fast-carpenter Documentation

Release 0.17.4

['Ben Krikler']

Mar 12, 2020

Contents:

FAST carpenter

Turns your trees into tables (ie. reads ROOT TTrees, writes summary Pandas DataFrames)

fast-carpenter can:

- Be controlled using YAML-based config files
- Define new variables
- Cut out events or define phase-space "regions"
- Produce histograms stored as CSV files using multiple weighting schemes
- Make use of user-defined stages to manipulate the data

Powered by:

- AlphaTwirl (presently): to run the dataset splitting
- Atuproot: to adapt AlphaTwirl to use uproot
- uproot: to load ROOT Trees into memory as numpy arrays
- fast-flow: to manage the processing config files
- fast-curator: to orchestrate the lists of datasets to be processed
- Espresso: to keep the developer(s) writing code

A tool from the Faster Analysis Software Taskforce: http://fast-hep.web.cern.ch/

CHAPTER 1

From Pypi

The simplest way to install things is from pypi.

```
pip install fast-carpenter
```

Note: In general it's better to install this in a specific environment (e.g. using virtualenv or better still conda).

Otherwise you might need to use the `--prefix` or `--user` options for `pip install`. If you do provide any of these options, make sure you have the `/bin` directory in your `PATH` (e.g. if you used `--user` this will be mean you need `~/.local/bin` included in the PATH). Using virtaulenv or conda will avoid this complication

CHAPTER 2

From Source

If you want to edit the code or contribute something back, you might want to install things directly from github. Any of these options should work:

Directly with pip:

```
pip install -e 'git+https://github.com/FAST-HEP/fast-carpenter.git#egg=fast_carpenter
↪' --src .
```

Clone first and install:

```
git clone https://github.com/FAST-HEP/fast-carpenter.git
cd fast-carpenter
python setup.py develop
```

CHAPTER 3

Key Concepts

3.1 Goals of fast-carpenter

3.1.1 From the user's perspective

fast-carpenter's principal goal is to help a user ask **"what do I want to see"** as opposed to **"how do I implement this"** which has been the more traditional way of thinking for a particle physics analyst.

In that sense, most of the control of this code is "declarative" in the sense that a user should typically not have to say how to move data through the analysis, only what they want it to do. That way fast-carpenter can make decisions behind the scenes as to how to handle this. Python dictionaries are therefore the main way to configure carpenter, which we describe using YAML.

The net result of this should mean:

- What the user writes is closer to the actual mathematical description of what they want to do.
- There is less actual analysis "code" and so less opportunity to put bugs in the analysis.
- It should be quick to do a simple study, and then scale smoothly to a full-blown analysis.
- Python dictionaries can be built easily in other tools, and so fast-carpenter directly called from inside a Jupyter notebook, for example.
- When you want to do something more exotic, which is not (yet) catered for in fast-carpenter itself, there is an easy "plugin" style system to add your own custom code into the processing.

Although fast-carpenter is focussed on input ROOT trees at this point (which inspires its name), this may well evolve in the future.

3.1.2 From the code and development perspective

- We have tried to make the code as modular as possible (hence fast-flow and fast-curator not being contained in this package).
- Wherever possible, we've tried to avoid writing code; if an existing package does that task, use it.

- If another tool provides most but not all of a wanted functionality, we prefer contributing to that package over putting code in here.
- Keep functions simple (e.g. under the mccabe metrics).
- Unit-tests should be clear, and are as important as documentation; ideally the two can serve a similar purpose.

3.2 Overall approach for data-processing

fast-carpenter is intended to be the first step in the main analysis pipeline. It is expected to be the only part of the processing chain which sees "event-level" data, and produces the necessary summary of this in a tabular form (which invariably means binned as histograms). Subsequent steps can then manipulate these to produce final analysis results, such as graphical figures, or doing some functional fit to the binned data.

For public examples of working with fast-carpenter and the other FAST tools, see *Example repositories*.

3.2.1 Step 1: Create dataset configs

fast-carpenter needs to know what input files to use, and will often need extra metadata (for example, does this data represent real or simulated data). This is where the fast-curator package comes in. It provides the *fast_curator* command which we use to generate descriptions of the input files in a format that is both human and machine readable, using YAML. These can then be put in a repository and updated periodically with extra meta-data, or when new data becomes available.

See the *fast_curator* section of the command lines tools for in-depth discussion of the fast_curator command which can automate the process of making these configs.

3.2.2 Step 2: Write a processing config

The next step is to prepare the "processing configuration" which defines what you want to do with this data. This has to be written by hand, and is the core of what you want to spend time on as an analyzer. Behind the scenes, this config file is interpreted using the small fast-flow package; documentation there might also be of interest.

For details about how to write this config file, see *The Processing Config*.

3.2.3 Step 3: Run fast_carpenter

You can now run the fast_carpenter command, giving it the dataset and processing configurations from the previous steps. Depending on how many files you have and what type of computing resources (e.g. multiple cpus, batch systems) you might want to use some of the different processing modes.

For more on how to use the `fast_carpenter` command, see the *fast_carpenter* section of the command line tools.

3.2.4 Step 4: Produce plots

In order to visualise the results of running `fast_carpenter`, you can use the `fast-plotter` package, which gives you both a command-line interface and helper python functions to produce plots from the dataframes. Fine tuning of the command-line plots is again possible using a YAML configuration file.

See the dedicated fast-plotter documentation for more guidance on this package.

CHAPTER 4

Command-line Usage

The command-line tools are the primary way to use fast-carpenter and friends at this point. All of the FAST commands provide built-in help by providing the `--help` option.

4.1 `fast_curator`

The fast-curator package handles the description of the input datasets. These are saved as YAML files, which contain a dictionary that lists the different datasets, the list of files for each dataset, and additional meta-data.

You can build these files semi-automatically by using the `fast_curator` command. This can be called once per dataset and given a wildcarded expression for the input files of this dataset, which it will then expand, build some simple summary meta-data (number of events, number of files, etc) and write this to an output YAML file. If the output file already exists, and is a valid fast-curator description, the new information will be appended to the existing file.

Input ROOT files can also be stored on xrootd servers, with a file-path specified by the `root://` protocol. You can also provide wild-cards for such files, but make sure to check that you pick all files that you expect; wildcarded searches on xrootd directories can depend on permissions, access rights, storage mirroring and so on.

For an in-depth description of the dataset description files, see the fast-curator pages.

```
$ fast_curator --help
usage: fast_curator [-h] -d DATASET [-o OUTPUT] [--mc] [--data] [-t TREE_NAME]
                    [-u USER] [-q QUERY_TYPE] [--no-empty-files]
                    [--allow-missing-tree]
                    [--ignore-inaccessible IGNORE_INACCESSIBLE] [-p PREFIX]
                    [--no-defaults-in-output] [--version] [-m META]
                    [files [files ...]]

positional arguments:
  files

optional arguments:
```

(continues on next page)

(continued from previous page)

```
  -h, --help            show this help message and exit
  -d DATASET, --dataset DATASET
                        Which dataset to associate these files to
  -o OUTPUT, --output OUTPUT
                        Name of output file list
  --mc                  Specify if this dataset contains simulated data
  --data                Specify if this dataset contains real data
  -t TREE_NAME, --tree-name TREE_NAME
                        Provide the name of the tree in the input files to
                        calculate number of events, etc
  -u USER, --user USER  Add a user function to extend the dataset dictionary,
                        eg. my_package.my_module.some_function
  -q QUERY_TYPE, --query-type QUERY_TYPE
                        How to interpret file arguments to this command.
                        Allows the use of experiment-specific file catalogues
                        or wild-carded file paths. Known query types are:
                        xrootd, local
  --no-empty-files      Don't include files that contain no events
  --allow-missing-tree  Allow files that don't contain the named tree in
  --ignore-inaccessible IGNORE_INACCESSIBLE
                        Don't include files that can't be opened
  -p PREFIX, --prefix PREFIX
                        Provide a common prefix to files, useful for
                        supporting multiple sites
  --no-defaults-in-output
                        Explicitly list all settings for each dataset in
                        output file instead of grouping them in default block
  --version             show program's version number and exit
  -m META, --meta META  Add other metadata (eg cross-section, run era) for
                        this dataset. Must take the form of 'key=value'
```

4.2 fast_curator_check

Sometimes it can be useful to check that you're dataset config files are valid, in particular if you use the `import` section (which allows you to include dataset configs from another file). The `fast_curator_check` command can help you by expanding such sections and dumping the result to screen or to an output file.

```
$ fast_curator_check --help
usage: fast_curator_check [-h] [-o OUTPUT] [-f FIELDS] [-p PREFIX]
                          files [files ...]

positional arguments:
  files

optional arguments:
  -h, --help            show this help message and exit
  -o OUTPUT, --output OUTPUT
                        Name of output file list to expand things to
  -f FIELDS, --fields FIELDS
                        Comma-separated list of fields to dump for each
                        dataset
  -p PREFIX, --prefix PREFIX
                        Choose one of the file prefixes to use
```

4.3 fast_carpenter

The `fast_carpenter` is the star of the show. It is what actually converts your event-level datasets to the binned summaries.

The built-in help should tell you everything you need to know:

```
$ fast_carpenter --help
usage: fast_carpenter [-h] [--outdir OUTDIR] [--mode MODE] [--ncores NCORES]
                      [--nblocks-per-dataset NBLOCKS_PER_DATASET]
                      [--nblocks-per-sample NBLOCKS_PER_SAMPLE]
                      [--blocksize BLOCKSIZE] [--quiet] [--profile]
                      [--help-stages [stage-name-regex]]
                      [--help-stages-full stage] [-v]
                      dataset_cfg sequence_cfg

Chop up those trees into nice little tables and dataframes

positional arguments:
  dataset_cfg           Dataset config to run over
  sequence_cfg          Config for how to process events

optional arguments:
  -h, --help            show this help message and exit
  --outdir OUTDIR       Where to save the results
  --mode MODE           Which mode to run in (multiprocessing, htcondor, sge)
  --ncores NCORES       Number of cores to run on
  --nblocks-per-dataset NBLOCKS_PER_DATASET
                        Number of blocks per dataset
  --nblocks-per-sample NBLOCKS_PER_SAMPLE
                        Number of blocks per sample
  --blocksize BLOCKSIZE
                        Number of events per block
  --quiet               Keep progress report quiet
  --profile             Profile the code
  --help-stages [stage-name-regex]
                        Print help specific to the available stages
  --help-stages-full stage
                        Print the full help specific to the available stages
  -v, --version         show program's version number and exit
```

In its simplest form therefore, you can just provide a dataset config and a processing config and run:

```
fast_carpenter datasets.yml processing.yml
```

Quite often you will want to use some of the acceleration options which allow you to process the jobs more quickly using multiple cpu cores, or by running things on a batch system. When you do this, the `fast_carpenter` command will submit tasks to the batch system and wait for them to finish, so you need to make sure the command is not killed in between e.g. by running fast-carpenter on a remote login node and breaking or losing the connection to that login. Use a tool tmux or screen in such cases.

To use multiple CPUs on the local machine, use `--mode multiprocessing` (this is the default, so not generally needed) and specify the number of cores to use, `--ncores 4`.

```
fast_carpenter --ncores 4 datasets.yml processing.yml
```

Alternatively, if you have access to an htcondor or SGE batch system (i.e. `qsub`), then the `fast_carpenter` command can submit many tasks to un at the same time using the batch system. In this case you need to choose an appropriate option for the `--mode` option. In addition the options with `block` in them can control how many events are processed on each task and for each dataset.

Note: For all modes, the `--blocksize` option can be helpful to present fast-carpenter reading too many events into memory in one go. It's default value of 100,000 might be too large, in which case reducing it to some other value (e.g. 20,000) can help. Common symptons of the blocksize being too large are:

- Extremely slow processing, or
- Batch jobs crashing or not being started

4.4 `fast_plotter`

Once you have produced your binned dataframes, the next thing you'll likely want to do is to make these into figures. The `fast_plotter` command and library can help with this. Its command-line interface gives a simple way to make plots from the dataframes with reasonable defaults, whereas its internal functions can be useful when needing more specific ways or presenting results.

Command 'fast_plotter –help' failed: [Errno 2] No such file or directory: 'fast_plotter': 'fast_plotter'

See also:

See the dedicated fast-plotter documentation for more guidance on this package.

CHAPTER 5

The Processing Config

The processing config file tells fast-carpenter what to do with your data and is written in YAML.

An example config file looks like:

```yaml
stages:
    - jet_cleaning: fast_carpenter.Define
    - event_selection: fast_carpenter.CutFlow
    - histogram: fast_carpenter.BinnedDataframe

jet_cleaning:
    variables:
        - BtaggedJets: Jet_bScore > 0.9
        - nBJets: {reduce: count, formula: BtaggedJets}

event_selection:
    selection:
        All:
            - nElectron == 0
            - nJet > 1
            - {reduce: 0, formula: Jet_pt > 100}
            - Any:
                - HT >= 200
                - MHT >= 200

histogram:
    binning:
        - {in: nJet}
        - {in: nBJets}
        - {in: MET, out: met, bins: {edges: [0, 200, 400, 700, 1000]}}
    weights: weight_nominal
```

Other, more complete examples are listed in *Example repositories*.

Tip: Since this is a YAML file, things like anchors and block syntax are totally valid, which can be helpful to define "aliases" or reuse certain parts of a config. For more guidance on YAML, this is a good overview of the concepts and syntax: https://kapeli.com/cheat_sheets/YAML.docset/Contents/Resources/Documents/index.

5.1 Anatomy of the config

The `stages` section This is the most important section of the config because it defines what steps to take with the data. It uses a list of single-length dictionaries, whose key is the name for the stage (e.g. `histogram`) and whose values is the python-importable class that implements it (e.g. `fast_carpenter.BinnedDataframes`). The following sections discuss what are valid stage classes. Lines 1 to 4 of the config above show an example of this section and others can be found in the linked *Example repositories*.

Stage configuration sections Each stage must be given a complete description by adding a top-level section in the yaml file with the same name provided in the `stages` section. This should then contain a dictionary which will be passed as keyword-arguments to the underlying class' init method. Lines 22 to 26 of the above example config file show how the stage called `histogram` is configured. See below for more help on configuring specific stages.

Importing other config files Sometimes it can be helpful to re-use one config in another, for example, defining a list of common variables and event selections, but then changing the BinnedDataframes that are produced. The processing config supports this by using the reserved word `IMPORT` as the key for a stage, followed by the path to the config file to import. If the path starts with `{this_dir}` then the imported file will be located relative to the directory of the importing config file.

For example:

```
- IMPORT: "{this_dir}/another_processing_config.yml"
```

See also:

The interpretation of the processing config is handled by the fast-flow package so its documentation can also be helpful to understanding the basic anatomy and handling.

5.2 Built-in Stages

The list of stages known to fast_carpenter already can be found using the built-in `--help-stages` option.

```
$ fast_carpenter --help-stages
fast_carpenter.Define
   config: variables
   purpose: Creates new variables using a string-based expression.

fast_carpenter.SystematicWeights
   config: weights, out_format=weight_{}, extra_variations=[]
   purpose: Combines multiple weights and variations to produce a single event weight

fast_carpenter.CutFlow
   config: selection_file=None, keep_unique_id=False, selection=None, counter=True,␣
↪weights=None
   purpose: Prevents subsequent stages seeing certain events.

fast_carpenter.SelectPhaseSpace
   config: region_name, **kwargs
   purpose: Creates an event-mask and adds it to the data-space.

```

(continues on next page)

(continued from previous page)

```
fast_carpenter.BinnedDataframe
   config: binning, weights=None, dataset_col=True, pad_missing=False, file_
↪format=None
   purpose: Produces a binned dataframe (a multi-dimensional histogram).

fast_carpenter.BuildAghast
   config: binning, weights=None, dataset_col=True
   purpose: Builds an aghast histogram.

fast_carpenter.EventByEventDataframe
   config: collections, mask=None, flatten=True
   purpose: Write out a pandas dataframe with event-level values
```

Further guidance on the built-in stages can be found using `--help-stages-full` and giving the name of the stage. All the built-in stages of `fast_carpenter` are available directly from the `fast_carpenter` module, e.g. `fast_carpenter.Define`.

See also:

In-depth discussion of the built-in stages and their configuration can be found on the `fast_carpenter` module page: *`fast_carpenter`*, or directly at:

- *`fast_carpenter.Define`*
- *`fast_carpenter.SystematicWeights`*
- *`fast_carpenter.CutFlow`*
- *`fast_carpenter.SelectPhaseSpace`*
- *`fast_carpenter.BinnedDataframe`*
- *`fast_carpenter.BuildAghast`*

Todo: Build that list programmatically, so its always up to date and uses the built-in docstrings for a description.

5.3 Used-defined Stages

fast-carpenter is still evolving, and so it is natural that many analysis tasks cannot be implemented using the existing stages. In this case, it is possible to implement your own stage and making sure it can be imported by python (e.g. by setting the `PYTHONPATH` variable to point to the directory containing its code). The class implementing a custom stage should provide the following methods:

`__init__`(*name*, *out_dir*, *...*)

This is the method that will receive configuration from the config file, creating the stage itself.

Parameters

- **`name`** (*`str`*) – will contain the name of the stage as given in the config file.
- **`out_dir`** (*`path`*) – receives the path to the output directory that should be used if the stage produces output.

Additional arguments can be added, which will be configurable from the processing config file.

`event`(*chunk*)

Called once for each chunk of data.

Parameters **chunk** – provides access to the dataset configuration (`chunk.config`) and the current data-space (`chunk.tree`). Typically one wants an array, or set of arrays representing the data for each event, in which case these can be obtained using:

```
jet_pt = chunk.tree.array("jet_pt")
jet_pt, jet_eta = chunk.tree.arrays(["jet_pt", "jet_eta"],␣
↪outputtype=tuple)
```

If your stage produces a new variable, which you want other stages to be able to see, then use the *new_variable* method:

```
chunk.tree.new_variable("number_good_jets", number_good_jets)
```

For more details on working with `chunk.tree`, see *`fast_carpenter.masked_tree.MaskedUprootTree`*.

Returns `True` or `False` for whether to continue processing the chunk through subsequent stages.

Return type bool

See also:

An example of such a user stage can be seen in the cms_public_tutorial demo repository: https://gitlab.cern.ch/fast-hep/public/fast_cms_public_tutorial/blob/master/cms_hep_tutorial/__init__.py

Warning: Make sure that your stage can be imported by python, most likely by setting the `PYTHONPATH` variable to point to the containing directory. Then to check a stage called `AddMyFancyVar` and defined in a module called `my_custom_module` can be imported, make sure no errors are raised by doing:

```
python -c "import my_custom_module.AddMyFancyVar"
```

Todo: Describe the collector and merge methods to allow a user stage to save results to disk.

CHAPTER 6

Example repositories

- A full demo based on the public CMS tutorial using 2012 data: https://github.com/FAST-HEP/fast_cms_public_tutorial

6.1 Related Presentations

1. IRIS-HEP, 4th March 2019: https://indico.cern.ch/event/802182/contributions/3334624/
2. Analysis Descrition Languages for the LHC, 7th May 2019: https://indico.cern.ch/event/769263/contributions/3406084/

CHAPTER 7

Glossary

cut-flow A series of "cuts" which remove events from the processing.

data-space The current set of variables known to fast-carpenter and passed between stages. Stages can modify the data-space which will affect what subequent stages see. Before any stages have been run, the data-space contains only those variables given in the input datasets.

processing config A YAML-based description of the way the input data should be processed.

processing stage A single step in the processing chain, which can modify the data-space or produce new outputs.

dataset config A YAML-based description of the input files which form the datasets to be processed.

expression A string representing some mathematical manipulation of variables in the data-space.

dataframe A programmatic interface to a table-like representation of data. In the context of fast-carpenter, "dataframe" will usually refer to the `pandas.DataFrame` implementation.

jagged array A generalisation of a multi-dimensional numpy array where the length of each sub-array in the second (and third, and fourth, and so on) dimension can vary. For example, if each event contains a list of particles produced in that event, this would be represented by a jagged array, since there can be different numbers of particles in each event. Typically for fast-carpenter, a jagged array refers to the specific implementation from the awkward-array package

CHAPTER 8

fast_carpenter package

Top-level package for fast-carpenter.

class fast_carpenter.**Define**(*name*, *out_dir*, *variables*)

Bases: object

Creates new variables using a string-based expression.

There are two types of expressions:

- Simple formulae, and
- Reducing formulae.

The essential difference, unfortunately, is an internal one: simple expressions are nearly directly handled by numexpr, whereas reducing expressions add a layer on top.

From a users perspective, however, simple expressions are those that preserve the dimensionality of the input. If one of the input variables represents a list of values for each event (whose length might vary), then the output will contain an equal-length list of values for each event.

If, however, a reducing expression is used, then there will be one less dimension on the resulting variable. In this case, if an input variable has a list of values for each event, the result of the expression will only contain a single value per event.

Parameters **variables** (*list[dictionary]*) – A list of single-length dictionaries whose key is the name of the resulting variable, and whose value is the expression to create it.

Other Parameters

- **name** (*str*) – The name of this stage (handled automatically by fast-flow)
- **out_dir** (*str*) – Where to put the summary table (handled automatically by fast-flow)

Example

```
variables:
  - Muon_pt: "sqrt(Muon_px**2 + Muon_py**2)"
  - Muon_is_good: (Muon_iso > 0.3) & (Muon_pt > 10)
  - NGoodMuons: {reduce: count_nonzero, formula: Muon_is_good}
  - First_Muon_pt: {reduce: 0, formula: Muon_pt}
```

See also:

- `fast_carpenter.define.reductions`– for how reductions are handled and exactly what is valid.
- numexpr: which is used for the internal expression handling.

event(*chunk*)

class fast_carpenter.**SystematicWeights**(*name*, *out_dir*, *weights*, *out_format='weight_{}'*, *extra_variations=[]*)

Bases: `object`

Combines multiple weights and variations to produce a single event weight

To study the impact of systematic uncertainties it is common to re-weight events using a variation of the weights representing, for example, a 1-sigma increase or decrease in the weights. Once there are multiple weight schemes involved writing out each possible combination of these weights becomes tedious and potentially error-prone; this stage makes it easier.

It forms the `nominal` weight for each event by multiplying all nominal weights together, then the specific variation by replacing a given nominal weight with its corresponding "up" or "down" variation.

Each variation of a weight should just be a string giving an expression to use for that variation. This stage then combines these into a single expression by joining each set of variations with "*", i.e. multiplying them together. The final results then use an internal `Define` stage to do the calculation.

Parameters

- **weights** (*dictionary[str, dictionary]*) – A Dictionary of weight variations to combine. The keys in this dictionary will determine how this variation is called in the output variable. The values of this dictionary should either be a single string – the name of the input variable to use for the "nominal" variation, or a dictionary containing any of the keys, `nominal`, `up`, or `down`. Each of these should then have a value providing the expression to use for that variation/
- **out_format** (*str*) – The format string to use to build the name of the output variations. Defaults to "weight_{}". Should contain a pair of empty braces which will be replaced with the name for the current variation, e.g. "nominal" or "PileUp_up".
- **extra_variations** (*list[str]*) – A list of additional variations to allow

Other Parameters

- **name** (*str*) – The name of this stage (handled automatically by fast-flow)
- **out_dir** (*str*) – Where to put the summary table (handled automatically by fast-flow)

Example

```
syst_weights:
  energy_scale: {nominal: WeightEnergyScale, up: WeightEnergyScaleUp, down:␣
↪WeightEnergyScaleDown}
```

(continues on next page)

(continued from previous page)

```
  trigger: TriggerEfficiency
  recon: {nominal: ReconEfficiency, up: ReconEfficiency_up}
```

which will create 4 new variables:

```
weight_nominal =  WeightEnergyScale * TriggerEfficiency * ReconEfficiency
weight_energy_scale_up =  WeightEnergyScaleUp * TriggerEfficiency *␣
↪ReconEfficiency
weight_energy_scale_down =  WeightEnergyScaleDown * TriggerEfficiency *␣
↪ReconEfficiency
weight_recon_up =  WeightEnergyScale * TriggerEfficiency * ReconEfficiency_up
```

event (*chunk*)

class fast_carpenter.**CutFlow** (*name*, *out_dir*, *selection_file=None*, *keep_unique_id=False*, *selection=None*, *counter=True*, *weights=None*)

Bases: `object`

Prevents subsequent stages seeing certain events.

The two most important parameters to understand are the `selection` and `weights` parameters.

Parameters

- **selection** (*str or dict*) – The criteria for selecting events, formed by a nested set of "cuts". Each cut must either be a valid Expressions or a single-length dictionary, with one of `Any` or `All` as the key, and a list of cuts as the value.
- **weights** (*str or list[str], dict[str, str]*) – How to weight events in the output summary table. Must be either a single variable, a list of variables, or a dictionary where the values are variables in the data and keys are the column names that these weights should be called in the output tables.

Example

Mask events using a single cut based on the `nJet` variable being greater than 2 and weight events in the summary table by the `EventWeight` variable:

```
cut_flow_1:
    selection:
        nJet > 2
    weights: EventWeight
```

Mask events by requiring both the `nMuon` variable being greater than 2 and the first `Muon_energy` value in each event being above 20. Don't weight events in the summary table:

```
cut_flow_2:
    selection:
        All:
          - nMuon > 2
          - {reduce: 0, formula: Muon_energy > 20}
```

Mask events by requiring the `nMuon` variable be greater than 2 and either the first `Muon_energy` value in each event is above 20 or the `total_energy` is greater than 100. The summary table will weight events by both the EventWeight variable (called weight_nominal in the table) and the SystUp variable (called weight_syst_up in the summary):

```
cut_flow_3:
    selection:
        All:
          - nMuon > 2
          - Any:
            - {reduce: 0, formula: Muon_energy > 20}
            - total_energy > 100
    weights: {weight_nominal: EventWeight, weight_syst_up: SystUp}
```

Other Parameters

- **name** (*str*) – The name of this stage (handled automatically by fast-flow)
- **out_dir** (*str*) – Where to put the summary table (handled automatically by fast-flow)
- **selection_file** (*str*) – Deprecated
- **keep_unique_id** (*bool*) – If `True`, the summary table will contain a column that gives each cut a unique id. This is used internally to maintain the cut order, and often will not be useful in subsequent manipulation of the output table, so by default this is removed.
- **counter** (*bool*) – Currently unused

Raises `BadCutflowConfig` – If neither or both of `selection` and `selection_file` are provided, or if a bad selection or weight configuration is given.

See also:

`SelectPhaseSpace`: Adds the resulting event-mask as a new variable to the data.

`selection.filters.build_selection()`: Handles the actual creation of the event selection, based on the configuration.

numexpr: which is used for the internal expression handling.

`collector`()

`event`(*chunk*)

`merge`(*rhs*)

class `fast_carpenter.`**`SelectPhaseSpace`**(*name*, *out_dir*, *region_name*, ***kwargs*)

Bases: *`fast_carpenter.selection.stage.CutFlow`*

Creates an event-mask and adds it to the data-space.

This is identical to the *`CutFlow`* class, except that the resulting mask is added to the list of variables in the data-space, rather than being used directly to remove events. This allows multiple "regions" to be defined using different CutFlows in a single configuration.

Parameters **`region_name`** – The name given to the resulting mask when added to back to the data-space.

See also:

`CutFlow`: for a description of the other parameters.

`event`(*chunk*)

class `fast_carpenter.`**`BinnedDataframe`**(*name*, *out_dir*, *binning*, *weights=None*, *dataset_col=True*, *pad_missing=False*, *file_format=None*)

Bases: `object`

Produces a binned dataframe (a multi-dimensional histogram).

def __init__(self, name, out_dir, binning, weights=None, dataset_col=False):

Parameters

- **binning** (*list[dict]*) – A list of dictionaries describing the variables to bin on, and how they should be binned. Each of these dictionaries can contain the following:

Parameter	Default	Description
`in`		The name of the attribute on the event to use.
`out`	same as `in`	The name of the column to be filled in the output dataframe.
`bins`	`None`	Must be either `None` or a dictionary. If a dictionary, it must contain one of the follow sets of key-value pairs: 1. `nbins`, `low`, `high`: which are used to produce a list of bin edges equivalent to: `numpy.linspace(low, high, nbins + 1)` 2. `edges`: which is treated as the list of bin edges directly. If set to `None`, then the input variable is assumed to already be categorical (ie. binned or discrete)

- **weights** (*str or list[str], dict[str, str]*) – How to weight events in the output table. Must be either a single variable, a list of variables, or a dictionary where the values are variables in the data and keys are the column names that these weights should be called in the output tables.
- **file_format** (*str or list[str], dict[str, str]*) – determines the file format to use to save the binned dataframe to disk. Should be either a) a string with the file format, b) a dict containing the keyword *extension* to give the file format and then all other keyword-argument pairs are passed on to the corresponding pandas function, or c) a list of values matching a) or b).
- **dataset_col** (*bool*) – adds an extra binning column with the name for each dataset.
- **pad_missing** (*bool*) – If `False`, any bins that don't contain data are excluded from the stored dataframe. Leaving this `False` can save some disk-space and improve processing

time, particularly if the bins are only very sparsely filled.

Other Parameters

- **name** (*str*) – The name of this stage (handled automatically by fast-flow)
- **out_dir** (*str*) – Where to put the summary table (handled automatically by fast-flow)

Raises `BadBinnedDataframeConfig` – If there is an issue with the binning description.

collector()

event(*chunk*)

merge(*rhs*)

class fast_carpenter.**BuildAghast**(*name*, *out_dir*, *binning*, *weights=None*, *dataset_col=True*)

Bases: `object`

Builds an aghast histogram.

Can be parametrized in the same way as *fast_carpenter.BinnedDataframe* (and actually uses that stage behind the scenes) but additionally writes out a Ghast which can be reloaded with other ghast packages.

See also:

- *fast_carpenter.BinnedDataframe* for a version which only produces binned pandas dataframes.
- The aghast main page: https://github.com/scikit-hep/aghast.

collector()

contents

event(*chunk*)

merge(*rhs*)

8.1 Subpackages

8.1.1 fast_carpenter.backends package

Provides a common interface to select a backend

Each backend is wrapped in a function so that it is only imported if requested

fast_carpenter.backends.**get_alphatwirl**()

fast_carpenter.backends.**get_backend**(*name*)

fast_carpenter.backends.**get_coffea**()

Submodules

fast_carpenter.backends.alphatwirl module

Functions to run a job using alphatwirl

class fast_carpenter.backends.alphatwirl.**AtuprootContext**

Bases: `object`

class fast_carpenter.backends.alphatwirl.**DummyCollector**

Bases: object

collect (**args*, ***kwargs*)

fast_carpenter.backends.alphatwirl.**execute** (*sequence*, *datasets*, *args*)

Run a job using alphatwirl and atuproot

fast_carpenter.backends.coffea module

8.1.2 fast_carpenter.define package

class fast_carpenter.define.**Define** (*name*, *out_dir*, *variables*)

Bases: object

Creates new variables using a string-based expression.

There are two types of expressions:

- Simple formulae, and
- Reducing formulae.

The essential difference, unfortunately, is an internal one: simple expressions are nearly directly handled by numexpr, whereas reducing expressions add a layer on top.

From a users perspective, however, simple expressions are those that preserve the dimensionality of the input. If one of the input variables represents a list of values for each event (whose length might vary), then the output will contain an equal-length list of values for each event.

If, however, a reducing expression is used, then there will be one less dimension on the resulting variable. In this case, if an input variable has a list of values for each event, the result of the expression will only contain a single value per event.

Parameters variables (*list[dictionary]*) – A list of single-length dictionaries whose key is the name of the resulting variable, and whose value is the expression to create it.

Other Parameters

- **name** (*str*) – The name of this stage (handled automatically by fast-flow)
- **out_dir** (*str*) – Where to put the summary table (handled automatically by fast-flow)

Example

```
variables:
  - Muon_pt: "sqrt(Muon_px**2 + Muon_py**2)"
  - Muon_is_good: (Muon_iso > 0.3) & (Muon_pt > 10)
  - NGoodMuons: {reduce: count_nonzero, formula: Muon_is_good}
  - First_Muon_pt: {reduce: 0, formula: Muon_pt}
```

See also:

- *fast_carpenter.define.reductions*– for how reductions are handled and exactly what is valid.
- numexpr: which is used for the internal expression handling.

event (*chunk*)

class fast_carpenter.define.**SystematicWeights**(*name*, *out_dir*, *weights*, *out_format='weight_{}'*, *extra_variations=[]*)

Bases: `object`

Combines multiple weights and variations to produce a single event weight

To study the impact of systematic uncertainties it is common to re-weight events using a variation of the weights representing, for example, a 1-sigma increase or decrease in the weights. Once there are multiple weight schemes involved writing out each possible combination of these weights becomes tedious and potentially error-prone; this stage makes it easier.

It forms the `nominal` weight for each event by multiplying all nominal weights together, then the specific variation by replacing a given nominal weight with its corresponding "up" or "down" variation.

Each variation of a weight should just be a string giving an expression to use for that variation. This stage then combines these into a single expression by joining each set of variations with "*", i.e. multiplying them together. The final results then use an internal `Define` stage to do the calculation.

Parameters

- **weights** (*dictionary[str, dictionary]*) – A Dictionary of weight variations to combine. The keys in this dictionary will determine how this variation is called in the output variable. The values of this dictionary should either be a single string – the name of the input variable to use for the "nominal" variation, or a dictionary containing any of the keys, `nominal`, `up`, or `down`. Each of these should then have a value providing the expression to use for that variation/
- **out_format** (*str*) – The format string to use to build the name of the output variations. Defaults to "weight_{}". Should contain a pair of empty braces which will be replaced with the name for the current variation, e.g. "nominal" or "PileUp_up".
- **extra_variations** (*list[str]*) – A list of additional variations to allow

Other Parameters

- **name** (*str*) – The name of this stage (handled automatically by fast-flow)
- **out_dir** (*str*) – Where to put the summary table (handled automatically by fast-flow)

Example

```
syst_weights:
  energy_scale: {nominal: WeightEnergyScale, up: WeightEnergyScaleUp, down:␣
↪WeightEnergyScaleDown}
  trigger: TriggerEfficiency
  recon: {nominal: ReconEfficiency, up: ReconEfficiency_up}
```

which will create 4 new variables:

```
weight_nominal =  WeightEnergyScale * TriggerEfficiency * ReconEfficiency
weight_energy_scale_up =  WeightEnergyScaleUp * TriggerEfficiency *␣
↪ReconEfficiency
weight_energy_scale_down =  WeightEnergyScaleDown * TriggerEfficiency *␣
↪ReconEfficiency
weight_recon_up =  WeightEnergyScale * TriggerEfficiency * ReconEfficiency_up
```

event(*chunk*)

Submodules

fast_carpenter.define.reductions module

`fast_carpenter.define.reductions.`**`get_pandas_reduction`**(*stage_name*, *reduction*)

fast_carpenter.define.systematics module

`exception` `fast_carpenter.define.systematics.`**`BadSystematicWeightsConfig`**

Bases: `Exception`

`class` `fast_carpenter.define.systematics.`**`SystematicWeights`**(*name*, *out_dir*, *weights*, *out_format='weight_{}'*, *extra_variations=[]*)

Bases: `object`

Combines multiple weights and variations to produce a single event weight

To study the impact of systematic uncertainties it is common to re-weight events using a variation of the weights representing, for example, a 1-sigma increase or decrease in the weights. Once there are multiple weight schemes involved writing out each possible combination of these weights becomes tedious and potentially error-prone; this stage makes it easier.

It forms the `nominal` weight for each event by multiplying all nominal weights together, then the specific variation by replacing a given nominal weight with its corresponding "up" or "down" variation.

Each variation of a weight should just be a string giving an expression to use for that variation. This stage then combines these into a single expression by joining each set of variations with "*", i.e. multiplying them together. The final results then use an internal `Define` stage to do the calculation.

Parameters

- **`weights`** (*`dictionary[str, dictionary]`*) – A Dictionary of weight variations to combine. The keys in this dictionary will determine how this variation is called in the output variable. The values of this dictionary should either be a single string – the name of the input variable to use for the "nominal" variation, or a dictionary containing any of the keys, `nominal`, `up`, or `down`. Each of these should then have a value providing the expression to use for that variation/
- **`out_format`** (*`str`*) – The format string to use to build the name of the output variations. Defaults to "weight_{}". Should contain a pair of empty braces which will be replaced with the name for the current variation, e.g. "nominal" or "PileUp_up".
- **`extra_variations`** (*`list[str]`*) – A list of additional variations to allow

Other Parameters

- **name** (*str*) – The name of this stage (handled automatically by fast-flow)
- **out_dir** (*str*) – Where to put the summary table (handled automatically by fast-flow)

Example

```
syst_weights:
  energy_scale: {nominal: WeightEnergyScale, up: WeightEnergyScaleUp, down:␣
↪WeightEnergyScaleDown}
  trigger: TriggerEfficiency
  recon: {nominal: ReconEfficiency, up: ReconEfficiency_up}
```

which will create 4 new variables:

```
weight_nominal =  WeightEnergyScale * TriggerEfficiency * ReconEfficiency
weight_energy_scale_up =  WeightEnergyScaleUp * TriggerEfficiency *␣
↪ReconEfficiency
weight_energy_scale_down =  WeightEnergyScaleDown * TriggerEfficiency *␣
↪ReconEfficiency
weight_recon_up =  WeightEnergyScale * TriggerEfficiency * ReconEfficiency_up
```

event (*chunk*)

fast_carpenter.define.variables module

exception fast_carpenter.define.variables.**BadVariablesConfig**

Bases: Exception

class fast_carpenter.define.variables.**CalculationCfg** (*name*, *expression*, *reduction*, *fill_missing*, *mask*)

Bases: tuple

expression

Alias for field number 1

fill_missing

Alias for field number 3

mask

Alias for field number 4

name

Alias for field number 0

reduction

Alias for field number 2

class fast_carpenter.define.variables.**Define** (*name*, *out_dir*, *variables*)

Bases: object

Creates new variables using a string-based expression.

There are two types of expressions:

- Simple formulae, and
- Reducing formulae.

The essential difference, unfortunately, is an internal one: simple expressions are nearly directly handled by numexpr, whereas reducing expressions add a layer on top.

From a users perspective, however, simple expressions are those that preserve the dimensionality of the input. If one of the input variables represents a list of values for each event (whose length might vary), then the output will contain an equal-length list of values for each event.

If, however, a reducing expression is used, then there will be one less dimension on the resulting variable. In this case, if an input variable has a list of values for each event, the result of the expression will only contain a single value per event.

Parameters **variables** (*list[dictionary]*) – A list of single-length dictionaries whose key is the name of the resulting variable, and whose value is the expression to create it.

Other Parameters

- **name** (*str*) – The name of this stage (handled automatically by fast-flow)
- **out_dir** (*str*) – Where to put the summary table (handled automatically by fast-flow)

Example

```
variables:
  - Muon_pt: "sqrt(Muon_px**2 + Muon_py**2)"
  - Muon_is_good: (Muon_iso > 0.3) & (Muon_pt > 10)
  - NGoodMuons: {reduce: count_nonzero, formula: Muon_is_good}
  - First_Muon_pt: {reduce: 0, formula: Muon_pt}
```

See also:

- *fast_carpenter.define.reductions*– for how reductions are handled and exactly what is valid.
- numexpr: which is used for the internal expression handling.

event (*chunk*)

class fast_carpenter.define.variables.**DefinePandas** (*name*, *out_dir*, *variables*)
Bases: object

event (*chunk*)

fast_carpenter.define.variables.**full_evaluate** (*tree*, *expression*, *fill_missing*, *mask=None*, *reduction=None*)

8.1.3 fast_carpenter.selection package

class fast_carpenter.selection.**CutFlow** (*name*, *out_dir*, *selection_file=None*, *keep_unique_id=False*, *selection=None*, *counter=True*, *weights=None*)
Bases: object

Prevents subsequent stages seeing certain events.

The two most important parameters to understand are the `selection` and `weights` parameters.

Parameters

- **selection** (*str or dict*) – The criteria for selecting events, formed by a nested set of "cuts". Each cut must either be a valid Expressions or a single-length dictionary, with one of `Any` or `All` as the key, and a list of cuts as the value.
- **weights** (*str or list[str], dict[str, str]*) – How to weight events in the output summary table. Must be either a single variable, a list of variables, or a dictionary where the values are variables in the data and keys are the column names that these weights should be called in the output tables.

Example

Mask events using a single cut based on the `nJet` variable being greater than 2 and weight events in the summary table by the `EventWeight` variable:

```
cut_flow_1:
    selection:
        nJet > 2
    weights: EventWeight
```

Mask events by requiring both the `nMuon` variable being greater than 2 and the first `Muon_energy` value in each event being above 20. Don't weight events in the summary table:

```
cut_flow_2:
    selection:
        All:
          - nMuon > 2
          - {reduce: 0, formula: Muon_energy > 20}
```

Mask events by requiring the `nMuon` variable be greater than 2 and either the first `Muon_energy` value in each event is above 20 or the `total_energy` is greater than 100. The summary table will weight events by both the EventWeight variable (called weight_nominal in the table) and the SystUp variable (called weight_syst_up in the summary):

```
cut_flow_3:
    selection:
        All:
          - nMuon > 2
          - Any:
            - {reduce: 0, formula: Muon_energy > 20}
            - total_energy > 100
    weights: {weight_nominal: EventWeight, weight_syst_up: SystUp}
```

Other Parameters

- **name** (*str*) – The name of this stage (handled automatically by fast-flow)
- **out_dir** (*str*) – Where to put the summary table (handled automatically by fast-flow)
- **selection_file** (*str*) – Deprecated
- **keep_unique_id** (*bool*) – If `True`, the summary table will contain a column that gives each cut a unique id. This is used internally to maintain the cut order, and often will not be useful in subsequent manipulation of the output table, so by default this is removed.
- **counter** (*bool*) – Currently unused

Raises `BadCutflowConfig` – If neither or both of `selection` and `selection_file` are provided, or if a bad selection or weight configuration is given.

See also:

`SelectPhaseSpace`: Adds the resulting event-mask as a new variable to the data.

`selection.filters.build_selection()`: Handles the actual creation of the event selection, based on the configuration.

numexpr: which is used for the internal expression handling.

`collector`()

`event`(*chunk*)

`merge`(*rhs*)

class `fast_carpenter.selection.`**`SelectPhaseSpace`**(*name*, *out_dir*, *region_name*, ***kwargs*)

Bases: *`fast_carpenter.selection.stage.CutFlow`*

Creates an event-mask and adds it to the data-space.

This is identical to the `CutFlow` class, except that the resulting mask is added to the list of variables in the data-space, rather than being used directly to remove events. This allows multiple "regions" to be defined using different CutFlows in a single configuration.

Parameters **`region_name`** – The name given to the resulting mask when added to back to the data-space.

See also:

`CutFlow`: for a description of the other parameters.

`event`(*chunk*)

Submodules

fast_carpenter.selection.filters module

class `fast_carpenter.selection.filters.`**`All`**(*selection*, *depth*, *cut_id*, *weights*)

Bases: *`fast_carpenter.selection.filters.BaseFilter`*

class `fast_carpenter.selection.filters.`**`Any`**(*selection*, *depth*, *cut_id*, *weights*)

Bases: *`fast_carpenter.selection.filters.BaseFilter`*

class `fast_carpenter.selection.filters.`**`BaseFilter`**(*selection*, *depth*, *cut_id*, *weights*)

Bases: `object`

`columns`

`increment_counters`(*data*, *is_mc*, *excl*, *before*, *after*)

`index_values`

`merge`(*rhs*)

`to_dataframe`()

`values`

class `fast_carpenter.selection.filters.`**`Counter`**(*weights*)

Bases: `object`

`add`(*rhs*)

`counts`

static `get_unweighted_increment`(*data*, *mask*)

static `get_weighted_increment`(*weight_names*, *data*, *mask*)

`increment`(*data*, *is_mc*, *mask=None*)

class `fast_carpenter.selection.filters.`**`OuterCounterIncrementer`**(**args*, ***kwargs*)

Bases: *`fast_carpenter.selection.filters.BaseFilter`*

class `fast_carpenter.selection.filters.`**`ReduceSingleCut`**(*stage_name*, *depth*, *cut_id*, *weights*, *selection*)

Bases: *`fast_carpenter.selection.filters.BaseFilter`*

class `fast_carpenter.selection.filters.`**`SingleCut`**(*selection*, *depth*, *cut_id*, *weights*)

Bases: `fast_carpenter.selection.filters.BaseFilter`

`fast_carpenter.selection.filters.`**`build_selection`**(*stage_name*, *config*, *weights=[]*)

Creates event selectors based on the configuration.

Parameters

- **`stage_name`** – Used to help in error messages.
- **`config`** – The event selection configuration.
- **`weights`** – How to weight events, used to produce the resulting cut efficiency table.

Raises `RuntimeError` – if any of the configurations are invalid.

`fast_carpenter.selection.filters.`**`handle_config`**(*stage_name*, *config*, *weights*, *depth=0*, *cut_id=[0]*)

`fast_carpenter.selection.filters.`**`safe_and`**(*left*, *right*)

`fast_carpenter.selection.filters.`**`safe_or`**(*left*, *right*)

fast_carpenter.selection.stage module

Stages to remove events from subsequent stages

Provides two stages:

- `CutFlow` – Prevent subsequent stages from seeing certain events,
- `SelectPhaseSpace` – Create a new variable which can be used as a mask

Both stages are configured very similarly, and both stages produce an output table describing how many events pass each subsequent cut to make it into the final mask.

class `fast_carpenter.selection.stage.`**`CutFlow`**(*name*, *out_dir*, *selection_file=None*, *keep_unique_id=False*, *selection=None*, *counter=True*, *weights=None*)

Bases: `object`

Prevents subsequent stages seeing certain events.

The two most important parameters to understand are the `selection` and `weights` parameters.

Parameters

- **`selection`** (*`str` or `dict`*) – The criteria for selecting events, formed by a nested set of "cuts". Each cut must either be a valid Expressions or a single-length dictionary, with one of `Any` or `All` as the key, and a list of cuts as the value.
- **`weights`** (*`str` or `list[str]`, `dict[str, str]`*) – How to weight events in the output summary table. Must be either a single variable, a list of variables, or a dictionary where the values are variables in the data and keys are the column names that these weights should be called in the output tables.

Example

Mask events using a single cut based on the `nJet` variable being greater than 2 and weight events in the summary table by the `EventWeight` variable:

```
cut_flow_1:
    selection:
        nJet > 2
    weights: EventWeight
```

Mask events by requiring both the `nMuon` variable being greater than 2 and the first `Muon_energy` value in each event being above 20. Don't weight events in the summary table:

```
cut_flow_2:
    selection:
        All:
          - nMuon > 2
          - {reduce: 0, formula: Muon_energy > 20}
```

Mask events by requiring the `nMuon` variable be greater than 2 and either the first `Muon_energy` value in each event is above 20 or the `total_energy` is greater than 100. The summary table will weight events by both the EventWeight variable (called weight_nominal in the table) and the SystUp variable (called weight_syst_up in the summary):

```
cut_flow_3:
    selection:
        All:
          - nMuon > 2
          - Any:
            - {reduce: 0, formula: Muon_energy > 20}
            - total_energy > 100
    weights: {weight_nominal: EventWeight, weight_syst_up: SystUp}
```

Other Parameters

- **name** (*str*) – The name of this stage (handled automatically by fast-flow)
- **out_dir** (*str*) – Where to put the summary table (handled automatically by fast-flow)
- **selection_file** (*str*) – Deprecated
- **keep_unique_id** (*bool*) – If `True`, the summary table will contain a column that gives each cut a unique id. This is used internally to maintain the cut order, and often will not be useful in subsequent manipulation of the output table, so by default this is removed.
- **counter** (*bool*) – Currently unused

Raises `BadCutflowConfig` – If neither or both of `selection` and `selection_file` are provided, or if a bad selection or weight configuration is given.

See also:

`SelectPhaseSpace`: Adds the resulting event-mask as a new variable to the data.

`selection.filters.build_selection()`: Handles the actual creation of the event selection, based on the configuration.

numexpr: which is used for the internal expression handling.

`collector`()

`event`(*chunk*)

`merge`(*rhs*)

class `fast_carpenter.selection.stage.`**`SelectPhaseSpace`**(*name*, *out_dir*, *region_name*, ***kwargs*)

Bases: *`fast_carpenter.selection.stage.CutFlow`*

Creates an event-mask and adds it to the data-space.

This is identical to the *`CutFlow`* class, except that the resulting mask is added to the list of variables in the data-space, rather than being used directly to remove events. This allows multiple "regions" to be defined using different CutFlows in a single configuration.

Parameters **`region_name`** – The name given to the resulting mask when added to back to the data-space.

See also:

`CutFlow`: for a description of the other parameters.

`event`(*chunk*)

8.1.4 fast_carpenter.summary package

class `fast_carpenter.summary.`**`BuildAghast`**(*name*, *out_dir*, *binning*, *weights=None*, *dataset_col=True*)

Bases: `object`

Builds an aghast histogram.

Can be parametrized in the same way as *`fast_carpenter.BinnedDataframe`* (and actually uses that stage behind the scenes) but additionally writes out a Ghast which can be reloaded with other ghast packages.

See also:

- *`fast_carpenter.BinnedDataframe`* for a version which only produces binned pandas dataframes.
- The aghast main page: https://github.com/scikit-hep/aghast.

`collector`()

`contents`

`event`(*chunk*)

`merge`(*rhs*)

class `fast_carpenter.summary.`**`BinnedDataframe`**(*name*, *out_dir*, *binning*, *weights=None*, *dataset_col=True*, *pad_missing=False*, *file_format=None*)

Bases: `object`

Produces a binned dataframe (a multi-dimensional histogram).

def __init__(self, name, out_dir, binning, weights=None, dataset_col=False):

Parameters

- **`binning`** (*`list[dict]`*) – A list of dictionaries describing the variables to bin on, and how they should be binned. Each of these dictionaries can contain the following:

Parameter	Default	Description
`in`		The name of the attribute on the event to use.
`out`	same as `in`	The name of the column to be filled in the output dataframe.
`bins`	`None`	Must be either `None` or a dictionary. If a dictionary, it must contain one of the follow sets of key-value pairs: 1. `nbins`, `low`, `high`: which are used to produce a list of bin edges equivalent to: `numpy.linspace(low, high, nbins + 1)` 2. `edges`: which is treated as the list of bin edges directly. If set to `None`, then the input variable is assumed to already be categorical (ie. binned or discrete)

- **`weights`** (*`str or list[str], dict[str, str]`*) – How to weight events in the output table. Must be either a single variable, a list of variables, or a dictionary where the values are variables in the data and keys are the column names that these weights should be called in the output tables.
- **`file_format`** (*`str or list[str], dict[str, str]`*) – determines the file format to use to save the binned dataframe to disk. Should be either a) a string with the file format, b) a dict containing the keyword *extension* to give the file format and then all other keyword-argument pairs are passed on to the corresponding pandas function, or c) a list of values matching a) or b).
- **`dataset_col`** (*`bool`*) – adds an extra binning column with the name for each dataset.
- **`pad_missing`** (*`bool`*) – If `False`, any bins that don't contain data are excluded from the stored dataframe. Leaving this `False` can save some disk-space and improve processing time, particularly if the bins are only very sparsely filled.

Other Parameters

- **name** (*str*) – The name of this stage (handled automatically by fast-flow)
- **out_dir** (*str*) – Where to put the summary table (handled automatically by fast-flow)

Raises `BadBinnedDataframeConfig` – If there is an issue with the binning description.

collector()

event(*chunk*)

merge(*rhs*)

class fast_carpenter.summary.**EventByEventDataframe**(*name*, *out_dir*, *collections*, *mask=None*, *flatten=True*)

Bases: object

Write out a pandas dataframe with event-level values

collector()

event(*chunk*)

merge(*rhs*)

Submodules

fast_carpenter.summary.aghast module

class fast_carpenter.summary.aghast.**BuildAghast**(*name*, *out_dir*, *binning*, *weights=None*, *dataset_col=True*)

Bases: object

Builds an aghast histogram.

Can be parametrized in the same way as *fast_carpenter.BinnedDataframe* (and actually uses that stage behind the scenes) but additionally writes out a Ghast which can be reloaded with other ghast packages.

See also:

- *fast_carpenter.BinnedDataframe* for a version which only produces binned pandas dataframes.
- The aghast main page: https://github.com/scikit-hep/aghast.

collector()

contents

event(*chunk*)

merge(*rhs*)

class fast_carpenter.summary.aghast.**Collector**(*filename*, *axes*, *edges*, *by_dataset*)

Bases: object

collect(*dataset_readers_list*)

fast_carpenter.summary.aghast.**bin_one_dimension**(*low=None*, *high=None*, *nbins=None*, *edges=None*, ***kwargs*)

fast_carpenter.summary.aghast.**complete_axes**(*axes*, *df_index*)

fast_carpenter.summary.aghast.**convert_to_counters**(*df*)

fast_carpenter.summary.binned_dataframe module

Summarize the data by producing binned and possibly weighted counts of the data.

class fast_carpenter.summary.binned_dataframe.**BinnedDataframe**(*name*, *out_dir*, *binning*, *weights=None*, *dataset_col=True*, *pad_missing=False*, *file_format=None*)

Bases: object

Produces a binned dataframe (a multi-dimensional histogram).

def __init__(self, name, out_dir, binning, weights=None, dataset_col=False):

Parameters

- **binning** (*list[dict]*) – A list of dictionaries describing the variables to bin on, and how they should be binned. Each of these dictionaries can contain the following:

Parameter	Default	Description
`in`		The name of the attribute on the event to use.
`out`	same as `in`	The name of the column to be filled in the output dataframe.
`bins`	`None`	Must be either `None` or a dictionary. If a dictionary, it must contain one of the follow sets of key-value pairs: 1. `nbins`, `low`, `high`: which are used to produce a list of bin edges equivalent to: `numpy.linspace(low, high, nbins + 1)` 2. `edges`: which is treated as the list of bin edges directly. If set to `None`, then the input variable is assumed to already be categorical (ie. binned or discrete)

- **weights** (*str or list[str], dict[str, str]*) – How to weight events in the output table. Must be either a single variable, a list of variables, or a dictionary where the values are variables in the data and keys are the column names that these weights should be called in the output tables.

- **file_format** (*str or list[str], dict[str, str]*) – determines the file format to use to save the binned dataframe to disk. Should be either a) a string with the file format, b) a dict containing the keyword *extension* to give the file format and then all other keyword-argument pairs are passed on to the corresponding pandas function, or c) a list of values matching a) or b).
- **dataset_col** (*bool*) – adds an extra binning column with the name for each dataset.
- **pad_missing** (*bool*) – If `False`, any bins that don't contain data are excluded from the stored dataframe. Leaving this `False` can save some disk-space and improve processing time, particularly if the bins are only very sparsely filled.

Other Parameters

- **name** (*str*) – The name of this stage (handled automatically by fast-flow)
- **out_dir** (*str*) – Where to put the summary table (handled automatically by fast-flow)

Raises `BadBinnedDataframeConfig` – If there is an issue with the binning description.

collector()

event(*chunk*)

merge(*rhs*)

class `fast_carpenter.summary.binned_dataframe.`**Collector**(*filename, dataset_col, binnings, file_format*)

Bases: `object`

collect(*dataset_readers_list, doReturn=True, writeFiles=True*)

valid_ext = {'dta': 'stata', 'h5': 'hdf', 'msg': 'msgpack', 'p': 'pickle', 'pkl':

`fast_carpenter.summary.binned_dataframe.`**combined_dataframes**(*dataset_readers_list, dataset_col, binnings=None*)

`fast_carpenter.summary.binned_dataframe.`**densify_dataframe**(*in_df, binnings*)

`fast_carpenter.summary.binned_dataframe.`**explode**(*df*)

Based on this answer: https://stackoverflow.com/questions/12680754/split-explode-pandas -dataframe-string-entry-to-separate-rows/40449726#40449726

fast_carpenter.summary.binning_config module

exception `fast_carpenter.summary.binning_config.`**BadBinnedDataframeConfig**

Bases: `Exception`

`fast_carpenter.summary.binning_config.`**bin_one_dimension**(*low=None, high=None, nbins=None, edges=None, overflow=True, underflow=True*)

`fast_carpenter.summary.binning_config.`**create_binning_list**(*name, bin_list, make_bins=None*)

`fast_carpenter.summary.binning_config.`**create_file_format**(*stage_name, file_format*) `fast_carpenter.summary.binning_config.`**`create_one_dimension`**(*stage_name*, *_in*, *_out=None*, *_bins=None*, *_index=None*, *make_bins=None*)

`fast_carpenter.summary.binning_config.`**`create_weights`**(*stage_name*, *weights*)

fast_carpenter.summary.event_level_dataframe module

`class` `fast_carpenter.summary.event_level_dataframe.`**`Collector`**(*filename*)

Bases: `object`

`collect`(*dataset_readers_list*)

`class` `fast_carpenter.summary.event_level_dataframe.`**`EventByEventDataframe`**(*name*, *out_dir*, *col-lec-tions*, *mask=None*, *flat-ten=True*)

Bases: `object`

Write out a pandas dataframe with event-level values

`collector`()

`event`(*chunk*)

`merge`(*rhs*)

fast_carpenter.summary.import_aghast module

`class` `fast_carpenter.summary.import_aghast.`**`AghastCatcher`**

Bases: `object`

8.2 Submodules

8.2.1 fast_carpenter.event_builder module

`class` `fast_carpenter.event_builder.`**`BEventsWrapped`**(*tree*, **args*, ***kwargs*)

Bases: `atuproot.BEvents.BEvents`

`class` `fast_carpenter.event_builder.`**`EventBuilder`**(*config*)

Bases: `object`

`class` `fast_carpenter.event_builder.`**`EventRanger`**

Bases: `object`

`entries_in_block`

`set_owner`(*owner*)

`start_entry`

stop_entry

8.2.2 fast_carpenter.expressions module

fast_carpenter.expressions.**get_branches**(*cut*, *valid*)

fast_carpenter.expressions.**evaluate**(*tree*, *expression*)

8.2.3 fast_carpenter.help module

class fast_carpenter.help.**StageGuidanceHelper**(*stage_class*, *module_name*)

Bases: object

class_name

docstring(*nlines=-1*)

matches(*regex*)

parameters()

stage

fast_carpenter.help.**format_signature**(*args*, *vargs*, *kwargs*, *defaults*, *annots*)

fast_carpenter.help.**get_signature**(*function*)

fast_carpenter.help.**help_stages**(*stage_name*, *full_output=False*)

8.2.4 fast_carpenter.masked_tree module

class fast_carpenter.masked_tree.**MaskedUprootTree**(*tree*, *event_ranger*, *mask=None*)

Bases: object

class PandasWrap(*owner*)

Bases: object

df(**args*, ***kwargs*)

apply_mask(*new_mask*)

array(**args*, ***kwargs*)

arrays(**args*, ***kwargs*)

mask

pandas

reset_mask()

unmasked_array(**args*, ***kwargs*)

unmasked_arrays(**args*, ***kwargs*)

fast_carpenter.masked_tree.**mask_df**(*df*, *mask*, *start_event*)

8.2.5 fast_carpenter.tree_wrapper module

This has to be what is probably the hackiest piece of code I've ever written. It's very tightly coupled to uproot, and just by importing it will change the way uproot works. However, it allows me to achieve the functionality of adding a branch to uproot trees with no changes to actual code in uproot and with minimal coding on my side…

class `fast_carpenter.tree_wrapper.`**WrappedTree**(*tree*, *event_ranger*)
Bases: `object`

class FakeBranch(*name*, *values*, *event_ranger*)
Bases: `object`

array(*entrystart=None*, *entrystop=None*, *blocking=True*, ***kws*)

class PandasWrap(*owner*)
Bases: `object`

df(**args*, ***kwargs*)

array(**args*, ***kwargs*)

arrays(**args*, ***kwargs*)

itervalues(**args*, ***kwargs*)

new_variable(*name*, *value*)

pandas

reset_cache()

update_array_args(*kwargs*)

`fast_carpenter.tree_wrapper.`**recursive_type_wrap**(*array*)

class `fast_carpenter.tree_wrapper.`**wrapped_asgenobj**(*cls*, *context*, *skipbytes*)
Bases: `uproot.interp.objects.asgenobj`

finalize(**args*, ***kwargs*)

`fast_carpenter.tree_wrapper.`**wrapped_interpret**(*branch*, **args*, ***kwargs*)

8.2.6 fast_carpenter.utils module

`fast_carpenter.utils.`**mkdir_p**(*path*)

8.2.7 fast_carpenter.version module

Defines version of codebase

`fast_carpenter.version.`**split_version**(*version*)
Split a semantic version string into a version_info tuple

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index