the data type you provide. For example,

```
>>> text = "Hello, world."
>>> cwd["hello.txt"] = text
>>> cwd["hello.pickle"] = text
>>> cwd["hello.root"] = text
Traceback (most recent call last):
...
AttributeError: ’str’ object has no attribute ’keys’
```

Here, the first assignment stores the text in a plain text file, and the second stores it as a pickled Python string. The third assignment fails, since it doesn't make sense to create a Root file with a string.

You can also use the `set` method to set keys. With `set`, you can specify keyword arguments. For example, you can override the file type determination with the `type` argument. For example, this assignment tell PyHEP to store a plain text file, even though the extension '.log' is not known.

```
>>> cwd.set("hello.log", text,type="text")
```

You can other standard map methods to modify the directory's contents as well: `setdefault`, `update`, and `popitem`. To delete files, use the `del` statement or the directory's `delete` method. The `clear` method empties the directory.

These methods can take additional keyword arguments that controls how directories are modified:

- `deldirs` (true by default): Allow automatic recursive deletion of directories and their contents.
- `makedirs` (true by default): Create missing intermediate subdirectories automatically when setting a key (like "`mkdir -p`").
- `replace` (true by default): If true, allow keys to be replaced. Otherwise, raise an exception when setting a key that already exists.
- `replacedirs` (false by default): Like `replace` but applies to subdirectories.

You can also create a directory by setting a key. Since a directory is represented by a map, it looks like this:

```
>>> cwd["subdir"] = {}
```

You may populate the map you assign; the items are stored as files in the new directory.

```
>>> cwd["subdir"] = { "contents.txt": "calibration constants",
...                   "constants.pickle": (10, 11, 12) }
```

To create a directory if it doesn't exist, or obtain it if it does, use `setdefault`:

```
>>> output_dir = cwd.setdefault("output", {})
```

5.5 Working with HBOOK files

You can use directory objects, as described above, to access contents of HBOOK files. An HBOOK file is represented by a directory object, as is a subdirectory in an HBOOK file.

For example, to create a new HBOOK file in the directory represented by `data_dir` containing two histograms,

```
>>> data_dir["histograms.hbook"] = { "hist1": histogram1,
...                                  "hist2": histogram2 }
```

To create a new, empty HBOOK file or return the existing one if it exists,

```
>>> hbook_file = data_dir.setdefault("histograms.hbook", {})
```

5.5.1 The `hep.cernlib.hbook` module

The `hep.cernlib.hbook` module contains PyHEP's implementation of HBOOK file access. HBOOK is a part of the CERNLIB library, and provides a structured file format containing histograms and n-tuples. Note that not all HBOOK features are supported.

Some details particular to HBOOK directory objects:

- PyHEP's module `hep.cernlib.hbook` is linked statically against CERNLIB 2002.
- HBOOK's names are case-insensitive. PyHEP always uses lower case for names in HBOOK files.
- To load a histogram from an HBOOK directory, simply obtain it by name using the subscript operator or `get`. This returns a Python object representing the histogram, which may be modified freely. Note that unlike HBOOK itself, where histograms are always stored in a global "PAWC" memory region, PyHEP constructs ordinary Python objects for histograms. There is no need to manage "PAWC" explicitly.
- Not all histogram features supported in HBOOK are also supported in PyHEP, and visa versa. Therefore, if a histogram is saved to an HBOOK file and later loaded, it may differ in some of its characteristics. The basic histogram binning, and bin contents and errors (including overflow and underflow bins), are stored correctly, however. Note that PyHEP does not provide profile histograms.
- An HBOOK file is not closed until the file object is destroyed, i.e. all references to it are released. Especially when writing an HBOOK file, be careful to release all references to the file object.
- When PyHEP closes an HBOOK file, it "purges cycles", *i.e.* removes old revisions of all entries stored in the file. To disable this behavior, specify `purge_cycles=False` as a keyword argument when creating or accessing the HBOOK file.
- In an HBOOK file, each object is given not only a name identifying it in the directory that contains it, but a numerical RZ identification as well. When you retrieve an object from an HBOOK file, PyHEP stores this value in the object's `rz_id` attribute. When you store an objet from an HBOOK file, you may specify the value to use either by setting the object's `rz_id` attribute or by using an `rz_id` keyword argument; otherwise, PyHEP chooses an available value. An info object obtained from an HBOOK directory's `getinfo` method also has an `rz_id` attribute.

5.5.2 N-tuples

An HBOOK n-tuple is represented in PyHEP by a table. The table satisfies the same protocol as the default table implementation (see `hep.table`), except in the method to create or open tables. Note that because of HBOOK's limitations, certain table features are not supported.

To access an n-tuple in an HBOOK file, use the file object's subscript operator or `get` method, just access the n-tuple's name, just as you would for a histogram. Unlike a histogram, the table object is still connected to the n-tuple in the HBOOK file. A new row appended to the table is incorporated immediately into the n-tuple. Also, the table object carries a reference to the HBOOK file, in its `file` attribute, so the HBOOK file is not closed as long as there is an outstanding table object for an n-tuple in the file.

To create a new n-tuple in an HBOOK file, use the `hep.cernlib.hbook.createTable` function. The arguments are the name of the n-tuple, the HBOOK directory object in which to create the n-tuple, and the schema (as for `hep.table.create`). You may use the optional *rz_id* argument to specify the n-tuple's RZ ID.

By default, a column-wise n-tuple is used for the table; to create a row-wise n-tuple, pass a false value for the optional *column_wise* argument to `createTable`. When creating a column-wise n-tuple, the schema may only contain columns of types `"int32"`, `"int64"`, `"float32"`, and `"float64"`. The schema for a row-wise n-tuple may use only `"float32"` columns.

This program creates an HBOOK file containing a row-wise n-tuple filled with random values. It then re-opens the file, creates a histogram from the values, and stores it in the file.

```
from   hep.cernlib import hbook
import hep.fs
import hep.hist
import hep.table
from   random import random

# Construct a schema with three columns.
schema = hep.table.Schema(a="float32", b="float32", c="float32")
# Create a new HBOOK file.
hbook_file = hep.fs.getdir("test.hbook", makedirs=True, writable=True)
# Create a row-wise n-tuple in it.
table = hbook.createTable("ntuple", hbook_file, schema, column_wise=False)
# Fill 100 random rows into the ntuple.
for i in xrange(0, 100):
    table.append(a=random(), b=random(), c=random())
# Release these to close the HBOOK file.
del table, hbook_file

# Reopen the HBOOK file.
hbook_file = hep.fs.getdir("test.hbook", writable=True)
# Get the n-tuple.
table = hbook_file["ntuple"]
# Project a histogram of the sum of the three values in each row.
histogram = hep.hist.Histogram1D(30, (0.0, 3.0), expression="a + b + c")
hep.hist.project(table.rows, (histogram, ))
# Write the histogram to the HBOOK file.
hbook_file["histogram"] = histogram
```

5.6 Working with Root files

Root set of libraries and programs for high energy physics analysis. Among other things, Root provides a file format for histograms, n-tuples (which are called “trees” in Root), and other data objects. PyHEP provides partial data and file compatibility with Root.

You can use directory objects, as described above, to access contents of Root files. An Root file is represented by a directory object, as is a subdirectory in an Root file.

For example, to create a new Root file in the directory represented by `data_dir` containing two histograms,

```
>>> data_dir["histograms.root"] = { "hist1": histogram1,
...                                 "hist2": histogram2 }
```

To create a new, empty Root file or return the existing one if it exists,

```
>>> root_file = data_dir.setdefault("histograms.root", {})
```

The API and capabilities of `hep.root` are very similar to those of `hep.cernlib.hbook`. A program written for one can be used with the other with minimal modification, and it is easy to write functions, scripts, or programs that can work files from either format.

5.6.1 The `hep.root` module

The `hep.root` contains PyHEP’s implementation of Root file access.

PyHEP's module `hep.root` is built and linked against Root shared libraries which are distributed with PyHEP. The module does not depend on any other version of Root installed on your system.

Some details particular to Root directory objects:

- A Root file is not closed until the file object is destroyed, i.e. all references to it are released. Especially when writing a Root file, be careful to release all references to the file object.
- Not all histogram features supported in Root are also supported in PyHEP, and visa versa. Therefore, if a histogram is saved to a Root file and later loaded, it may differ in some of its characteristics. For instance, any additional attributes added to the histogram will be lost. The basic histogram binning, and bin contents and errors (including overflow and underflow bins), are stored correctly, however.
- In a Root file, each object is given not only a name identifying it in the directory that contains it, but a title. When you retrieve an object from an Root file, PyHEP stores this value in the object's `title` attribute. When you store an object from an Root file, you may specify the title to use either by setting the object's `title` attribute or by using an `title` keyword argument. An info object obtained from an Root directory's `getinfo` method also has an `title` attribute.

5.6.2 Trees

An Root tree is represented in PyHEP by a table. The table satisfies the same protocol as the default table implementation (see `hep.table`), except in the method to create or open tables. Note that because of Root's limitations, certain table features are not supported.

To open a tree in a Root file as a table, use the file object's subscript operator or `get` method, just as you would for a histogram. Note that unlike a histogram returned from `load`, though, the table object that `load` returns is still connected to the tree in the Root file. A new row appended to the table is incorporated in the tree. Also, the table object carries a reference to the Root file, in its `file` attribute, so the Root file is not closed as long as there is an outstanding table object for a tree in the file.

To create a new tree in a Root file, use the `hep.root.createTable` function. The arguments are the name of the new tree, the Root directory in which to create it, and the schema (as for `hep.table.create`). You may specify a title for the table with the *title* argument.

This program creates a Root file containing a row-wise n-tuple filled with random values. It then re-opens the file, creates a histogram from the values, and stores it in the file.

```
import hep.root
import hep.hist
import hep.table
from   random import random

# Construct a schema with three columns.
schema = hep.table.Schema(a="float32", b="float32", c="float32")

# Create a new Root file.
root_file = hep.fs.getdir("test.root", makedirs=True, writable=True)
# Create a tree in it.
table = hep.root.createTable("tree", root_file, schema, title="PyHEP test")
# Fill 100 random rows into the ntuple.
for i in xrange(0, 100):
    table.append(a=random(), b=random(), c=random())
# Release these to close the Root file.
del table, root_file

# Reopen the Root file.
root_file = hep.fs.getdir("test.root", writable=True)
```

```
# Get the table.
table = root_file["tree"]
# Project a histogram of the sum of the three values in each row.
histogram = hep.hist.Histogram1D(30, (0.0, 3.0), expression="a + b + c")
hep.hist.project(table.rows, (histogram, ))
# Write the histogram to the Root file.
root_file["histogram"] = histogram
```

CHAPTER

SIX

Drawing

PyHEP includes a drawing layer that provides device-independent output for simple two-dimensional figures and line drawings. Classes and functions for producing plots use the drawing layer; these are described in the next chapter. The drawing layer is in the module `hep.draw`.

Drawings, plots, and the like are represented in PyHEP by *figures*. A figure is a device-independent memory representation of a drawing in a rectangular region, including all visual attributes—the *style*—of the drawing.

A *renderer* produces output by rendering a figure. Each renderer represents a particular output channel, such as a display window or an output file. The drawing layer supports output to an X11 window, generation of PostScript files, and generation of enhanced Windows metafiles.

6.1 Units and types

The unit of measurement in the PyHEP drawing layer is meters, regardless of the output device; PyHEP attempts to preserve the scale of output. All measurements are given as 'float' values.

The module `hep.draw` includes the conversion factors `point` and `inch` to help you use these units. For instance, the length `12 * point` is about 17 cm, or a sixth of an inch.

Colors are represented by the `Color` class. Its constructor arguments are red, green, and blue color components, each between zero and one. The function `Gray` produces a shade of gray; it produces a color with the same value for all three components. The function `HSV` constructs a color from hue, saturation, and value components. For example,

```
    >>> from hep.draw import *
    >>> dark_red = Color(0.6, 0.1, 0.1)
    >>> medium_gray = Gray(0.6)
    >>> print medium_gray
    Color(0.600000, 0.600000, 0.600000)
    >>> aqua = HSV(0.5, 0.8, 0.5)
    >>> print aqua
    Color(0.100000, 0.500000, 0.500000)
```

Also, `hep.draw` includes the constants `black` and `white`.

The appearance of a line is given by its color, thickness, and dash pattern. Specify the thickness in meters as with any other length. A dash pattern is a tuple of lengths, that specifies the lengths of alternating "on" and "off" segments. For instance, this dash pattern produces a line with alternating long (2 mm) and short (1 mm) dashes, with a fixed small distance (0.5 mm) between the dashes.

```
    >>> dash = (0.002, 0.0005, 0.001, 0.0005)
```

For a solid line, specify `None` as the dash pattern. You can also use these constants for predefined dash patterns: `"solid"`, `"dot"`, `"dash"`, `"dot-dash"`, and `"dot-dot-dash"`.

6.2 Canvases

Not documented yet.

6.3 Renderers

A renderer draws a figure to a display or to an output file. Call the `render` method, passing the figure to render.

PyHEP provides renderers for generating PostScript files, for generating Windows enhanced metafiles, and for displaying in X windows.

6.3.1 PostScript renderers

The module `hep.draw.postscript` provides renderers for PostScript files.

An instance of class `PSFile` generates a multi-page ADSC PostScript file. Pass the constructor the path to the output file.

You may specify the page size using the `page_size` keyword argument. The page size is either a `(width, height)` pair. You can also specify a page size by name, such as `"letter"` or `"A4"`. Named pages sizes are stored in the dictionary `hep.draw.page_size`. If the page size is omitted, letter is assumed.

Each time you invoke the `render` method of a `PSFile` object, you generate a new page in the PostScript. When you are done, simply delete the `PSFile` object to close the file.

This example shows the generatation of a two-page PostScript file.

```
>>> import hep.draw.postscript
>>> ps_file = hep.draw.postscript.PSFile("output.ps", page_size="letter")
>>> ps_file.render(figure1)
>>> ps_file.render(figure2)
>>> del ps_file
```

The class `EPSFile` generates an encapsulated PostScript file. Specify the bounding box size of the image using the `size` keyword argument. You should only invoke the `render` method once per `EPSFile` object.

6.3.2 Enhanced metafile renderer

The module `hep.draw.metafile` provides a render for files in the enhanced Windows metafile (EMF) format. This is a vector file format widely supported by Windows applications. An EMF file essentially contains a recording of the Windows graphics system calls required to draw a figure. The standard file extension for EMF files is “`.emf`”.

The `EnhancedMetafile` class renders figures into EMF files. As with `PSFile`, specify the path to the output file and the image size when creating an `EnhancedMetafile` object. The `render` method should only be invoked once.

This is how you can quickly save a figure as an EMF file.

```
>>> from hep.draw.metafile import EnhancedMetafile
>>> EnhancedMetafile("figure.emf", (0.06, 0.04)).render(figure)
```

6.3.3 X window renderer

The class `hep.draw.xwindow.Window` is a window that can render figures in an X window. Each instance creates a new window.

When you create a new `Window` instance, provide the desired window size *in meters* when creating a new window. You can resize an existing window with its `resize` method, or simply by resizing the window through your window manager. When creating a `Window`, you may also provide a title for the title bar provided by your window manager, or set this later by assigning the `title` attribute.

Each time you call the `render` method, the window's contents are replaced by the specified figure. Note that if the window's contents are destroyed (for example, if you resize the window), you will have to call `render` again to restore the window's contents. Also, if you modify the figure, the window contents do not reflect the changes until you call `render` again.

`Window` uses anti-aliasing and subpixel interpolation when rendering its contents.

6.3.4 Figure windows

A `hep.draw.xwindow.FigureWindow` provides a more convenient way of displaying a figure in a window. The `FigureWindow` knows the figure it is rendering, and redraws its contents when necessary (when the window is exposed or resized).

Using a `FigureWindow`, it's easy to display a figure. The second argument is the window size.

```
>>> from hep.draw.xwindow import FigureWindow
>>> window = FigureWindow(figure, (0.16, 0.10))
```

You can change the figure displayed in the window by setting the `figure` attribute. If the value is `None`, nothing is drawn in the window. If the figure changes, call the `redraw` method to force the window to redraw it.

6.4 Layouts

A *layout* is a composite figure that arranges several other figures. A layout object is itself a figure, so it can be rendered directly by renderers, and included in other layouts. The layout objects described here are in the `hep.draw` module.

A simple layout class is `SplitLayout`. It arranges two figures next to each other, either horizontally or vertically. A `SplitLayout` allows you to specify the fractions of the entire drawing region in which to draw each of the two figures. When you create a split layout, specify the orientation of the split, either `"vertical"` or `"horizontal"`, and the two figures. Either figure may be `None`. You may also specify the fraction of the first figure (the default is 0.5); the remainder is used to draw the second figure.

For example, this code creates a layout displaying `fig1` on the left, occupying three-quarters of the layout, and `fig2` on the right, occypting the remaining quarter.

```
>>> from hep.draw import *
>>> layout = SplitLayout("vertical", fig1, fig2, fraction=0.75)
```

You can create more flexible layouts with a `BrickLayout` object. A *brick layout* resembles a row of bricks: the region is divided into equally-spaced rows, each of which is divided into equally-spaced cells. Each row may have a different number of cells. To specify the arrangement of cells, provide a sequence containing the number of cells in each row. The number of elements in the sequence is the number of rows. For instance, to create a layout with three rows, of which the first and third are divided in half and the middle is divided into thirds,

```
>>> layout = BrickLayout((2, 3, 2))
```

You may also specify style attributes as keyword arguments; as with other figures, a layout's style attributes are stored in a dictionary attribute `style`. For example, the `margin` style attribute controls the size of an empty margin inserted between rows and cells in a row.

```
>>> layout = BrickLayout((2, 3, 2), margin=12*point)
```

You can index the figures in a brick layout by column and row index, for instance

```
>>> layout[0, 0] = fig1
>>> layout[1, 2] = fig2
```

Any entry that contains `None` (the default) is left empty.

A `GridLayout` is a subclass of `BrickLayout` in which all rows have the same number of cells. This produces a grid of evenly-sized cells. Specify the number of columns and rows when creating an instance.

```
>>> layout = GridLayout(2, 3)
```

CHAPTER

SEVEN

Plotting

PyHEP includes classes and function for producing plots of histograms, functions, and scatter plots. The `hep.hist.plot.Plot` class is the basic class used for producing plots. An instance of `Plot` is a figure object and can be used with renderers and layouts, as described in the previous chapter.

Here's a basic example. First, create a histogram:

```
>>> import hep.hist
>>> hist = hep.hist.makeSample1D("flat", 500)
```

Now create a `Plot` object, which is a figure depicting the histogram.

```
>>> from hep.hist.plot import Plot
>>> plot = Plot(1, hist)
```

When creating the plot, you specified the number of dimensions (independent variables) that are depicted in the plot, and the histogram we want to plot. Finally, create a window, and render the plot in the window.

```
>>> from hep.draw.xwindow import Window
>>> window = Window((0.2, 0.1))
>>> window.render(plot)
```

The argument to `Window` is the width and height of the window in meters.

By default, a bin of a one-dimensional histograms is drawn as crosses. The horizontal line shows the bin value and bin width, and the vertical line shows the range of errors on the bin content. On the left and right sides are displayed the underflow and overflow bins, labelled "UF" and "OF" respectively.

You can just as easily generate a PostScript file containing the plot. Just use a PostScript renderer instead of the `Window` object.

```
>>> from hep.draw.postscript import PSFile
>>> ps_file = PSFile("plot.ps")
>>> ps_file.render(plot)
>>> del ps_file
```

By deleting the `PSFile` object, you generate the output and close the file.

7.1 Plots and series

A `hep.hist.plot.Plot` object is a figure that displays a plot of histograms, functions, and similar data. A plot can display either data with one independent variable, such as one-dimensional histograms and functions of one variable, or data with two independent variables, such as two-dimensional histograms and scatter plots. More than one histogram, function, or scatter plot can be shown at the same time; each is called a *series*. All series in the same plot share the same vertical and horizontal axes (and z range, for two-dimensional plots).

To create a plot object, specify the number of dimensions in the plot, either 1 or 2. You may specify the series to plot as additional arguments. For example, to create a plot overlaying a histogram and a function,

```
>>> from hep.hist.plot import Plot
>>> plot = Plot(1, histogram, function)
```

You can also use the `append` method to add a series to a plot, so the following code is equivalent:

```
>>> plot = Plot(1)
>>> plot.append(histogram)
>>> plot.append(function)
```

A `Plot` object has a `series` attribute, which is a list of series in the plot. The series are not the histograms and functions themselves, but figure objects representing the series. (Generally, you wouldn't to render these figures directly.)

```
>>> plot.series[0]
<hep.hist.plot.Histogram1DPlot object at 0xf6d01fcc>
```

The plot object for a histogram references the histogram itself in its `histogram` attribute.

```
>>> plot.series[0].histogram
Histogram(EvenlyBinnedAxis(20, (0.0, 1.0)), bin_type=int, error_model='poisson')
```

You can remove series from a plot simply by removing the corresponding element from the `series` list. Similarly, you can reorder elements in the list to change the stacking order of series in the plot.

7.2 Plot styles

Styles control the visual attributes of plots. Style information is not stored in a histogram, which only contains statistical data and annotations (such as units). Instead, style is stored in a plot.

Each plot has a `style` attribute, which is a dictionary containing style items. Keys in the style dictionary are names of style attributes. Any style attributes may be stored in the style dictionary; different plot types use different style attributes to control their output.

A `Plot` object has a `style` attribute; in addition, the plot object representing each series in a plot has a `style` attribute as well. This allows different styles to be used for each series in a plot (for instance, to assign a different color to each series). If a particular style attribute is missing from a series's style dictionary, it uses the value from the parent `Plot` object's style dictionary. Thus, if you set a style attribute in a `Plot`, it will apply to each series in that plot, except for a series which overrides it in its own style dictionary.

Suppose you have a plot showing two histograms.

```
>>> plot = Plot(1, histogram1, histogram2)
```

To draw 2 mm dots at bin contents for both series in the plot, set the `"marker"` and `"marker size"` attributes in the plot's style dictionary.

```
>>> plot.style["marker"] = "filled dot"
>>> plot.style["marker_size"] = 0.002
```

To draw the second series (but not the first) in red, set the `"color"` attribute in that series's style dictionary.

```
>>> plot.series[1].style["color"] = hep.draw.Color(0.7, 0, 0)
```

You can set style attributes for a `Plot` when you create it, by adding keyword arguments. Similarly, you can set style attributes for a series by adding keyword arguments to `append`. For example, this code produces the same plot as the code above does:

```
>>> plot = Plot(1, marker="filled dot", marker_size=0.002)
>>> plot.append(histogram1)
>>> plot.append(histogram2, color=Color(0.7, 0, 0))
```

The sections below list style attributes used by PyHEP's plot classes to determine the visual style of plots. For an explanation of how to specify fonts, colors, marker styles, *etc.*, see the chapter on drawing.

7.2.1 Styles for plot objects

These style attributes control the visual style of `Plot` objects.

- `"bottom_margin"`: The size of the margin on the bottom of the plot.
- `"color"`: Specifies the color for the entire plot, including axes and labels.
- `"font"`: Specifies the font to use for titles and labels on the plot.
- `"font_size"`: Specifies the font size to use for titles and labels on the plot.
- `"left_margin"`: The size of the margin on the left side of the plot.
- `"log_scale"`: If true, a logarithmic scale is used for the dependent axis (the y axis in one-dimensional plots, the z range in two-dimensional plots).
- `"normalize_bin_size"`: How to normalize bin contents to a common bin size. The contents of bins of different sizes are normalized to a common effective bin width (for one-dimensional histograms) or bin area (for two-dimensional histograms) to facilitate the interpretation of bin contents as an approximation of probability density. The contents of overflow bins are never normalized.

  If the value of this style attribute is a number, the content of each bin is normalized to this bin size. If the value is `"auto"` (the default), a bin size is automatically chosen (for a histogram with evenly-binned axes, the histogram's actual bin size is used). If the value is `None`, bins are not normalized, and the actual values of bin contents are plotted.
- `"overflows"`: If true, show underflow and overflow bins for the x axis (for one-dimensional plots) or the x and y axis (for two-dimensional plots). For two-dimensional plots, underflow and overflow bins can be enabled for the x and y axes individually with the `"x_axis_overflows"` and `"y_axis_overflows"` style attributes.
- `"overflow_line"`, `"overflow_line_color"`, `"overflow_line_dash"`, `"overflow_line_thickness"`: Whether to draw lines separating the overflow bins from the rest of the bin data, and the color, dash pattern, and thickness of the line.
- `"right_margin"`: The size of the margin on the right side of the plot.
- `"title"`: A title to draw on the plot.
- `"top_margin"`: The size of the margin on the top of the plot.
- `"x_axis_font"`, `"x_axis_font_size"`: The font and font size to use for labelling the x axis and its ticks. Likewise for the y axis.

- `"x_axis_line"`, `"x_axis_color"`, `"x_axis_thickness"`: Whether to draw the x axis, and its color and thickness. Likewise for the y axis.
- `"x_axis_offset"`: The distance between the x axis and the edge of the bin data. Likewise for the y axis.
- `"x_axis_position"`, `"y_axis_position"`: The position of the axes on the plot. For the x axis, the value may be either `"bottom"` or `"top"`; for the y axis, either `"left"` or `"right"`.
- `"x_axis_range"`: The range of values to display along the x axis, as a `(lo, hi)` pair. Likewise for the y axis.
- `"x_axis_ticks"`: Specifies the tick marks to draw along the x axis. The value may be the approximate number of tick marks to use (chosen heuristally from the axis range); or a sequence of tick positions; or `None`. Likewise for the y axis.
- `"x_axis_tick_size"`, `"x_axis_tick_thickness"`: The length and thickness of tick marks on the x axis. Likewise for the y axis.
- `"z_range"`: For two-dimensional plots, the range of values to display on the (virtual) z axis, as a `(lo, hi)` pair.
- `"zero_line"`, `"zero_line_color"`, `"zero_line_thickness"`: In one-dimensional plots, whether to draw the zero line, and the zero line's color and thickness.

In addition, the style attributes that control the border, aspect ratio, and size of layouts can be specified for plot objects as well. See the section on layout styles in the chapter on drawing.

7.2.2 Styles for all plot series

These style attribute control the visual styles of individual series in plots. If a series's style dictionary doesn't contain a particular attribute, the value from the parent `Plot` object's style dictionary is used. Certain style attributes are used only for some bin styles.

- `"color"`: The color in which to draw this series.
- `"dash"`: The dash pattern to use for lines.
- `"errors"`: If true, display bin errors for histograms that contain error information. The representation of bin errors depends on the bin style.
- `"marker"`: The marker to use when drawing points at bin contents. If `None`, no markers are drawn.
- `"marker_size"`: The marker size to use when drawing points at bin contents.
- `"thickness"`: The line thickness to use.
- `"suppress_zero_bins"`: If true, a bins with zero contents will not be drawn.

7.2.3 Styles for 1D histograms plot series

These style attributes are specific to plot series of one-dimensional histograms.

- `"bins"`: The style to use to draw bins. These bin styles may be used:
  - `"points"` (the default): Draws a marker and/or cross at the value of each bin.
  - `"skyline"`: Draws the traditional outline or filled region representing bin contents.

- `"bin_center"`: In the `"points"` bin style, the fractional horizontal location within the bin to draw the marker and the vertical error bar. The value should be between zero and one. The default is 0.5, which centers the marker and error bar in the bin. When superimposing two series with identical binning and similar bin contents, it is useful to offset the error bars of one or both to prevent them from overlapping.
- `"cross"`: In the `"points"` bin style, whether to draw a horizontal line in each bin. The position of the line shows the bin contents, and the length of the line shows the bin width.
- `"error_hatch_color"`, `"error_hatch_pitch"`, `"error_hatch_thickness"`: In the `"skyline"` bin style, bin errors are depicted with a 45-degree hatch pattern. These style attributes control the color, spacing, and thickness of the hacth pattern. If the color is `None`, the hatch pattern uses the fill pattern for errors above the bin contents, and the background color below the bin contents.
- `"fill_color"`: In the `"skyline"` bin style, the color with which to fill the bins. If `None`, the bins are not filled.
- `"line_color"`: In the `"skyline"` bin style, the color for the outline of the bin contetns. If `None`, no outline is drawn.

7.2.4 Styles for 2D histograms plot series

These style attributes are specific to plot series of two-dimensional histograms.

- `"bins"`: The style to use to draw bins. These bin styles may be used:
  - `"box"`: Draws a box for each bin with area proportional to the value of the bin content.
  - `"density"`: Shades each bin with a color reprenting the bin content.
- `"negative_color"`: If a color is specified, an empty bin represents zero bin contents, and this color is used to draw the contents of negative bins. If this style attribute is `None`, an empty bin represents the low end of the z range, and all values are represented relative to this value.
- `"overrun_color"`, `"underrun_color"`: Colors to use to draw bins whose values are outside the z range of the plot.

7.2.5 Styles for function plot series

These style attributes are specific to plot series of one-dimensional functions.

- `"bins"`: The style to use to draw bins. These bin styles may be used:
  - `"curve"`: Draw the function value as a curve.
- `"number_of_samples"`: The number of points at which to sample the function.

7.3 Annotations and decorations

Using the class `hep.hist.plot.Annotation`, you can add textual annotations to a plot. The annotation contains one or more lines of text. You may specify the text when constructing the object, or incrementally using the `append` method or left-shift operator. If the text contains newline characters, it is split into multiple lines.

The annotations are drawn left-justified in the upper-left corner of the plot, or right-justified in the upper-right corner of the plot. Specify the position with the `"position"` style, which may be either `"left"` or "right". The styles "annotation_font", "annotation_font_size", "annotation_color", and "annotation_leading" control how the text is drawn.

Simply add the Annotation object as a series to the plot.

For example, this code makes a plot of a histogram with an annotation describing its bin and axis type

```
>>> plot = hep.hist.plot.Plot(1, histogram)
>>> annotation = hep.hist.plot.Annotation("title: %s" % histogram.title)
>>> annotation << "bin type: %s" % histogram.bin_type.__name__
>>> annotation << "axis type: %s" % histogram.axis.type.__name__
>>> plot.append(annotation)
```

The class hep.hist.plot.Statistics is a subclass of Annotation which annotates histogram statistics. To create one, provide a sequence of statistic names to include, and one or more histograms. Statistic names can be "sum", "mean", "variance", "sd", and "overflows". Then simply add the statistics annotation object as a series to the plot.

For example, to plot a histogram and display its sum (integral), mean, and standard deviation,

```
>>> plot = hep.hist.plot.Plot(1, histogram)
>>> plot.append(hep.hist.plot.Statistics(("sum", "mean", "sd"), histogram))
```

If you want to include the statistics with other annotations, you can generate the text of the statistics annotation with the function hep.hist.plot.formatStatistics.

7.4 Prepackaged plot functions

CHAPTER

EIGHT

Lorentz geometry and kinematics

PyHEP provides an implementation of Lorentz vectors, momenta, and transformations. The implementation classes do not simply provide quadruplets of numbers equipped with a Minkowski inner product; instead, they represent coordinate-independent objects. Ultimately, though, the coordinates of a vector must be specified and obtained; this is done using a *reference frame*, which specifies the coordinate system to use.

Coordinates are given in the order (t, x, y, z), the metric signature $(+, -, -, -)$ is employed, and the speed of light c is assumed to be unity.

8.1 Reference frames

A four-vector is a geometric object, which can be used to represent, for instance, the space-time position of an event, or the energy-momentum of a particle. The typical representation of a four-vector is a quadruplet of four coordinates, but the coordinate values for a particular four-vector depends on the basis used, or equivalently on the reference frame in which the coordinates are specified.

The Python class `hep.lorentz.Frame` represents a reference frame. The principle of relativity implies that there is no absolute way of specifying a frame; the frame may only be specified in relation to another. In PyHEP, a frame is specified in relation to a special frame, the canonical *lab frame*. The lab frame is `hep.lorentz.lab`, an instance of `Frame`.

8.2 Vectors

A four-vector is represented by an instance of `hep.lorentz.Vector` class. A `Vector` instance represents the geometric object, which is independent of reference frame, so you can specify or obtain its coordinate values only in reference to a frame. You may use the lab frame for this, or any other frame which you create.

To create a vector by specifying its coordinates in a particular frame, use the `Vector` method of that frame. For instance,

```
from hep.lorentz import lab
vector = lab.Vector(5.0, 1.0, 0.0, -2.0)
```

creates a four-vector whose time coordinate is 5.0 and whose space coordinates are (1.0, 0.0, -2.0) in the lab frame.

To obtain the coordinates of a four-vector in a reference frame, use the `coordinatesOf` method of that frame. For instance,

```
t, x, y, z = lab.coordinatesOf(vector)
```

You may negate (invert) a vector or scale it by a constant. You may also add or subtract two vectors. Since these are geometric operations, no frame is specified. For example,

```
vector3 = - vector1 / 2 + 3 * vector2
```

Use the ˆ operator to obtain the inner product of two vectors. Each vector also has an attribute `norm`, its Lorentz-invariant normal. For example,

```
c = (vector1 ˆ vector2) / (vector1.norm * vector2.norm)
```

You may also use `hep.lorentz.Momentum`, a subclass of `Vector` that represents a four-momentum. It provided an additional attribute `mass`, which is equivalent to `norm`, plus an attribute `rest_frame`, which is a `Frame` object representing the rest frame of a particle with that four-momentum.

8.3 Transformations and frames

A `hep.lorentz.Transformation` object represents a general Lorentz transformation. It can be used to transform either a geometric object, such as a four-vector, or a reference frame.

Typically, a transformation is specified as a rotation or a boost. A rotation is specified by the Euler angles ϕ, θ, ψ in a particular reference frame. A boost is specified by the vector $\vec{\beta}$ in a particular reference frame. The frame object's `Rotation` and `Boost` methods, respectively, create these transformations. For example,

```
from hep.lorentz import lab
from math import pi
rotation = lab.Rotation(pi / 4, pi / 4, 0)
boost = lab.Boost(0.0, 0.0, 0.5)
```

The arguments to `Rotation` are the Euler angles, and the arguments to `Boost` are the components of $\vec{\beta}$.

Transformations may be composed using the operation. Be careful about the frame in which you specify each one; generally, for sequential transformations, you will want to apply the previous transformation to your starting frame of reference before specifying the next one.

You can apply a transformation to a four-vector using the ˆ operator; this returns a different geometric four-vector.

FIXME

A transformation can also be used to create a new reference frame.

CHAPTER

NINE

Particle properties

The `hep.pdt` module provides code to access a *particle data table*, which contains measured properties of particles studied in high energy physics.

PyHEP includes a particle data table in `hep.pdt.default`. This table contains particle data from the XXXX edition of the *Review of Particle Properties* published by the Particle Data Group (PDG). The function `hep.pdt.loadPdtFile` can be used to load particle data from a file in the PDG's text file format.

A particle data table is represented by a `hep.pdt.Table` instance, which acts as a dictionary keyed with the plain-text names of particles. The text (ASCII) names are set by the PDG. These names are generally the same as the conventional particle designations, with Greek letters spelled out and underscores used to denote subscripts. For particle names associated with more than one charge state, the charge muse be indicated. Neutral antiparticles conventionally designated with an overbar are denoted by the "`anti-`" prefix. For example, a positron is spelled "`e+`", and the short-lived kaon eigenstate "`K_S0`". The usual `keys` method will return the plain-text names of all particles in the table. A particle's name may have alternate common text spellings; these are stored as aliases, and a particle may be accessed in the table by any of its aliases.

The values in a particle data table are `hep.pdt.Particle` instances. Each represents a single species of particle, with unique quantum numbers. (Some special particle codes representing bound states, particles that have not been established, Monte Carlo internal constructs, etc. are also included.)

A `Particle` object has these attributes. Central values are given for measured properties.

- `name` is the particle's text name.
- `aliases` is a sequence of alternate text names for the particle.
- `charge_conjugate` is the `Particle` object for the particle's charge conjugate.
- `mass` is the particle's nominal mass, in GeV.
- `width` is the particle's width, in GeV.
- `charge` is the particle's electric charge.
- `spin` is the particle's spin, in units of half h-bar.
- `is_stable` is true if the particle is considered stable.
- `id` is the particle's Monte Carlo ID number in the PDG's numbering scheme.

The `Table` object also as a method `findId`, which returns the particle corresponding to a Monte Carlo ID number.

The following is a demonstration of using the default particle data table to look up some particle properties.

```
>>> from hep.pdt import default as particle_data
>>> print particle_data["e-"].mass
0.000510999
>>> print particle_data["J/psi"].spin
1.0
>>> print particle_data.findId(22).name
gamma
```

CHAPTER

TEN

Using EvtGen

PyHEP provides a simple interface to the EvtGen event generator. You can easily generate randomized decays of particles, and examine the decay products. The EvtGen interface is in the `hep.evtgen` module.

To create a particle decay, follow these steps:

1. Create a `Generator` instance. The two arguments to its constructor are the path to the particle data listing file, which contains particle property information, and the path to the main decay file, which contains decays and branching fractions. The default EvtGen particle data and decay files are used if these arguments are omitted. You may specify paths to user decay files, which override the main decay file, as additional arguments. See the EvtGen documentation for information about these files.
2. Create a `Particle` object to represent the initial-state particle. Specify the name of the particle, as listed in the particle data file, as the argument. The particle is originally at rest at the origin of the lab frame.
3. Produce the decay by calling the generator's `decay` method on the particle object.

A `Particle` object's momentum is stored in its `momentum` attribute, as a `hep.lorentz.FourMomentum` object. Use `hep.lorentz.lab.coordinatesOf` to obtain its lab-frame coordinates. Similarly, its production position is stored in its `position` attribute, as a `hep.lorentz.FourVector`. The name of the Particle's species is in its `species` attribute.

Use the `decay_products` attribute to access the decay products of a decayed particle. That value is a sequence of `Particle` objects representing the particle's decay products.

The following script produces a single decay of an Upsilon(4S), using a particle data listing file and a decay file in the current directory. It prints out the decay tree of the Upsilon(4S), with the lab components of each product's momentum, using the `printParticleTree` function.

```
import hep.evtgen
from   hep.lorentz import lab

def printParticleTree(particle, depth=0):
    indentation = depth * " "
    name = "%-12s" % particle.species
    pad = (8 - depth) * " "
    momentum = "%5.2f %5.2f %5.2f %5.2f" % lab.coordinatesOf(particle.momentum)
    print indentation + name + pad + momentum
    for child in particle.decay_products:
        printParticleTree(child, depth + 1)


generator = hep.evtgen.Generator("evt.pdl", "DECAY.DEC")
upsilon4s = hep.evtgen.Particle("Upsilon(4S)")
generator.decay(upsilon4s)
printParticleTree(upsilon4s)
```

CHAPTER

ELEVEN

Interactive PyHEP

This chapter describes how to use interactive PyHEP. Interactive PyHEP is simply the ordinary Python interpreter with certain PyHEP modules preloaded, commonly-used names imported into the global namespace, and some additional interactive functions and variables are provided.

11.1 Invoking interactive PyHEP

The `pyhep` script launches interactive PyHEP. This is installed by default in `/usr/bin`; check your installation for your specific location.

In addition to the Python interpreter's usual startup message, the PyHEP version is displayed.

11.2 Imported names

These names are imported into the global namespace:

- The contents of the standard `math` module.
- The contents of the `hep.num` module.
- If your version of Python doesn't include built-in `bool`, `True`, and `False`, these names are added.
- The `hep.lorentz.lab` reference frame object, as `lab`.
- The `default` particle data dictionary from `hep.pdt`, as `pdt`.

11.3 Interactive functions and variables

The names of interactive PyHEP functions always begin with '`i`', and the names of variables always begin with '`i_`'.

The `ihelp` function displays some help information for interactive PyHEP.

11.3.1 Plotting functions

Interactive PyHEP plots histograms in a pop-up plot window, similar to PAW. The plot window may be divided into rectangular "zones", each displaying one plot. All plots are shown in the same window, replacing other plots where necessary.

iplot(*histogram*[, ***style*])
: Show a new plots of a histogram or scatter plot in the plot window. The argument is a histogram or `Scatter` object. Any keyword arguments are used as style attributes for the new plot.

iseries(*histogram*[, ***style*])
: Add a series to the current plot in the plot window. The argument is a histogram or `Scatter` object. Any keyword arguments are used as style attributes for the new plot.

igrid(*columns, rows*)
: Divide the plot window into a grid of rectangular plot zones. The arguments are the number of columns and rows in the grid. Each call to `iplot` uses the next plot zone, proceeding left-to-right and then top-to-bottom. Note that calling `igrid` always removes all plots and clears the plot window.

iselect(*column, row*)
: Select a plot zone. Its arguments are the column and row coordinates of the zone. The next plot is drawn in this zone.

ishow(*figure*)
: Show `figure` in the plot window. If you have divided the figure with `igrid`, shows `figure` in the current cell.

iprint(*file_name*)
: Print the contents of the plot window to a file. The type of the file is inferred from the file name extension: ".ps" produces a PostScript file, and ".eps" produces an encapsulated PostScript file.

The global variable `ifig` always refers to the current `Plot` object, the current figure (if you added one with `ishow`, or `None` (if the current plot zone is empty). The global variable `i_plots` is a sequence of sequences of `Plot` objects for all zones in the plot window.

The global variable `iwin` is a draw object for the plot window.

The global variable `istyle` is a style dictionary that is used for plots and series created with `iplot` and `iseries`. If you set a style attribute in `istyle`, it will become the default for subsequent plots. You can also call the `setall` method of `istyle`, which sets a style attribute not only in this dictionary, but in all of the plots currently shown in the plot window. For example, to set a log scale for the plots currently shown as well as subsequent plots,

```
>>> istyle.setall("log_scale", True)
```

To set a style attribute in the current plot only, access its `style` member through `ifig`, like this:

```
>>> ifig.style["log_scale"] = True
```

PyHEP keeps track of when a plot's style dictionary is changed, and redraws the plot automatically.

You may modify any plot displayed in any plot zone. Select the zone with `iselect`, access and modify the plot with `i_plot` (for instance, adjust its style or add a series), and then call `iredraw` to redraw it.

11.3.2 Table functions

iproject(*table, expression*[, *selection, number_of_bins, range*])
: Project a histogram from a table. The first argument is the table object, and the second argument is the expression to accumulate in the histogram. The optional `selection` argument is a selection expression; only table rows for which this expression is true are projected into the histogram. The number of bins and histogram range are automatically determined, but to override these, use the `number_of_bins` and `range` (a pair of values) arguments, respectively. The return value is a histogram object.

: To project a one-dimensional histogram, use an ordinary expression. To project a two-dimensional histogram, specify two expressions in `expression` separated by commas; for instance `"p_x, p_y"`. You can also use the `iproj1` function to project a one-dimensional histogram from a table.

idump(*rows, *expressions*)
: Dumps values from a table. The first argument is a table or an iterator over table rows. One or more additional

arguments are expressions whose values are to be displayed. The output is a table with the row number followed by the values of the specified expressions.