Lena Documentation

Release 0.1

Yaroslav Nikitenko

Apr 16, 2020

Contents:

1 Tutorial **3**
- 1.1 Introduction to Lena . . . 3
- 1.2 Split . . . 11
- 1.3 Answers to exercises . . . 23

2 Reference **29**
- 2.1 Context . . . 29
- 2.2 lena.core . . . 32
- 2.3 Flow . . . 38
- 2.4 math package . . . 45
- 2.5 Output . . . 51
- 2.6 Structures . . . 55
- 2.7 Variables . . . 61

3 Indices and tables **65**

4 Installation **67**
- 4.1 From pip . . . 67
- 4.2 From github . . . 67
- 4.3 Additional programs . . . 67

5 Documentation **69**

6 License **71**

7 Alternatives **73**

Python Module Index **75**

Index **77**

Lena is an architectural framework for data analysis. It is written in Python and works with Python versions 2, 3 and PyPy.

These are Lena features from programming point of view:

- modularity, weak coupling. Algorithms can be easily added, replaced or reused.
- performance. Lazy evaluation is good for memory and speed. Several analyses can be done reading data once. PyPy with just-in-time compiler can be used if needed.
- code reuse. Logic is separated from presentation. One template can be used for several plots.
- rapid development. One can run only those elements which already work. During development only a small subset of data can be analysed. Results of heavy calculations can be easily saved.
- easy to understand, structured and beautiful code.

From data analysis perspective:

- comparison of analyses with arbitrary changes (including different input data or algorithms).
- algorithm reuse for a subset of data (for example, to see how an algorithm works at different coordinates in the detector).
- analysis consistency. When we run several algorithms for same data or reuse an algorithm, we are sure that we use same data and algorithm.
- algorithms can be combined into a more complex analysis.

Lena originated from experimental neutrino physics and is named after a great Siberian river.

CHAPTER 1

Tutorial

1.1 Introduction to Lena

In our data analysis we often face changing data or algorithms. For example, we may want to see how our analysis works for another dataset or for a specific subset of the data. We may also want to use different algorithms and compare their results.

To handle this gracefully, we must be able to easily change or extend our code at any specified point. The idea of Lena is to split our code into small independent blocks, which are later composed together. The tutorial will show us how to do that and what implications this idea will have for our code.

Contents

- *The three ideas behind Lena*
  - *1. Sequences and elements*
  - *2. Lazy evaluation*
  - *3. Context*
- *A real analysis example*
- *Elements for development*

1.1.1 The three ideas behind Lena

1. Sequences and elements

The basic idea of *Lena* is to join our computations into sequences. Sequences consist of elements.

The simplest *Lena* program may be the following. We use a sequence with one element, an anonymous function, which is created in Python by *lambda* keyword:

```
>>> from __future__ import print_function
>>> from lena.core import Sequence
>>> s = Sequence(
...     lambda i: pow(-1, i) * (2 * i + 1),
... )
>>> results = s.run([0, 1, 2, 3])
>>> for res in results:
...     print(res)
1 -3 5 -7
```

Lena supports both Python versions, 2 and 3. It is simple to do it in your code, if you want. The first line allows to use *print()* for any version of Python. The next line imports a *Lena* class.

A *Sequence* can be initialized from several elements. To make the *Sequence* do the actual work, we use its method *run*. *Run*'s argument is an iterable (in this case a list of four numbers).

To obtain all results, we iterate them in the cycle *for*.

Let us move to a more complex example. It is often convenient not to pass any data to a function, which gets it somewhere else itself. In this case use a sequence *Source*:

```
from lena.core import Sequence, Source
from lena.flow import CountFrom, ISlice

s = Sequence(
    lambda i: pow(-1, i) * (2 * i + 1),
)
spi = Source(
    CountFrom(0),
    s,
    ISlice(10**6),
    lambda x: 4./x,
    Sum(),
)
results = list(spi())
# [3.1415916535897743]
```

The first element in *Source* must have a *__call__* special method, which accepts no arguments and generates values itself. These values are propagated by the sequence: each following element receives as input the results of the previous element, and the sequence call gives the results of the last element.

A *CountFrom* is an element, which produces an infinite series of numbers. *Elements* must be functions or objects, but not classes[1]. We pass the starting number to *CountFrom* during its initialization (in this case zero). The initialization arguments of *CountFrom* are *start* (by default zero) and *step* (by default one).

The following elements of a *Source* (if present) must be callables or objects with a method called *run*. They can form a simple *Sequence* themselves.

Sequences can be joined together. In our example, we use our previously defined sequence *s* as the second element of *Source*. There would be no difference if we used the lambda from *s* instead of *s*.

A *Sequence* can be placed before, after or inside another *Sequence*. A *Sequence* can't be placed before a *Source*, because it doesn't accept any incoming flow.

Note: If we try to instantiate a *Sequence* with a *Source* in the middle, the initialization will instantly fail and throw a *LenaTypeError* (a subtype of Python's *TypeError*).

[1] This possibility may be added in the future.

All *Lena* exceptions are subclassed from *LenaException*. They are raised as early as possible (not after a long analysis was fulfilled and discarded).

Since we can't use an infinite series in practice, we must stop it at some point. We take the first million of its items using an *ISlice* element. *ISlice* and *CountFrom* are similar to *islice* and *count* functions from Python's standard library module *itertools*. *ISlice* can also be initialized with *start, stop[, step]* arguments, which allow to skip some initial or final subset of data (defined by its index), or take each *step*-th item (if the *step* is two, use all even indices from zero).

We apply a further transformation of data with a *lambda*, and sum the resulting values.

Finally, we materialize the results in a *list*, and obtain a rough approximation of *pi*.

2. Lazy evaluation

Let us look at the last element of the previous sequence. Its class has a method *run*, which accepts the incoming *flow*:

```
class Sum():
    def run(self, flow):
        s = 0
        for val in flow:
            s += val
        yield s
```

Note that we give the final number not with *return*, but with *yield*. *Yield* is a Python keyword, which turns a usual function into a *generator*.

Generators are Python's implementation of *lazy evaluation*. In the very first example we used a line

```
>>> results = s.run([0, 1, 2, 3])
```

The method *run* of a *Sequence* is a generator. When we call a generator, we obtain the result, but no computation really occurs, no statement from the generator's code is executed. To actually calculate the results, the generator must be materialized. This can be done in a container (like a *list* or *tuple*) or in a cycle:

```
>>> for res in results:
...     print(res)
```

Lazy evaluation is good for:

- performance. Reading data files may be one of the longest steps in simple data analysis. Since lazy evaluation uses only one value at a time, this value can be used immediately without waiting when the reading of the whole data set is finished. This allows us to make a complete analysis in almost the same time as just to read the input data.
- low memory impact. Data is immediately used and not stored anywhere. This allows us to analyse data sets larger than the physical memory, and thus makes our program *scalable*.

Lazy evaluation is very easy to implement in Python using a *yield* keyword. Generators must be carefully distinguished from ordinary functions in Lena. If an object inside a sequence has a *run* method, it is assumed to be a generator. Otherwise, if the object is callable, it is assumed to be a function, which makes some simple transformation of the input value.

Generators can yield zero or multiple values. Use them to alter or reduce data *flow*. Use functions or callable objects for calculations that accept and return a single *value*.

3. Context

Lena's goal is to cover the data analysis process from beginning to end. The final results of an analysis are tables and plots, which can be used by people.

Lena doesn't draw anything itself, but relies on other programs. It uses a library *Jinja* to render text templates. There are no predefined templates or magic constants in Lena, and users have to write their own ones. An example for a one-dimensional LaTeX plot is:

```
% histogram_1d.tex
\documentclass{standalone}
\usepackage{tikz}
\usepackage{pgfplots}
\pgfplotsset{compat=1.15}

\begin{document}
\begin{tikzpicture}
\begin{axis}[]
\addplot [
    const plot,
]
table [col sep=comma, header=false] {\VAR{ output.filepath }};
\end{axis}
\end{tikzpicture}
\end{document}
```

This is a simple TikZ template except for one line: *\VAR{ output.filepath }*. *\VAR{ var }* is substituted with the actual value of *var* during rendering. This allows to use one template for different data, instead of creating many identical files for each plot. In that example, variable *output.filepath* is passed in a rendering *context*.

A more sophisticated example could be the following:

```
\BLOCK{ set var = variable if variable else '' }
\begin{tikzpicture}
\begin{axis}[
    \BLOCK{ if var.latex_name }
        xlabel = { $\VAR{ var.latex_name }$
        \BLOCK{ if var.unit }
            [$\mathrm{\VAR{ var.unit }}$]
        \BLOCK{ endif }
        },
    \BLOCK{ endif }
]
...
```

If there is a *variable* in *context*, it is named *var* for brevity. If it has a *latex_name* and *unit*, then these values will be used to label the x axis. For example, it could become *x [m]* or *E [keV]* on the plot. If no name or unit were provided, the plot will be rendered without a label, but also without an error or a crash.

Jinja allows very rich programming possibilities. Templates can set variables, use conditional operators and cycles. Refer to Jinja documentation[2] for details.

To use *Jinja* with LaTeX, Lena slightly changed its default syntax[3]: blocks and variables are enclosed in *\BLOCK* and *\VAR* environments respectively.

A *context* is a simple Python dictionary or its subclass. *Flow* in Lena consists of tuples of *(data, context)* pairs. It is usually not called *dataflow*, because it also has context. As it was shown earlier, context is not necessary for Lena

[2] Jinja documentation: https://jinja.palletsprojects.com/

[3] To use Jinja to render LaTeX was proposed here and here, template syntax was taken from the original article.

sequences. However, it greatly simplifies plot creation and provides complementary information with the main data. To add context to the flow, simply pass it with data as in the following example:

```
class ReadData():
    """Read data from CSV files."""

    def run(self, flow):
        """Read filenames from flow and yield vectors.

        If vector component could not be cast to float,
        *ValueError* is raised.
        """
        for filename in flow:
            with open(filename, "r") as fil:
                for line in fil:
                    vec = [float(coord)
                           for coord in line.split(',')]
                    # (data, context) pair
                    yield (vec, {"data": {"filename": filename}})
```

We read names of files from the incoming *flow* and yield coordinate vectors. We add file names to a nested dictionary "data" (or whatever we call it). *Filename* could be referred in the template as *data["filename"]* or simply *data.filename*.

Template rendering is widely used in a well developed area of web programming, and there is little difference between rendering an HTML page or a LaTeX file, or any other text file. Even though templates are powerful, good design suggests using their full powers only when necessary. The primary task of templates is to produce plots, while any nontrivial calculations should be contained in data itself (and provided through a context).

Context allows *separation of data and presentation* in Lena. This is considered a good programming practice, because it makes parts of a program focus on their primary tasks and avoids code repetition.

Since all data flow is passed inside sequences of the framework, context is also essential if one needs to pass some additional data to the following elements. Different elements update the context from flow with their own context, which persists unless it is deleted or changed.

1.1.2 A real analysis example

Now we are ready to do some real data processing. Let us read data from a file and make a histogram of *x* coordinates.

Note: The complete example with other files for this tutorial can be found in *docs/examples/tutorial* directory of the framework's tree or online.

Listing 1: main.py

```
from __future__ import print_function

import os

from lena.core import Sequence, Source
from lena.math import mesh
from lena.output import ToCSV, Writer, LaTeXToPDF, PDFToPNG
from lena.output import MakeFilename, RenderLaTeX
from lena.structures import Histogram

```

(continues on next page)

(continued from previous page)

```
from read_data import ReadData


def main():
    data_file = os.path.join("..", "data", "normal_3d.csv")
    s = Sequence(
        ReadData(),
        lambda dt: (dt[0][0], dt[1]),
        Histogram(mesh((-10, 10), 10)),
        ToCSV(),
        MakeFilename("x"),
        Writer("output"),
        RenderLaTeX("histogram_1d.tex"),
        Writer("output"),
        LaTeXToPDF(),
        PDFToPNG(),
    )
    results = s.run([data_file])
    print(list(results))

if __name__ == "__main__":
    main()
```

If we run the script, the resulting plots and intermediate files will be written to the directory *output/*, and the terminal output will be similar to this:

$ python main.py

pdflatex -halt-on-error -interaction batchmode -output-directory output output/x.tex

pdftoppm output/x.pdf output/x -png -singlefile

[('output/x.png', {'output': {'filetype': 'png'}, 'data': {'filename': '../data/normal_3d.csv'}, 'histogram': {'ranges': [(-10, 10)], 'dim': 1, 'nbins': [10]}})]

During the run, the element *LaTeXToPDF* called *pdflatex*, and *PDFToPNG* called *pdftoppm* program. The commands are printed with all arguments, so that if there was an error during LaTeX rendering, you can run this command manually until the rendered file *output/x.tex* is fixed (and then fix the template).

The last line of the output is the data and context, which are the results of the sequence run. The elements which produce files usually yield *(file path, context)* pairs. In this case there is one resulting value, which has a string *output/x.png* as its *data* part.

Let us return to the script to see the sequence in more details. The sequence *s* runs one data file (the list could easily contain more). Since our *ReadData* produces a *(data, context)* pair, the following lambda leaves the *context* part unchanged, and gets the zeroth index of each incoming vector (which is the zeroth part of the *(data, context)* pair).

This lambda is not very readable, and we'll see a better and more general approach in the next part of the tutorial. But it shows how the *flow* can be intercepted and transformed at any point within a sequence.

The resulting *x* components fill a *Histogram*, which is initialized with *edges* defined a *mesh* from *-10* to *10* with 10 bins.

This histogram, after it has been fed with the complete *flow*, is transformed to a *CSV* (comma separated values) text. In order for external programs (like *pdflatex*) to use the resulting table, it must be written to a file.

MakeFilename adds file name to *context["output"]* dictionary. *context.output.filename* is the file name without path and extension (the latter will be set by other elements depending on the format of data: first it is a *csv* table, then it may become a *pdf* plot, etc.) Since there is only one file expected, we can simply call it *x*.

Writer element writes text data to the file system. It is initialized with the name of the output directory. To be written, the context of a value must have an "output" subdictionary.

After we have produced the *csv* table, we can render our LaTeX template *histogram_1d.tex* with that table and *context*, and convert the plot to *pdf* and *png*. As earlier, *RenderLaTeX* produces text, which must be written to the file system before used.

Congratulations: now you can do a complete analysis using the framework, from the beginning to the final plots. In the end of this part of the tutorial we'll show several Lena elements which may be useful during development.

1.1.3 Elements for development

Let us use the structure of the previous analysis and add some more elements to the sequence:

```
from lena.context import Context
from lena.flow import Cache, End, Print

s = Sequence(
    Print(),
    ReadData(),
    # Print(),
    ISlice(1000),
    lambda val: val[0][0], # data.x
    Histogram(mesh((-10, 10), 10)),
    Context(),
    Cache("x_hist.pkl"),
    # End(),
    ToCSV(),
    # ...
)
```

Print outputs values, which pass through it in the *flow*. If we suspect an error or want to see exactly what is happening at a given point, we can put any number of *Print* elements anywhere we want. We don't need to search for other files and add print statements there to see the input and output values.

ISlice, which we met earlier when approximating *pi*, limits the flow to the specified number of items. If we are not sure that our analysis is already correct, we can select only a small amount of data to test that.

Context is an element, which is a subclass of *dictionary*, and it can be used as a context when a formatted output is needed. If a *Context* object is inside a sequence, it transforms the *context* part of the flow to its class, which is indented during output (not in one line, as a usual dict). This may help during manual analysis of many nested contexts.

Cache stores the incoming flow or loads it from file. Its initialization argument is the file name to store the flow. If the file is missing, then *Cache* creates that, runs the previous elements, and stores values from the flow into the file. On subsequent runs it loads the flow from file, and no previous elements are run. *Cache* uses *pickle*, which allows serialization and deserialization of most Python objects (except function's code). If you have some lengthy calculation and want to save the results (for example, to improve plots, which follow in the sequence), you can use *Cache*. If you changed the algorithm before *Cache*, simply delete the file to refill that with the new flow.

End runs all previous elements and stops analysis here. If we enabled that in this example, *Cache* would be filled or read (as without the *End* element), but nothing would be passed to *ToCSV* and further. One can use *End* if they know for sure, that the following analysis is incomplete and will fail.

Summary

Lena encourages to split analysis into small independent *elements*, which are joined into *sequences*. This allows to substitute, add or remove any element or transform the *flow* at any place, which may be very useful for development. Sequences can be elements of other sequences, which allows their *reuse*.

Elements can be callables or *generators*. Simple callables can be easily added to transform each value from the *flow*, while generators can transform the *flow*, adding more values or reducing that. Generators allow lazy evaluation, which benefits memory impact and generalizes algorithms to use potentially many values instead of one.

Complete information about the analysis is provided through the *context*. It is the user's responsibility to add the needed context and to write templates for plots. The user must also provide some initial context for naming files and plots, but apart from that the framework transfers and updates context itself.

We introduced two basic sequences. A *Sequence* can be placed before, after or inside another *Sequence*. A *Source* is similar to a *Sequence*, but no other sequence can precede that.

Table 1: Sequences

Sequence	Initialization	Usage
Sequence	Elements with a *__call__(value)* or *run(flow)* method (or callables)	s.run(*flow*)
Source	The first element has a *__call__()* method (or is callable), others form a *Sequence*	s()

In this part of the tutorial we have learnt how to make a simple analysis of data read from a file and how to produce several plots using only one template. In the next part we'll learn about new types of elements and sequences and how to make several analyses reading a data file only once.

Exercises

1. Ivan wants to become more familiar with generators and implements an element *End*. He writes this class:

```
class End(object):
    """Stop sequence here."""

    def run(self, flow):
        """Exhaust all preceding flow and stop iteration."""
        for val in flow:
            pass
        raise StopIteration()
```

and adds this element to *main.py* example above. When he runs the program, he gets

```
Traceback (most recent call last):
    File "main.py", line 46, in <module>
        main()
    File "main.py", line 42, in main
        results = s.run([data_file])
    File "lena/core/sequence.py", line 70, in run
        flow = elem.run(flow)
    File "main.py", line 24, in run
        raise StopIteration()
StopIteration
```

It seems that no further elements were executed, indeed. However, Ivan recalls that *StopIteration* inside a generator should lead to a normal exit and should not be an error. What was done wrong?

2. Svetlana wants to make sure that no statement is really executed during a generator call. Write a simple generator to check that.
3. *Count* counts values passing through that. In order for that not to change the data flow, it should add results to the context. What other design decisions should be considered? Write its simple implementation and check that it works as a sequence element.
4. Lev doesn't like how the output in previous examples is organised.

   "In our object-oriented days, I could use only one object to make the whole analysis", - he says. "Histogram to CSV, Write, Render, Write again,. . . : if our output system remains the same, and we need to repeat that in every script, this is a code bloat".

   How to make only one element for the whole output process? What are advantages and disadvantages of these two approaches?

5. ** Remember the implementation of *Sum* earlier. Suppose you need to split one flow into two to make two analyses, so that you don't have to read the flow several times or store it completely in memory.

   Will this *Sum* allow that, why? How should it be changed? These questions will be answered in the following part of the tutorial.

The answers to the excercises are given in the end of the tutorial.

1.2 Split

In this part of the tutorial we'll learn how to make several analyses reading input data only once and without storing that in memory.

Contents

- *Introduction*
- *Variables*
  - *Combine*
  - *Compose*
- *Analysis example*
- *Adapters, elements and sequences*
- *Split*
- *Context. Performance and safety*

1.2.1 Introduction

If we want to process same data flow "simultaneously" by *sequence1* and *sequence2*, we use the element *Split*:

```
from lena.core import Split

s = Sequence(
    ReadData(),
    Split([
        sequence1,
        sequence2,
        # ...
    ]),
    ToCSV(),
    # ...
)
```

The first argument of *Split* is a list of sequences, which are applied to the incoming flow "in parallel" (not in the sense of processes or threads).

However, not every sequence can be used in parallel with others. Recall the example of an element *Sum* from the first part of the tutorial:

```
class Sum1():
    def run(self, flow):
        s = 0
        for val in flow:
            s += val
        yield s
```

The problem is that if we pass it a *flow*, it will consume it completely. After we call *Sum1().run(flow)*, there is no way to stop iteration in the inner cycle and resume that later. To reiterate the *flow* in another sequence we would have to store that in memory or reread all data once again.

To run analyses in parallel, we need another type of element. Here is *Sum* refactored:

```
class Sum():
    def __init__(self):
        self._sum = 0

    def fill(self, val):
        self._sum += val

    def compute(self):
        yield self._sum
```

This *Sum* has methods *fill(value)* and *compute()*. *Fill* is called by some external code (for example, by *Split*). After there is nothing more to fill, the results can be generated by *compute*. The method name *fill* makes its class similar to a histogram. *Compute* in this example is trivial, but it may include some larger computations. We call an element with methods *fill* and *compute* a *FillCompute* element. An element with a *run* method can be called a *Run* element.

A *FillCompute* element can be generalized. We can place before that simple functions, which will transform values before they fill the element. We can also add other elements after *FillCompute*. Since *compute* is a generator, these elements can be either simple functions or *Run* elements. A sequence with a *FillCompute* element is called a *FillComputeSeq*.

Here is a working example:

Listing 2: tutorial/2_split/main1.py

```
data_file = os.path.join("..", "data", "normal_3d.csv")
s = Sequence(
```

(continues on next page)

(continued from previous page)

```
    ReadData(),
    Split([
        (
            lambda vec: vec[0],
            Histogram(mesh((-10, 10), 10)),
            ToCSV(),
            Writer("output", "x"),
        ),
        (
            lambda vec: vec[1],
            Histogram(mesh((-10, 10), 10)),
            ToCSV(),
            Writer("output", "y"),
        ),
    ]),
    RenderLaTeX("histogram_1d.tex", "templates"),
    Writer("output"),
    LaTeXToPDF(),
    PDFToPNG(),
)
results = s.run([data_file])
for res in results:
    print(res)
```

Lena Histogram is a FillCompute element. The elements of the list in *Split* (tuples in this example) during the initialization of *Split* are transformed into FillCompute sequences. The *lambdas* select parts of vectors, which will fill the corresponding histogram. After the histogram is filled, it is given appropriate name by *Writer* (so that they could be distinguished in the following flow).

Writer has two initialization parameters: the default directory and the default file name. Writer only writes strings (and *unicode* in Python 2). Its corresponding context is called *output* (as its module). If *output* is missing in the context, values pass unchanged. Otherwise, file name and extension are searched in *context.output*. If *output.filename* or *output.fileext* are missing, then the default file name or "txt" are used. The default file name should be used only when you are sure that only one file is going to be written, otherwise it will be rewritten every time. The defaults *Writer*'s parameters are empty string (current directory) and "output" (resulting in *output.txt*).

ToCSV yields a string and sets *context.output.fileext* to *"csv"*. In the example above Writer objects write CSV data to *output/x.csv* and *output/y.csv*.

For each file written, *Writer* yields a tuple *(file path, context)*, where *context.output.filepath* is updated with the path to file.

After the histograms are filled and written, *Split* yields them into the following flow in turn. The containing sequence *s* doesn't distinguish *Split* from other elements, because *Split* acts as any *Run* element.

1.2.2 Variables

One of the basic principles in programming is "don't repeat yourself" (*DRY*).

In the example above, we wanted to give distinct names to histograms in different analysis branches, and used two *writers* to do that. However, we can move *ToCSV* and *Writer* outside the *Split* (and make our code one line shorter):

Listing 3: tutorial/2_split/main2.py

```
from lena.output import MakeFilename
s = Sequence(
```

(continues on next page)

(continued from previous page)

```
    ReadData(),
    Split([
        (
            lambda vec: vec[0],
            Histogram(mesh((-10, 10), 10)),
            MakeFilename("x"),
        ),
        (
            lambda vec: vec[1],
            Histogram(mesh((-10, 10), 10)),
            MakeFilename("y"),
        ),
    ]),
    ToCSV(),
    Writer("output"),
    # ... as earlier ...
)
```

Element *MakeFilename* adds file name to *context.output*. *Writer* doesn't need a default file name anymore. Now it writes two different files, because *context.output.filename* is different.

The code that we've written now is very explicit and flexible. We clearly see each step of the analysis and it as a whole. We control output names and we can change the logic as we wish by adding another element or *lambda*. The structure of our analysis is very transparent, but the code is not beautiful enough.

Lambdas don't improve readability. Indices *0* and *1* look like magic constants. They are connected to names *x* and *y* in the following flow, but let us unite them in one element (and improve the *cohesion* of our code):

Listing 4: tutorial/2_split/main3.py

```
from lena.variables import Variable

def main():
    data_file = os.path.join("..", "data", "normal_3d.csv")
    writer = Writer("output")
    s = Sequence(
        ReadData(),
        Split([
            (
                Variable("x", lambda vec: vec[0]),
                Histogram(mesh((-10, 10), 10)),
            ),
            (
                Variable("y", lambda vec: vec[1]),
                Histogram(mesh((-10, 10), 10)),
            ),
            (
                Variable("z", lambda vec: vec[2]),
                Histogram(mesh((-10, 10), 10)),
            ),
        ]),
        MakeFilename("{variable.name}"),
        ToCSV(),
        writer,
        RenderLaTeX("histogram_1d.tex", "templates"),
        writer,
        LaTeXToPDF(),
```

(continues on next page)

(continued from previous page)

```
        PDFToPNG(),
    )
    results = s.run([data_file])
    for res in results:
        print(res)
```

A *Variable* is essentially a function with a name. It transforms data and adds its own name to *context.variable.name*.

In this example we initialize a variable with a name and a function. It can accept arbitrary keyword arguments, which will be added to its context. For example, if our data is a series of *(positron, neutron)* events, then we can make a variable to select the second event:

```
neutron = Variable(
   "neutron", lambda double_ev: double_ev[1],
   latex_name="n", type="particle"
)
```

In this case *context.variable* will be updated not only with *name*, but also *latex_name* and *type*. In code their values can be got as variable's attributes (e.g. *neutron.latex_name*). Variable's function can be initialized with the keyword *getter* and is available as a method *getter*.

MakeFilename accepts not only constant, but also format strings, which take arguments from context. In our example, *MakeFilename("{variable.name}")* creates file name from *context.variable.name*.

Note also that since two *Writers* do the same thing, we rewrote them as one object.

Combine

Variables can be joined into a multidimensional variable using *Combine*.

Combine(var1, var2, . . .) applied to a *value* is a tuple *((var1.getter(value), var2.getter(value), . . .), context)*. The first element of the tuple is *value* transformed by each of the composed variables. *Variable.getter* is a function that returns only data without context.

Combine is a subclass of a *Variable*, and it accepts arbitrary keywords during initialization. All positional arguments must be *Variables*. Name of the combined variable can be passed as a keyword argument. If not provided, it is its variables' names joined with '_'.

The resulting context is that of a usual *Variable* updated with *context.variable.combine*, where *combine* is a tuple of each variable's context.

Combine has an attribute *dim*, which is the number of its variables. A constituting variable can be accessed using its index. For example, if *cv* is *Combine(var1, var2)*, then *cv.dim* is 2, *cv.name* is *var1.name_var2.name*, and *cv[1]* is *var2*.

Combine variables are used for multidimensional plots.

Compose

When we put several variables or functions into a sequence, we obtain their composition. In the Lena framework we want to preserve as much context as possible. If some previous element was a *Variable*, its context is moved into *variable.compose* subcontext.

Function composition can be also defined as *variables.Compose*.

In this example we first select the *neutron* part of the data, and then the *x* coordinate:

```
>>> from lena.variables import Variable, Compose
>>> # data is pairs of (positron, neutron) coordinates
>>> data = [((1.05, 0.98, 0.8), (1.1, 1.1, 1.3))]
>>> x = Variable(
...     "x", lambda coord: coord[0], type="coordinate"
... )
>>> neutron = Variable(
...     "neutron", latex_name="n",
...     getter=lambda double_ev: double_ev[1], type="particle"
... )
>>> x_n = Compose(neutron, x)
>>> x_n(data[0])[0] # data
1.1
```

Data part of the result, as expected, is the composition of variables *neutron* and *x*. Same result could be obtained as a sequence of variables: *Sequence(neutron, x).run(data)*, but the context of *Compose* is created differently.

The name of the composed variable is names of its variables (from left to right) joined with underscore. If there are two variables, LaTeX name will be also created from their names (or LaTeX names, if present) as a subscript in reverse order. In our example the context will be this:

```
>>> x_n(data[0])[1]
{
    'variable': {
        'name': 'neutron_x', 'particle': 'neutron',
        'latex_name': 'x_{n}', 'coordinate': 'x', 'type': 'coordinate',
        'compose': {
            'type': 'particle', 'latex_name': 'n',
            'name': 'neutron', 'particle': 'neutron'
        },
    }
}
```

Context of the composed variable is updated with a *compose* subcontext, which makes it similar to the context produced by variables in a sequence.

As for any variable, *name* or other parameters can be passed as keyword arguments during initialization.

Keyword *type* has a special meaning. If present, then during initialization of a variable its context is updated with *{variable.type: variable.name}* pair. During variable composition (in *Compose* or by subsequent application to the *flow*) *context.variable* is updated with new variable's context, but if its type is different, it will persist. This allows access to *context.variable.particle* even if it was later composed with other variables.

1.2.3 Analysis example

Let us combine what we've learnt before and use it in a real analysis. An important change would be that if we create 2-dimensional plots, we add another template for that. Below is a small example. All template commands were explained in the first part of the tutorial.

Listing 5: tutorial/2_split/templates/histogram_2d.tex

```
\documentclass{standalone}
\usepackage{tikz}
\usepackage{pgfplots}
\usepgfplotslibrary{colorbrewer}
\pgfplotsset{compat=1.15}
```

(continues on next page)

(continued from previous page)

```
\BLOCK{ set varx = variable.combine[0] }
\BLOCK{ set vary = variable.combine[1] }

\begin{document}
\begin{tikzpicture}
    \begin{axis}[
        view={0}{90},
        grid=both,
        \BLOCK{ set xcols = histogram.nbins[0]|int + 1 }
        \BLOCK{ set ycols = histogram.nbins[1]|int + 1 }
        mesh/cols=\VAR{xcols},
        mesh/rows=\VAR{ycols},
        colorbar horizontal,
        xlabel = {$\VAR{ varx.latex_name }$
            \BLOCK{ if varx.unit }[$\mathrm{\VAR{ varx.unit }}$]\BLOCK{ endif }},
        ylabel = {$\VAR{ vary.latex_name }$
            \BLOCK{ if vary.unit }[$\mathrm{\VAR{ vary.unit }}$]\BLOCK{ endif }},
    ]
    \addplot3 [
        surf,
        mesh/ordering=y varies,
    ] table [col sep=comma, header=false] {\VAR{ output.filepath }};
    \end{axis}
\end{tikzpicture}
\end{document}
```

If an axis has a *unit*, it will be added to its label (like *x [cm]*).

RenderLaTeX accepts a function as the first initialization argument or as a keyword *select_template*. That function must accept a value (presumably a *(data, context)* pair) from the flow, and return a template file name (to be found inside *template_path*).

Listing 6: tutorial/2_split/main4.py

```
from __future__ import print_function

import os

import lena.context
import lena.flow
from lena.core import Sequence, Split, Source
from lena.structures import Histogram
from lena.math import mesh
from lena.output import ToCSV, Writer, LaTeXToPDF, PDFToPNG
from lena.output import MakeFilename, RenderLaTeX
from lena.variables import Variable, Compose, Combine

from read_data import ReadDoubleEvents


positron = Variable(
    "positron", latex_name="e^+",
    getter=lambda double_ev: double_ev[0], type="particle"
)
neutron = Variable(
    "neutron", latex_name="n",
```

(continues on next page)

(continued from previous page)

```
    getter=lambda double_ev: double_ev[1], type="particle"
)
x = Variable("x", lambda vec: vec[0], latex_name="x", unit="cm", type="coordinate")
y = Variable("y", lambda vec: vec[1], latex_name="y", unit="cm", type="coordinate")
z = Variable("z", lambda vec: vec[2], latex_name="z", unit="cm", type="coordinate")

coordinates_1d = [
    (
        coordinate,
        Histogram(mesh((-10, 10), 10)),
    )
    for coordinate in [
        Compose(particle, coord)
            for coord in x, y, z
            for particle in positron, neutron
    ]
]


def select_template(val):
    data, context = lena.flow.get_data_context(val)
    if lena.context.get_recursively(context, "histogram.dim", None) == 2:
        return "histogram_2d.tex"
    else:
        return "histogram_1d.tex"


def main():
    data_file = os.path.join("..", "data", "double_ev.csv")
    writer = Writer("output")
    s = Sequence(
        ReadDoubleEvents(),
        Split(
            coordinates_1d
            +
            [(
                particle,
                Combine(x, y, name="xy"),
                Histogram(mesh(((-10, 10), (-10, 10)), (10, 10))),
                MakeFilename("{variable.particle}/{variable.name}"),
             )
             for particle in positron, neutron
            ]
        ),
        MakeFilename("{variable.particle}/{variable.coordinate}"),
        ToCSV(),
        writer,
        RenderLaTeX(select_template, template_path="templates"),
        writer,
        LaTeXToPDF(),
        PDFToPNG(),
    )
    results = s.run([data_file])
    for res in results:
        print(res)
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    main()
```

We import *ReadDoubleEvents* from a separate file. That class is practically the same as earlier, but it yields pairs of events instead of one by one.

We define *coordinates_1d* as a simple list of coordinates' composition. Note that we could make all combinations directly using the language. We could also do that in *Split*, but if we use all these coordinates together in different analyses or don't want to clutter the algorithm code, we can separate them.

In our new function *select_template* we use *lena.context.get_recursively*. This function is needed because we often have nested dictionaries, and Python's *dict.get* method doesn't recurse. We provide the default return value `None`, so that it doesn't raise an exception in case of a missing key.

In the *Split* element we fill histograms for 1- and 2-dimensional plots in one run. There are two *MakeFilename* elements, but *MakeFilename* doesn't overwrite file names set previously.

We created our first 2-dimensional histogram using *lena.math.mesh*. It accepts parameters *ranges* and *nbins*. In a multidimensional case these parameters are tuples of ranges and number of bins in corresponding dimensions, as in *mesh(((-10, 10), (-10, 10)), (10, 10))*.

After we run this script, we obtain two subdirectories in *output* for *positron* and *neutron*, each containing 4 plots (both *pdf* and *png*); in total 8 plots with proper names, units, axes labels, etc. It is straightforward to add other plots if we want, or to disable some of them in *Split* by commenting them out. The variables that we defined at the top level could be reused in other modules or moved to a separate module.

Note the overall design of our algorithm. We prepare all necessary data in *ReadDoubleEvents*. After that, *Split* uses different parts of these double events to create different plots. All important parameters should be contained in data itself. These allows a separation of data from presentation.

The knowledge we'll learn by the end of this chapter will be sufficient for most of practical analyses. Following sections give more details about Lena elements and usage.

1.2.4 Adapters, elements and sequences

Objects don't need to inherit from *Lena* classes to be used in the framework. Instead, they have to implement methods with specified names (like *run*, *fill*, etc). This is called structural subtyping in Python[1].

The specified method names can be changed using *adapters*. For example, if we have a legacy class

```
class MyEl():
    def my_run(self, flow):
        for val in flow:
            yield val
```

then we can create a *Run* element from a *MyEl* object with the adapter *Run*:

```
>>> from lena.core import Run
>>> my_run = Run(MyEl(), run="my_run")
>>> list(my_run.run([1, 2, 3]))
[1, 2, 3]
```

The adapter receives method name as a keyword argument. After it is created, it can be called with a method named *run* or inserted into a *Lena* sequence.

Similarly, a *FillCompute* adapter accepts names for methods *fill* and *compute*:

[1] PEP 544 – Protocols: Structural subtyping (static duck typing): https://www.python.org/dev/peps/pep-0544

```
FillCompute(el, fill='fill', compute='compute')
```

If callable methods *fill* and *compute* were not found in *el*, *LenaTypeError* is raised.

What other types of elements are possible in data analysis? A common algorithm in physics is event selection. We analyse a large set of data looking for specific events. These events can be missing there or contained in a large quantity. To deal with this, we have to be prepared not to consume all flow (as a *Run* element does) and not to store all flow in the element before that is yielded. We create an element with a *fill* method, and call the second method *request*. A *FillRequest* element is similar to *FillCompute*, but *request* can be called multiple times. As with *FillComputeSeq*, we can add *Call* elements (lambdas) before a *FillRequest* element and *Call* or *Run* elements after that to create a sequence *FillRequestSeq*.

Elements can be transformed one into another. During initialization a *Sequence* checks for each its argument whether it has a *run* method. If it is missing, it tries to convert the element to a *Run* element using the adapter.

Run can be initialized from a *Call* or a *FillCompute* element. A callable is run as a transformation function, which accepts single values from the flow and returns their transformations for each value:

```
for val in flow:
    yield self._el(val)
```

A *FillCompute* element is run the following way: first, *fill(value)* is called for the whole flow. After the flow is exhausted, *compute()* is called.

There are algorithms and structures which are inherently not memory safe. For example, *lena.structures.Graph* stores all filled data as its points, and it is a *FillRequest* element. Since *FillRequest* can't be used directly in a *Sequence*, or if we want to yield only the final result once, we cast that with *FillCompute(Graph())*. We can do that when we are sure that our data won't overflow memory, and that cast will be explicit in our code.

To sum up, adapters in Lena can be used for several purposes:

- provide a different name for a method (*Run(my_obj, run="my_run")*),
- hide unused methods to prevent ambiguity (if an element has many methods, we can wrap that in an adapter to expose only the needed ones),
- automatically convert objects of one type to another in sequences (*FillCompute* to *Run*),
- explicitly cast object of one type to another (*FillRequest* to *FillCompute*).

1.2.5 Split

In the examples above, *Split* contained several *FillComputeSeq* sequences. However, it can be used with all other sequences we know.

Split has a keyword initialization argument *bufsize*, which is the size of the buffer for the input flow.

During *Split.run(flow)*, the *flow* is divided into subslices of *bufsize*. Each subslice is processed by sequences in the order of their initializer list (the first positional argument in *Split.__init__*).

If a sequence is a *Source*, it doesn't accept the incoming *flow*, but produces its own complete flow and becomes inactive (is not called any more).

A *FillRequestSeq* is filled with the buffer contents. After the buffer is finished, it yields all values from *request()*.

A *FillComputeSeq* is filled with values from each buffer, but yields values from *compute* only after the whole *flow* is finished.

A *Sequence* is called with *run(buffer)* instead of the whole flow. The results are yielded for each buffer. If the whole flow must be analysed at once, don't use such a sequence in *Split*.

If the *flow* was empty, each *__call__* (from *Source*), *compute*, *request* or *run* is called nevertheless.

Source within *Split* can be used to add new data to *flow*. For example, we can create *Split([source, ()])*, and in this place of a sequence first all data from *source* will be generated, then all data from preceding elements will be passed (empty *Sequence* passes values unchanged). This can be used to provide several flows to a further element (like data, Monte Carlo and analytical approximation).

Split acts both as a sequence (because it contains sequences) and as an element. If all its elements (sequences, to be precise) have the same type, *Split* will have methods of this type. For example, if *Split* has only *FillComputeSeq* inside, it will create methods *fill* and *compute*. During *fill* all its sequences will be filled. During *compute* their results will be yielded in turn (all results from the first sequence, then from the second, etc). *Split* with *Source* sequences will act as a *Source*. Of course, *Split* can be used within a *Split*.

1.2.6 Context. Performance and safety

Dictionaries in Python are *mutable*, that is their content can change. If an element stores the current context, that may be changed by some other element. The simplest example: if your original data has context, it will be changed after being processed by a sequence.

This is how a typical *Run* element deals with context. To be most useful, it must be prepared to accept data with and without context:

```
class RunEl():
    def __init__(self):
        self._context = {"subcontext": "el"}

    def run(self, flow):
        for val in flow:
            data, context = lena.flow.get_data_context(val)
            # ... do something ...
            lena.flow.update_recursively(context, self._context)
            yield (new_data, context)
```

lena.flow.get_data_context(value) splits *value* into a pair of (data, context). If *value* contained only data without context, the *context* part will be an empty dictionary (therefore it is safe to use *get_data_context* with any *value*). If only one part is needed, *lena.flow.get_data* or *lena.flow.get_context* can be used.

If *subcontext* can contain other elements except *el*, then to preserve them we call not *context.update*, but *lena.flow.update_recursively*. This function doesn't overwrite subdictionaries, but only conflicting keys within them. In this case *context.subcontext* key will always be set to *el*, but if *self._context.subcontext* were a dictionary *{"el": "el1"}*, then all *context.subcontext* keys (if present) except *el* would remain.

Usually elements in a *Sequence* yield computed data and context, and never use or change that again. In *Split*, however, several sequences use the same data simultaneously. This is why *Split* makes a deep copy of the incoming flow in its buffer. A deep copy of a context is completely independent of the original or its other copies. However, to copy an entire dictionary requires some computational cost.

Split can be initialized with a keyword argument *copy_buf*. By default it is `True`, but can be set to `False` to disable deep copy of the flow. This may be a bit faster, but do it only if you are absolutely sure that your analysis will remain correct.

There are several things in Lena that help against context interference:

- elements change their own context (*Writer* changes *context.output* and not *context.variable*),
- if *Split* has several sequences, it makes a deep copy of the flow before feeding that to them,
- *FillCompute* and *FillRequest* elements make a deep copy of context before yielding[3].

[3] For framework elements this is obligatory, for user code this is recommended.

This is how a *FillCompute* element is usually organised in Lena:

```
class MyFillComputeEl():
    def __init__(self):
        self._val = 0
        self._context = {"subcontext": "el"}
        self._cur_context = {}

    def fill(self, val):
        data, context = lena.flow.get_data_context(val)
        self._val += data
        self._cur_context = context

    def compute(self):
        context = copy.deepcopy(self._cur_context)
        # or copy.deepcopy(self._context):
        lena.flow.update_recursively(context, self._context)
        yield (self._val, context)
```

During *fill* the last context is saved. During *compute* a deep copy of that is made (since *compute* is called only once, this can be done without performance loss), and it is updated with *self._context*.

Performance is not the highest priority in Lena, but it is always nice to have. When possible, optimizations are made. Performance measurements show that *deepcopy* can take most time in Lena analysis[2]. A linear *Sequence* or *Run* elements don't do a deep copy of data. If *Split* contains several sequences, it doesn't do a deep copy of the flow for the last sequence. It is possible to circumvent all copying of data in *Split* to gain more performance at the cost of more precautions and more streamlined code.

Summary

Several analyses can be performed on one flow using an element *Split*. It accepts a list of sequences as its first initialization argument.

Since *Split* divides the flow into buffered slices, elements must be prepared for that. In this part of the tutorial we introduced the *FillCompute* and the *FillRequest* elements. The former yields the results when its *compute* method is called. It is supposed that *FillCompute* is run only once and that it is memory safe (that it reduces data). If an element can consume much memory, it must be a *FillRequest* element.

If we add *Call* elements before and *Run* and *Call* elements after our *FillCompute* or *FillRequest* elements, we can generalize them to sequences *FillComputeSeq* and *FillRequestSeq*. They are created implicitly during *Split* initialization.

Variables connect functions with context. They have names and can have LaTeX names, units and other parameters, which helps to create plots and write output files. *Compose* corresponds to function composition, while *Combine* creates multidimensional variables for multidimensional plots.

If an element has methods with unusual names, adapters can be used to relate them to the framework names. Adapters are also used to explicitly cast one type of element to another or to implicitly convert an element to an appropriate type during a sequence initialization.

[2] One can use *tutorial/2_split/performance.py* to make a quick analysis. To create 3 histograms (like in *main4.py* example above) for one million generated events it took 82 seconds in Python 2 on a laptop. The longest total time was spent for *copy.deepcopy* (20 seconds). For Python 3, PyPy and PyPy 3 the total time was 71, 23 and 16 seconds. These numbers are approximate (the second measurement for PyPy gave 19 seconds). If we change *Variables* into *lambdas*, add *MakeFilename* after *Histogram* and set *copy_buf=False* in *Split*, the total time will be 18 seconds for Python 2 and 4 seconds for PyPy 3.

This difference may be not important in practice: for example, the author usually deals with data sets of several tens of thousands events, and a large amount of time is spent to create 2-dimensional plots with *pdflatex*.

To be most useful, elements should be prepared to accept values consisting of only data or data with context. To work safely with a mutable context, a deep copy of that must be made in *compute* or *request*. On the other hand, unnecessary deep copies (in *run*, *fill* or *__call__*) may slightly decrease the performance. Lena allows optimizations if they are needed.

Exercises

1. Extend the *Sum* example in this chapter so that it could handle context. Check that it works.
2. In the analysis example *main4.py* there are two *MakeFilename* elements. Is it possible to use only one of them? How?
3. We developed the example *main2.py* and joined *lambda* and *filename* into a *Variable*. We could also add a name to the *Histogram*. Which design decision would be better?
4. What are the consequences of calling *compute* even for an empty flow?
5. Alexander writes a diploma thesis involving some data analysis and wants to choose a framework for that. He asks colleagues and professors, and stops at three possible options. One library is easy to use and straight to the point, and is sufficient for most diploma theses. Another library is very rich and used by seasoned professionals, and its full power surpasses even its documentation. The third framework doesn't provide a plenty of mathematical functions, but promises structured and beautiful code. Which one would you advise?

1.3 Answers to exercises

1.3.1 Part 1

Ex. 1

End.run in this case is not a generator. To make it a generator, add a *yield* statement somewhere. Also note that since Python 3.7 all *StopIteration* are considered to be errors according to PEP 479. Use a simple *return* instead. This is the implementation in *lena.flow*:

```
class End(object):
    """Stop sequence here."""

    def run(self, flow):
        """Exhaust all preceding flow and stop iteration
        (yield nothing to the following flow).
        """
        for val in flow:
            pass
        return
        # otherwise it won't be a generator
        yield "unreachable"
```

Ex. 2

```
>>> def my_generator():
...     print("enter my generator")
...     yield True
...
```

(continues on next page)

(continued from previous page)

```
>>> results = my_generator()
>>> list(results)
enter my generator
[True]
```

Ex. 3

An implementation of *Count* is given below. An important consideration is that there may be several *Counts* in the sequence, so give them different names to distinguish.

```
class Count(object):
    """Count items that pass through.

    After the flow is exhausted, add {*name*: count} to the *context*.
    """

    def __init__(self, name="counter"):
        """*name* is this counter's name."""
        self._name = name
        self._count = 0
        self._cur_context = {}

    def run(self, flow):
        """Yield incoming values and increase counter.

        When the incoming flow is exhausted,
        update last value's context with *(count, context)*.

        If the flow was empty, nothing is yielded
        (so *count* can't be zero).
        """
        try:
            prev_val = next(flow)
        except StopIteration:
            # otherwise it will be an error since PEP 479
            # https://stackoverflow.com/a/51701040/952234
            return
            # raise StopIteration
        count = 1
        for val in flow:
            yield prev_val
            count += 1
            prev_val = val
        val = prev_val
        data, context = lena.flow.get_data(val), lena.flow.get_context(val)
        context.update({self._name: count})
        yield (data, context)
```

Ex. 4

A simple output function could be the following:

```
def output(output_dir="output"):
    writer = lena.output.Writer(output_dir)
```

(continues on next page)

(continued from previous page)

```
    s = lena.core.Sequence(
        lena.output.ToCSV(),
        writer,
        lena.context.Context(),
        lena.output.RenderLaTeX(), # initialize properly here
        writer,
        lena.output.LaTeXToPDF(),
        lena.output.PDFToPNG(),
    )
    return s
```

Then place *output()* in a sequence, and new initialized elements will be put there.

This approach is terse, but less flexible and explicit. In practice verbosity of several output elements was never a problem for the author.

Ex. 5

It is probably impossible in Python to stop a function and resume it at the given point. Inform the author if you know how to do that.

1.3.2 Part 2

Ex. 1

This is the *Sum* implementation from *lena.math*:

```
class Sum(object):
    """Calculate sum of input values."""

    def __init__(self, start=0):
        """*start* is the initial value of sum."""
        # start is similar to Python's builtin *sum* start.
        self._start = start
        self.reset()

    def fill(self, value):
        """Fill *self* with *value*.

        The *value* can be a *(data, context)* pair.
        The last *context* value (considered empty if missing)
        sets the current context.
        """
        data, context = lena.flow.get_data_context(value)
        self._sum += data
        self._cur_context = context

    def compute(self):
        """Calculate the sum and yield.

        If the current context is not empty, yield *(sum, context)*.
        Otherwise yield only *sum*.
        """
        if not self._cur_context:
```

(continues on next page)

(continued from previous page)

```
            yield self._sum
        else:
            yield (self._sum, copy.deepcopy(self._cur_context))

    def reset(self):
        """Reset sum and context.

        Sum is reset to the *start* value and context to {}.
        """
        self._sum = copy.deepcopy(self._start)
        self._cur_context = {}
```

Ex. 2

Delete the first *MakeFilename* and change the second one to

```
MakeFilename("{variable.particle}/{variable.name}")
```

Ex. 3

We believe that the essence of data is captured in the function with which it was obtained. Histogram is just its presentation. It may be tempting to name a histogram just for convenience, but a general *MakeFilename* would be more powerful.

Functional programming suggests that larger functions should be decomposed into smaller ones, while object-oriented design praises code cohesion. The decisions above were made by choosing between these principles. There are cases when a histogram is data itself. In such situations, however, the final result is often not a histogram but a function of that, like a mean or a mode (which again suggests a different name).

Ex. 4

In part 1 of the tutorial there was introduced an element *End*, which stops the flow at its location. However, if there are *Histograms* in the following flow, they will be yielded even if nothing was filled into them. Empty histogram is a legitimate histogram state. It may be also filled, but the result may fall out of the histogram's range. It is possible to write a special element if needed to check whether the flow was empty.

In the next chapter we will present a specific analysis during which a histogram may not be filled, but it must be produced. A *FillCompute* element is more general than a histogram (which we use here just for a concrete example).

Note also that if a histogram was not filled, preceding variables weren't called. The histogram will have no context, probably won't have a name and won't be plotted correctly. Take an empty flow into account when creating your own *FillCompute* elements.

Ex. 5

It depends on the student's priorities. If he wants to finish the diploma never to return to programming, or if he has a lot of work to do apart from writing code, the fastest option might be the best. General algorithms have a more complicated interface. However, if one decides to rely upon a "friendly" library, there is a risk that the programmer will have to rewrite all code when more functionality becomes needed.

Architectural choices rise for middle-sized or large projects. If the student's personal code becomes large and more time is spent on supporting and extending that, it may be a good time to define the architecture. Here the author estimates "large" programs to start from one thousand lines.

Another distinction is that when using a library one learns how to use a library. When using a good framework, one learns how to write good code. Many algorithms in programming are simple, but to choose a good design may be much more difficult, and to learn how to create good programs yourself may take years of studying and experience. When you feel difficulties with making programming decisions, it's time to invest into design skills.

CHAPTER 2

Reference

2.1 Context

Context:

Context([d, formatter])	Dictionary with easy-to-read formatting.

Functions:

difference(d1, d2)	Return a dictionary with items from *d1* not contained in *d2*.
get_recursively(d, keys[, default])	Get value from a dictionary *d* recursively.
intersection(*dicts, **kwargs)	Return a dictionary, such that each of its items are contained in all *dicts* (recursively).
str_to_dict(s)	Create a dictionary from a dot-separated string *s*.
update_nested(d, other)	Update dictionary *d* with items from *other* dictionary.
update_recursively(d, other)	Update dictionary *d* with items from *other* dictionary.

2.1.1 Context

Make better output for context. Example:

```
>>> from lena.context import Context
>>> c = Context({"1": 1, "2": {"3": 4}})
>>> print(c) # doctest: +NORMALIZE_WHITESPACE
{
    "1": 1,
    "2": {
        "3": 4
    }
}
```

class Context (*d={}*, *formatter=None*)

Bases: `dict`

Dictionary with easy-to-read formatting.

Initialize from a dictionary *d*.

Representation is defined by the *formatter*. That must be a callable, which should accept a dictionary and return a string. The default is `json.dumps`.

Tip: JSON and Python representations are different. In particular, JSON *True* is written lowercase *true*. To convert JSON back to Python, use `json.loads(string)`.

If *formatter* is given but is not callable, `LenaTypeError` is raised.

__call__ (*value*)

Convert *value*'s context to `Context` on the fly.

If the *value* is a *(data, context)* pair, convert its context part to `Context`. If the *value* doesn't contain a context, it is created as an empty `Context`.

2.1.2 Functions

difference (*d1*, *d2*)

Return a dictionary with items from *d1* not contained in *d2*.

If a key is present both in *d1* and *d2* but has different values, it is included into the difference.

get_recursively (*d*, *keys*, *default=<object object>*)

Get value from a dictionary *d* recursively.

keys can be a list of simple keys (strings), a dot-separated string or a dictionary with at most one key at each level. A string is split by dots and used as a list. A list of keys is searched in the dictionary recursively (it represents nested dictionaries). If any of them is not found, *default* is returned if "default" is given, otherwise `LenaKeyError` is raised.

Note: Python's dict.get in case of a missing value returns `None` and never raises an error. We implement it differently, because it allows more flexibility.

If *d* is not a dictionary or if *keys* have unknown types, `LenaTypeError` is raised. If *keys* is a dictionary with more than one key at some level, `LenaValueError` is raised.

If *keys* is empty, *d* is returned.

Examples:

```
>>> context = {"output": {"latex": {"name": "x"}}}
>>> get_recursively(context, ["output", "latex", "name"], default="y")
'x'
>>> get_recursively(context, "output.latex.name")
'x'
```

intersection (**dicts*, ***kwargs*)

Return a dictionary, such that each of its items are contained in all *dicts* (recursively).

dicts are several dictionaries. If *dicts* is empty, an empty dictionary is returned.

A keyword argument *level* sets maximum number of recursions. For example, if *level* is 0, all *dicts* must be equal (otherwise an empty dict is returned). If *level* is 1, the result contains those subdictionaries which are equal. For arbitrarily nested subdictionaries set *level* to -1 (default).

Example:

```
>>> from lena.context import intersection
>>> d1 = {1: "1", 2: {3: "3", 4: "4"}}
>>> d2 = {2: {4: "4"}}
>>> # by default level is -1, which means infinite recursion
>>> intersection(d1, d2) == d2
True
>>> intersection(d1, d2, level=0)
{}
>>> intersection(d1, d2, level=1)
{}
>>> intersection(d1, d2, level=2)
{2: {4: '4'}}
```

This function always returns a dictionary or its subtype (copied from dicts[0]). All values are deeply copied. No dictionary or subdictionary is changed.

If any of *dicts* is not a dictionary or if some *kwargs* are unknown, `LenaTypeError` is raised.

str_to_dict (*s*)
Create a dictionary from a dot-separated string *s*.

Dots represent nested dictionaries. *s* must have at least two dot-separated parts (*a.b*), otherwise `LenaValueError` is raised.

Example:

```
>>> str_to_dict("a.b.c d")
{'a': {'b': 'c d'}}
```

update_nested (*d*, *other*)
Update dictionary *d* with items from *other* dictionary.

other must be a dictionary of one element, which is used as a key. If *d* doesn't contain the key, *d* is updated with *other*. If *d* contains the key, the value with that key is nested inside the copy of *other* at the level which doesn't contain the key. *d* is updated.

If *d[key]* is not a dictionary or if there is not one key in *other*, `LenaValueError` is raised.

update_recursively (*d*, *other*)
Update dictionary *d* with items from *other* dictionary.

other can be a dot-separated string. In this case *str_to_dict()* is used to convert it to a dictionary.

Existing values are updated recursively, that is including nested subdictionaries. For example:

```
>>> d1 = {"a": 1, "b": {"c": 3}}
>>> d2 = {"b": {"d": 4}}
>>> update_recursively(d1, d2)
>>> d1 == {'a': 1, 'b': {'c': 3, 'd': 4}}
True
>>> # Usual update would have made d1["b"] = {"d": 4}, erasing "c".
```

Non-dictionary items from *other* overwrite those in *d*:

```
>>> update_recursively(d1, {"b": 2})
>>> d1 == {'a': 1, 'b': 2}
True
```

Both *d* and *other* must be dictionaries, otherwise `LenaTypeError` is raised.

2.2 lena.core

Sequences:

Sequence(*args)	Sequence of elements, such that next takes input from the previous during *run*.
Source(*args)	Sequence with no input flow.
FillComputeSeq(*args)	Sequence with one `FillCompute` element.
FillRequestSeq(*args, **kwargs)	Sequence with one `FillRequest` element.
Split(seqs[, bufsize, copy_buf])	Split data flow and run analysis in parallel.

Adapters:

Call(el[, call])	Adapter to provide *__call__(value)* method.
FillCompute(el[, fill, compute])	Adapter for a *FillCompute* element.
FillInto(el[, fill_into, explicit])	Adapter for a FillInto element.
FillRequest(el[, fill, request, reset, bufsize])	Adapter for a *FillRequest* element.
Run(el[, run])	Adapter for a *Run* element.
SourceEl(el[, call])	Adapter to provide *__call__()* method.

Exceptions:

LenaAttributeError	
LenaEnvironmentError	The base class for exceptions that can occur outside the Python system, like IOError or OSError.
LenaException	Base class for all Lena exceptions.
LenaIndexError	
LenaKeyError	
LenaRuntimeError	Raised when an error does not belong to other categories.
LenaStopFill	Signal that no more fill is accepted.
LenaTypeError	
LenaValueError	

2.2.1 Sequences

Lena combines calculations using *sequences*. *Sequences* consist of *elements*. Basic Lena sequences and element types are defined in this module.

class Sequence(**args*)

Sequence of elements, such that next takes input from the previous during *run*.

Sequence.run() must accept input flow. For sequence with no input data use *Source*.

args are objects which implement a method *run(flow)* or callables.

args can be a single tuple of such elements. In this case one doesn't need to check argument type when initializing a Sequence in a general function.

For more information about the *run* method and callables, see Run.

run (*flow*)

Generator, which transforms the incoming flow.

If this *Sequence* is empty, the flow passes untransformed, with a small change. This function converts input flow to an iterator, so that it always contains both *iter* and *next* methods. This is done for the flow entering the first sequence element and exiting from the sequence.

class Source (**args*)

Sequence with no input flow.

First argument is the initial element with no input flow. Following arguments (if present) form a sequence of elements, each accepting computational flow from the previous element.

```
>>> from lena.flow import CountFrom
>>> s = Source(CountFrom())
>>> for i in s():
...     if i == 5:
...         break
...     print(i, end=" ")
0 1 2 3 4
```

For a *sequence* which transforms the incoming flow, use *Sequence*.

__call__ ()

Generate flow.

class FillComputeSeq (**args*)

Sequence with one FillCompute element.

Input flow is preprocessed with the *Sequence* before the *FillCompute* element, then it fills the *FillCompute* element.

When the results are *computed*, they are postprocessed with the *Sequence* after that element.

args form a sequence with a *FillCompute* element.

If *args* contain several *FillCompute* elements, only the first one is chosen (the subsequent ones are used as simple *Run* elements). To change that, explicitly cast the first element to FillInto.

If *FillCompute* element was not found, or if the sequences before and after that could not be correctly initialized, LenaTypeError is raised.

compute ()

Compute the results and yield.

If the sequence after *FillCompute* is not empty, it postprocesses the results yielded from *FillCompute* element.

fill (*value*)

Fill *self* with *value*.

If the sequence before FillCompute is not empty, it preprocesses the *value* before filling *FillCompute*.

class FillRequestSeq (**args*, ***kwargs*)

Sequence with one FillRequest element.

Input flow is preprocessed with the *Sequence* before the *FillRequest* element, then it fills the *FillRequest* element.

When the results are yielded from the *FillRequest*, they are postprocessed with the *Sequence* after that element.

args form a sequence with a *FillRequest* element.

If *args* contains several *FillRequest* elements, only the first one is chosen (the subsequent ones are used as simple *Run* elements). To change that, explicitly cast the first element to `FillInto`.

kwargs can contain *bufsize*, which is used during *run*. See `FillRequest` for more information on *run*. By default *bufsize* is *1*. Other *kwargs* raise `LenaTypeError`.

If *FillRequest* element was not found, or if the sequences before or after that could not be correctly initialized, `LenaTypeError` is raised.

fill(*value*)
Fill *self* with *value*.

If the sequence before *FillRequest* is not empty, it preprocesses the *value* before filling *FillRequest*.

request()
Request the results and yield.

If the sequence after *FillRequest* is not empty, it postprocesses the results yielded from the *FillRequest* element.

reset()
Reset the *FillRequest* element.

class Split(*seqs*, *bufsize=1000*, *copy_buf=True*)
Split data flow and run analysis in parallel.

seqs must be a list of Sequence, Source, FillComputeSeq or FillRequestSeq sequences (any other container will raise `LenaTypeError`). If *seqs* is empty, *Split* acts as an empty *Sequence* and yields all values it receives.

bufsize is the size of the buffer for the input flow. If *bufsize* is `None`, whole input flow is materialized in the buffer. *bufsize* must be a natural number or `None`, otherwise `LenaValueError` is raised.

copy_buf sets whether the buffer should be copied during *run*. This is important if different sequences can change input data and interfere with each other.

Common type: If each sequence from *seqs* has a common type, *Split* creates methods corresponding to this type. For example, if each sequence is *FillCompute*, *Split* creates methods *fill* and *compute* and can be used as a *FillCompute* sequence. *fill* fills all its subsequences (with copies if *copy_buf* is True), and *compute* yields values from all sequences in turn (as would also do *request* or *Source.__call__*).

run(*flow*)
Iterate input *flow* and yield results.

The *flow* is divided into subslices of *bufsize*. Each subslice is processed by sequences in the order of their initializer list.

If a sequence is a *Source*, it doesn't accept the incoming *flow*, but produces its own complete flow and becomes inactive (is not called any more).

A *FillRequestSeq* is filled with the buffer contents. After the buffer is finished, it yields all values from *request()*.

A *FillComputeSeq* is filled with values from each buffer, but yields values from *compute* only after the whole *flow* is finished.

A *Sequence* is called with *run(buffer)* instead of the whole flow. The results are yielded for each buffer (and also if the *flow* was empty). If the whole flow must be analysed at once, don't use such a sequence in *Split*.

If the *flow* was empty, each *call*, *compute*, *request* or *run* is called nevertheless.

If *copy_buf* is True, then the buffer for each sequence except the last one is a deep copy of the current buffer.

2.2.2 Adapters

Adapters allow to use existing objects as Lena core elements.

Adapters can be used for several purposes:

- provide an unusual name for a method (*Run(my_obj, run="my_run")*).
- hide unused methods to prevent ambiguity.
- automatically convert objects of one type to another in sequences (*FillCompute* to *Run*).
- explicitly cast object of one type to another (*FillRequest* to *FillCompute*).

Example:

```
>>> class MyEl(object):
...     def my_run(self, flow):
...         for val in flow:
...             yield val
...
>>> my_run = Run(MyEl(), run="my_run")
>>> list(my_run.run([1, 2, 3]))
[1, 2, 3]
```

class Call(*el*, *call=<object object>*)

Adapter to provide *__call__(value)* method.

Name of the actually called method can be customized during the initialization.

The method *__call__(value)* is a simple (preferably pure) function, which accepts a *value* and returns its transformation.

Element *el* must contain a callable method *call* or be callable itself.

If *call* method name is not provided, it is checked whether *el* is callable itself.

If `Call` failed to instantiate with *el* and *call*, `LenaTypeError` is raised.

__call__(*value*)

Transform the *value* and return.

class FillCompute(*el*, *fill='fill'*, *compute='compute'*)

Adapter for a *FillCompute* element.

A *FillCompute* element has methods *fill(value)* and *compute()*.

Method names can be customized through *fill* and *compute* keyword arguments during the initialization.

FillCompute can be explicitly cast from *FillRequest*. In this case *compute* is *request*.

If callable methods *fill* and *compute* or *request* were not found, `LenaTypeError` is raised.

compute()

Yield computed values.

fill(*value*)

Fill *self* with *value*.

class FillInto(*el*, *fill_into=<object object>*, *explicit=True*)
Adapter for a FillInto element.

Element *el* must implement *fill_into* method, be callable or be a Run element.

If no *fill_into* argument is provided, then *fill_into* method is searched, then *__call__*, then *run*. If none of them is found and callable, `LenaTypeError` is raised.

Note that callable elements and elements with *fill_into* method have different interface. If the *el* is callable, it is assumed to be a simple function, which accepts a single value and transforms that, and the result is filled into the element by this adapter. *fill_into* method, on the contrary, takes two arguments (element and value) and fills the element itself. This allows to use lambdas directly in *FillInto*.

A *Run* element is converted to *FillInto* this way: for each value the *el* runs a flow consisting of this one value and fills the results into the output element. This can be done only if *explicit* is True.

fill_into(*element*, *value*)
Fill *value* into an *element*.

Value is transformed by the initialization element before filling *el*.

Element must provide a *fill* method.

class FillRequest(*el*, *fill='fill'*, *request='request'*, *reset=True*, *bufsize=1*)
Adapter for a *FillRequest* element.

A *FillRequest* element has methods *fill(value)* and *request()*.

Names for *fill* and *request* can be customized during initialization.

FillRequest can be initialized from a *FillCompute* element. If a callable *request* method was not found, *el* must have a callable *compute* method. *request* in this case is *compute*.

By default, *FillRequest* implements *run* method that splits the flow into subslices of *bufsize* elements. If *el* has a callable *run* method, it is used instead of the default one.

If a keyword argument *reset* is `True` (default), *el* must have a method *reset, and in this case :meth:'reset' is called after each :meth:'request' (including those during :meth:'run'). If *reset* is `False`, `reset()` is never called.

Attributes

`bufsize` is the maximum size of subslices during *run*.

bufsize must be a natural number, otherwise `LenaValueError` is raised. If callable *fill* and *request* methods were not found, or *FillRequest* could not be derived from *FillCompute*, or if *reset* is `True`, but *el* has no method *reset*, `LenaTypeError` is raised.

fill(*value*)
Fill *self* with *value*.

request()
Yield computed values.

May be called at any time, the flow may still contain zero or more items.

reset()
Reset the element *el*.

run(*flow*)
Implement *run* method.

First, *fill* is called for each value in a subslice of *flow* of *self.bufsize* size. After that, results are yielded from *self.request()*. This repeats until the *flow* is exhausted.

If *fill* was not called even once (*flow* is empty), the results for a general *FillRequest* are undefined (for example, it can run *request* or raise an exception). This adapter runs *request* in this case. If the last slice is empty, *request* is not run for that. Note that the last slice may contain less than *bufsize* values. If that is important, implement your own method.

A slice is a non-materialized list, which means that it will not take place of *bufsize* in memory.

class Run(*el*, *run=<object object>*)

Adapter for a *Run* element.

Name of the method *run* can be customized during initialization.

If *run* argument is supplied, *el* must be None or it must have a callable method with name given by *run*.

If *run* keyword argument is missing, then *el* is searched for a method *run*. If that is not found, a type cast is attempted.

A *Run* element can be initialized from a *Call* or a *FillCompute* element.

A callable element is run as a transformation function, which accepts single values from the flow and *returns* their transformations for each value.

A *FillCompute* element is run the following way: first, *el.fill(value)* is called for the whole flow. After the flow is exhausted, *el.compute()* is called.

It is possible to initialize `Run` using a generator function without an element. To do that, set the element to `None`: *Run(None, run=<my_function>)*.

If the initialization failed, `LenaTypeError` is raised.

`Run` is used implicitly during the initialization of `Sequence`.

run(*flow*)

Yield transformed elements from the incoming *flow*.

class SourceEl(*el*, *call=<object object>*)

Adapter to provide *__call__()* method. Name of the actually called method can be customized during the initialization.

The `__call__()` method is a generator, which yields values. It doesn't accept any input flow.

Element *el* must contain a callable method *__call__* or be callable itself.

If *call* function or method name is not provided, it is checked whether *el* is callable itself.

If `SourceEl` failed to instantiate with *el* and *call*, `LenaTypeError` is raised.

__call__()

Yield generated values.

2.2.3 Exceptions

All Lena exceptions are subclasses of `LenaException` and corresponding Python exceptions (if they exist).

exception LenaAttributeError

Bases: `lena.core.exceptions.LenaException`, `AttributeError`

exception LenaEnvironmentError

Bases: `lena.core.exceptions.LenaException`, `OSError`

The base class for exceptions that can occur outside the Python system, like IOError or OSError.

exception LenaException
Bases: `Exception`

Base class for all Lena exceptions.

exception LenaIndexError
Bases: *lena.core.exceptions.LenaException*, `IndexError`

exception LenaKeyError
Bases: *lena.core.exceptions.LenaException*, `KeyError`

exception LenaNotImplementedError
Bases: *lena.core.exceptions.LenaException*, `NotImplementedError`

exception LenaRuntimeError
Bases: *lena.core.exceptions.LenaException*, `RuntimeError`

Raised when an error does not belong to other categories.

exception LenaStopFill
Bases: *lena.core.exceptions.LenaException*

Signal that no more fill is accepted.

Analogous to StopIteration, but control flow is reversed.

exception LenaTypeError
Bases: *lena.core.exceptions.LenaException*, `TypeError`

exception LenaValueError
Bases: *lena.core.exceptions.LenaException*, `ValueError`

exception LenaZeroDivisionError
Bases: *lena.core.exceptions.LenaException*, `ZeroDivisionError`

2.3 Flow

Elements:

Cache(filename[, method, protocol])	Cache flow passing through.
DropContext(*args)	Sequence, which transform *(data, context)* flow so that only *data* remains in the inner sequence.
End	Stop sequence here.
Print([before, sep, end, transform])	Print values passing through.

Functions:

get_context(value)	Get context from a possible *(data, context)* pair.
get_data(value)	Get data from *value* (a possible *(data, context)* pair).
get_data_context(value)	Get (data, context) from *value* (a possible *(data, context)* pair).
seq_map(seq, container[, one_result])	Map Lena Sequence *seq* to the *container*.

Group plots:

GroupBy(group_by)	Group data.
GroupPlots(group_by, select[, transform, . . .])	Group several plots.
GroupScale(scale_to[, allow_zero_scale, . . .])	Scale a group of data.
Selector(selector)	Determine whether an item should be selected.

Iterators:

Chain(*iterables)	Chain generators.
CountFrom([start, step])	Generate numbers from *start* to infinity, with *step* between values.
ISlice(*args)	Slice iterable from *start* to *stop* with *step*.

Split into bins:

SplitIntoBins(seq, arg_func, edges[, transform])	Split analysis into bins.

2.3.1 Elements

Elements form Lena sequences. This group contains miscellaneous elements, which didn't fit other categories.

class Cache (*filename*, *method='cPickle'*, *protocol=2*)

Cache flow passing through.

On the first run, dump all flow to file (and yield the flow unaltered). On subsequent runs, load all flow from that file in the original order.

Example:

```
s = Source(
         ReadFiles(),
         ReadEvents(),
         MakeHistograms(),
         Cache("histograms.pkl"),
         MakeStats(),
         Cache("stats.pkl"),
      )
```

If *stats.pkl* exists, Cache will read data flow from that file and no other processing will be done. If the *stats.pkl* cache doesn't exist, but the cache for histograms exist, it will be used and no previous processing (from *ReadFiles* to *MakeHistograms*) will occur. If both caches are not filled yet, processing will run as usually.

Only pickleable objects can be cached (otherwise a *pickle.PickleError* is raised).

> **Warning:** The pickle module is not secure against erroneous or maliciously constructed data. Never unpickle data from an untrusted source.

filename is the name of file where to store the cache. You can give it *.pkl* extension.

method can be *pickle* or *cPickle* (faster pickle). For Python3 they are same.

protocol is pickle protocol. Version 2 is the highest supported by Python 2. Version 0 is "human-readable" (as noted in the documentation). 3 is recommended if compatibility between Python 3 versions is needed. 4 was added in Python 3.4. It adds support for very large objects, pickling more kinds of objects, and some data format optimizations.

static alter_sequence(*seq*)
If the Sequence *seq* contains a `Cache`, which has an up-to-date cache, a `Source` is built based on the flattened *seq* and returned. Otherwise the *seq* is returned unchanged.

cache_exists()
Return `True` if file with cache exists and is readable.

drop_cache()
Remove file with cache if that exists, pass otherwise.

If cache exists and is readable, but could not be deleted, `LenaEnvironmentError` is raised.

run(*flow*)
Load cache or fill it.

If we can read *filename*, load flow from there. Otherwise use the incoming *flow* and fill the cache. All loaded or passing items are yielded.

class DropContext(**args*)
Sequence, which transform *(data, context)* flow so that only *data* remains in the inner sequence. Context is restored outside *DropContext*.

DropContext works for most simple cases as a *Sequence*, but may not work in more advanced circumstances. For example, since *DropContext* is not transparent, `Split` can't judge whether it has a *FillCompute* element inside, and this may lead to errors in the analysis. It is recommended to provide *context* when possible.

**args* will form a `Sequence`.

run(*flow*)
Run the sequence without context, and generate output flow restoring the context before *DropContext*.

If the sequence adds a context, the returned context is updated with that.

class End
Stop sequence here.

run(*flow*)
Exhaust all preceding flow and stop iteration (yield nothing to the following flow).

class Print(*before="*, *sep="*, *end='n'*, *transform=None*)
Print values passing through.

before is a string appended before the first element in the item (which may be a container).

sep separates elements, *end* is appended after the last element.

transform is a function which transforms passing items (for example, it can select its specific fields).

2.3.2 Functions

Functions to deal with data and context, and `seq_map()`.

A value is considered a (data, context) pair, if it is a tuple of length 2, and the second element is a dictionary or its subclass.

get_context(*value*)
Get context from a possible *(data, context)* pair.

If context is not found, return an empty dictionary.

get_data(*value*)

Get data from *value* (a possible *(data, context)* pair).

If context is not found, return *value*.

get_data_context(*value*)

Get (data, context) from *value* (a possible *(data, context)* pair).

If context is not found, (value, {}) is returned.

Since `get_data()` and `get_context()` both check whether context is present, this function may be slightly more efficient and compact than the other two.

seq_map(*seq*, *container*, *one_result=True*)

Map Lena Sequence *seq* to the *container*.

For each value from the *container*, calculate `seq.run([value])`. This can be a list or a single value. If *one_result* is True, the result must be a single value. In this case, if results contain less than or more than one element, `LenaValueError` is raised.

The list of results (lists or single values) is returned. The results are in the same order as read from the *container*.

2.3.3 Group plots

Group several plots into one.

Since data can be produced in different places, several classes are needed to support this. First, the plots of interest must be selected (for example, one-dimensional histograms). This is done by `Selector`. Selected plots must be grouped. For example, we may want to plot data *x* versus Monte-Carlo *x*, but not data *x* vs data *y*. Data is grouped by `GroupBy`. To preserve the group, we can't yield it to the following elements, but have to transform the plots inside `GroupPlots`. We can also scale (normalize) all plots to one using `GroupScale`.

class GroupBy(*group_by*)

Group data.

Data is added during `update()`. Groups are available as `groups` attribute.

Groups is a mapping of *keys* (return values of *group_by*) and lists of items with the same key.

Combine data with same attributes.

group_by is a function, which returns distinct hashable results for items from different groups.

It can be a dot-separated string, which corresponds to context. Otherwise, `LenaTypeError` is raised.

clear()

Remove all groups.

update(*val*)

Find a group for *val* and add it there.

A group key is calculated by *group_by*. If no such key exists, a new group is created.

class GroupPlots(*group_by*, *select*, *transform=()*, *scale_to=None*, *yield_selected=False*)

Group several plots.

Plots to be grouped are chosen by *select*, which acts as a boolean function. If *select* is not a `Selector`, it is converted to that class. See `Selector` for more options.

Plots are grouped by *group_by*, which returns different keys for different groups. If it is not an instance of `GroupBy`, it is converted to that class. See `GroupBy` for more options.

scale_to is a number or a string. A number means the scale, to which plots must be normalized. A string is a name of the plot to which other plots must be normalized. If *scale_to* is not an instance of `GroupScale`, it is converted to that class. If a plot could not be rescaled, `LenaValueError` is raised. For more options, use *`GroupScale`*.

transform is a sequence, which processes individual plots before yielding. For example, `transform=(HistToCSV(), writer)`. *transform* is called after *scale_to*.

yield_selected defines whether selected items should be yielded during *run* like other items. Use it if you want to have both single and combined plots. By default, selected plots are not yielded.

run(*flow*)
Run the flow and yield final groups.

Each item of the flow is checked with the selector. If it is selected, it is added to groups. Otherwised it is yielded.

After the flow is finished, groups are yielded. Groups are lists of items, which have same keys from *group_by*. Each group's context (including empty) is inserted into a list in *context.group*. The resulting context is updated with the intersection of groups' contexts. For uniformity, if *yield_selected* is True, single values are also updated: data is put into a list of one element, and context is updated with *group* key. Its value is copy (not deep copy) of context's values, so future updates to subdictionaries which existed during this run will be effective in *context.group*.

If *scale_to* was set, plots are normalized to the given value or plot. If that plot was not selected (is missing in the captured group) or its norm could not be calculated, `LenaValueError` is raised.

class GroupScale(*scale_to*, *allow_zero_scale=False*, *allow_unknown_scale=False*)
Scale a group of data.

scale_to defines the method of scaling. If a number is given, group items are scaled to that. Otherwise it is converted to a *`Selector`*, which must return a unique item from the group. Group items will be scaled to the scale of that item.

By default, attempts to rescale a structure with unknown or zero scale raise an error. If *allow_zero_scale* and *allow_unknown_scale* are set to `True`, the corresponding errors are ignored and the structure remains unscaled.

scale(*group*)
Scale *group* and return a rescaled group as a list.

The *group* can contain *(structure, context)* pairs. The original group is unchanged as long as structures' *scale* method returns a new structure (default for Lena histograms and graphs).

If any item could not be rescaled and options were not set to ignore that, `LenaValueError` is raised.

class Selector(*selector*)
Determine whether an item should be selected.

Generally, *selected* means the result is convertible to `True`, but other values can be used as well.

The usage of *selector* depends on its type.

If *selector* is a class, *`__call__()`* checks that data part of the value is subclassed from that.

A callable is used as is.

A string means that value's context must conform to that (as in `lena.context.check_context_str()`).

selector can be a container. In this case its items are converted to selectors. If *selector* is a *list*, the result is *or* applied to results of each item. If it is a *tuple*, boolean *and* is applied to the results.

If incorrect arguments are provided, `LenaTypeError` is raised.

__call__(*value*)
Check whether *value* is selected.

If an exception occurs, the result is False. It is safe to use non-existing attributes, etc.

2.3.4 Iterators

Adapters to iterators from `itertools`.

class Chain(**iterables*)

Chain generators.

`Chain` can be used as a `Source` to generate data.

Example:

```
>>> c = lena.flow.Chain([1, 2, 3], ['a', 'b'])
>>> list(c())
[1, 2, 3, 'a', 'b']
```

iterables will be chained during `__call__()`, that is after the first one is exhausted, the second is called, etc.

__call__()

Generate values from chained iterables.

class CountFrom(*start=0*, *step=1*)

Generate numbers from *start* to infinity, with *step* between values.

Similar to `itertools.count()`.

__call__()

Yield values from *start* to infinity with *step*.

class ISlice(**args*)

Slice iterable from *start* to *stop* with *step*.

Initialization:

`ISlice` (*stop*)

`ISlice` (*start, stop* [, *step*])

Similar to `itertools.islice()` or `range()`.

fill_into(*element*, *value*)

Fill *element* with *value*.

Element must have a `fill(value)` method.

run(*flow*)

Yield values from *start* to *stop* with *step*.

2.3.5 Split into bins

Split analysis on groups set by bins.

class ReduceBinContent(*select*, *transform*, *drop_bins_context=True*)

Transform bin content of histograms.

This class is used when histogram bins contain complex structures. For example, in order to plot a histogram with a 3-dimensional vector in each bin, we shall create 3 histograms corresponding to vector's components.

Select determines which types should be transformed. The types must be given in a `list` (not a tuple) or as a general `Selector`. Example: `select=[lena.math.vector3, list]`.

transform is a *Sequence* or element applied to bin contents. If *transform* is not a *Sequence* or an element with *run* method, it is converted to a *Sequence*. Example: `transform=Split([X(), Y(), Z()])` (provided that you have X, Y, Z variables).

ReduceBinContent creates histograms, which may be plotted, that is bins contain only data without context. By default, context of all bins except one is not used. If *drop_bins_context* is `False`, a histogram of bin context is added to context.

In case of wrong arguments, `LenaTypeError` is raised.

run(*flow*)
Transform histograms from *flow*.

Not selected values pass unchanged.

Context is updated with *variable*, *histogram* and *bin_content*. *variable" and *histogram* copy context from *split_into_bins* (if present there). *bin_content* includes context for example bin in "example_bin" and (optionally) for all bins in "all_bins".

class SplitIntoBins(*seq*, *arg_func*, *edges*, *transform=None*)
Split analysis into bins.

seq is a *FillComputeSeq* sequence, which corresponds to the analysis being compared for different bins. It can be a tuple containing a *FillCompute* element. Deep copy of *seq* will be used to produce each bin's content.

arg_func is a function which takes data and returns argument value used to compute the bin index. A *Variable* must be provided. Example of a two-dimensional function: `arg_func = lena.variables.Variable("xy", lambda event:  (event.x, event.y))`.

edges is a sequence of arrays containing monotonically increasing bin edges along each dimension. Example: `edges = lena.math.mesh((0, 1), 10)`.

transform is a *Sequence*, which is applied to results. The final histogram may contain vectors, histograms and any other data the analysis produced. To be able to plot them, *transform* can extract vector components or do other work to simplify structures. By default, *transform* is *TransformBins*. Pass an empty tuple to disable it.

Attributes: bins, edges.

If *edges* are not increasing, *LenaValueError* is raised. In case of other argument initialization problems, *LenaTypeError* is raised.

compute()
Yield a *(Histogram, context)* for *compute()* for each bin.

Histogram is created from `edges` and bins taken from compute() for `bins`. Context is preserved in histogram bins.

SplitIntoBins context is added to *context.split_into_bins* as *histogram* (corresponding to *edges*) and *variable* (corresponding to *arg_func*) subcontexts.

In Python 3 the minimum number of *compute()* among all bins is used. In Python 2, if some bin is exhausted before the others, its content will be filled with None.

fill(*val*)
Fill the cell corresponding to *arg_func(val)* with *val*.

Values outside of *edges* range are ignored.

class TransformBins(*create_edges_str=None*)
Transform bins into a flattened sequence.

create_edges_str is a callable, which creates a string from bin's edges and coordinate names and adds that to context. It is passed parameters *(edges, var_context)*, where *var_context* is Variable context containing variable names (it can be a single `Variable` or `Combine`).

By default, it is `cell_to_string()`.

If *create_edges_str* is not callable, `LenaTypeError` is raised.

`cell_to_string`(*cell_edges*, *var_context=None*, *coord_names=None*, *coord_fmt='{}_lte_{}_lt_{}'*, *coord_join='_'*, *reverse=False*)

Transform cell edges into a string.

cell_edges is a tuple of pairs *(lower bound, upper bound)* for each coordinate.

coord_names is a list of coordinates names.

coord_fmt is a string, which defines how to format individual coordinates.

coord_join is a string, which joins coordinate pairs.

If *reverse* is True, coordinates are joined in reverse order.

`get_example_bin`(*struct*)

Return bin with zero index on each axis of the histogram bins.

For example, if the histogram is two-dimensional, return hist[0][0].

struct can be a `Histogram` or an array of bins.

2.4 math package

Functions of multidimensional arguments:

`mesh`(ranges, nbins)	Generate equally spaced mesh of *nbins* cells in the given range.
`md_map`(f, array)	Multidimensional map.

Functions of scalar and multidimensional arguments:

`clip`(a, interval)	Clip (limit) the value.
`isclose`(a, b[, rel_tol, abs_tol])	Return `True` if *a* and *b* are approximately equal, and `False` otherwise.

Elements:

`Mean`([start, pass_on_empty])	Calculate mean (average) of input values.
`Sum`([start])	Calculate sum of input values.

3-dimensional vector:

`vector3`(v)	3-dimensional vector with Cartesian and spherical coordinates.

2.4.1 Functions of multidimensional arguments

mesh (*ranges*, *nbins*)

Generate equally spaced mesh of *nbins* cells in the given range.

Parameters

- **ranges** – a pair of (min, max) values for 1-dimensional range, or a list of ranges in corresponding dimensions.
- **nbins** – number of bins for 1-dimensional range, or a list of number of bins in corresponding dimensions.

```
>>> from lena.math import mesh
>>> mesh((0, 1), 2)
[0, 0.5, 1]
>>> mesh(((0, 1), (10, 12)), (1, 2))
[[0, 1], [10, 11.0, 12]]
```

Note that because of rounding errors two meshes should not be naively compared, they will probably appear different. One should use *isclose* for comparison.

```
>>> from lena.math import isclose
>>> isclose(mesh((0, 1), 10),
...         [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0])
True
```

md_map (*f*, *array*)

Multidimensional map.

Return function *f* mapped to contents of a multidimensional *array*. *f* is a function of one argument.

Array must be a list of (possibly nested) lists. Its contents remain unchanged. Returned array has same dimensions as the initial one. If *array* is not a list, `LenaTypeError` is raised.

```
>>> from lena.math import md_map
>>> arr = [-1, 1, 0]
>>> md_map(abs, arr)
[1, 1, 0]
>>> arr = [[0, -1], [2, 3]]
>>> md_map(abs, arr)
[[0, 1], [2, 3]]
```

2.4.2 Functions of scalar and multidimensional arguments

clip (*a*, *interval*)

Clip (limit) the value.

Given an interval *(a_min, a_max)*, values of *a* outside the interval are clipped to the interval edges. For example, if an interval of *[0, 1]* is specified, values smaller than 0 become 0, and values larger than 1 become 1.

```
>>> clip(-1, (0, 1))
0
>>> # tuple looks better, but list can be used too
>>> clip(2, [0, 1])
1
>>> clip(0.5, (0, 1))
0.5
```

If *a_min* > *a_max* or if *interval* has length more than 2, `LenaValueError` is raised. If *interval* is not a container, `LenaTypeError` is raised.

isclose(*a*, *b*, *rel_tol=1e-09*, *abs_tol=0.0*)

Return `True` if *a* and *b* are approximately equal, and `False` otherwise.

rel_tol is the relative tolerance. It is multiplied by the greater of the magnitudes of the two arguments; as the values get larger, so does the allowed difference between them while still considering them close.

abs_tol is the absolute tolerance. If the difference is less than either of those tolerances, the values are considered equal.

a and *b* must be either numbers or lists/tuples of same dimensions (may be nested), or have a method *isclose*. Otherwise `LenaTypeError` is raised. For containers, *isclose* is called elementwise. If every corresponding element is close, the containers are close. Dimensions are not checked to be equal.

First, *a* and *b* are checked if any of them has *isclose* method. If *a* and *b* both have *isclose* method, then they must both return `True` to be close. Otherwise, if only one of *a* or *b* has *isclose* method, it is called.

Special values of `NaN`, `inf`, and `-inf` are not supported.

```
>>> isclose(1, 2)
False
>>> isclose([1, 2, 3], (1, 2., 3))
True
```

This function for scalar numbers appeared in `math` module in *Python 3.5*.

2.4.3 Elements

Elements for mathematical calculations.

class Mean(*start=0*, *pass_on_empty=False*)

Calculate mean (average) of input values.

start is the initial value of sum.

If *pass_on_empty* is True, then if nothing was filled, don't yield anything. By default it raises an error (see `compute()`).

compute()

Calculate mean and yield.

If the current context is not empty, yield *(mean, context)*. Otherwise yield only *mean*.

If no values were filled (count is zero), mean can't be calculated and `LenaZeroDivisionError` is raised. This can be changed to yielding nothing if *pass_on_empty* was initialized to True.

fill(*value*)

Fill *self* with *value*.

The *value* can be a *(data, context)* pair. The last *context* value (if missing, it is considered empty) is saved for output.

reset()

Reset sum, count and context.

Sum is reset to *start* value, count to zero and context to {}.

class Sum(*start=0*)

Calculate sum of input values.

start is the initial value of sum.

compute()
Calculate the sum and yield.

If the current context is not empty, yield *(sum, context)*. Otherwise yield only *sum*.

fill(*value*)
Fill *self* with *value*.

The *value* can be a *(data, context)* pair. The last *context* value (considered empty if missing) sets the current context.

reset()
Reset sum and context.

Sum is reset to *start* value and context to {}.

2.4.4 3-dimensional vector

vector3 is a 3-dimensional vector with float coordinates. It supports spherical coordinates and basic vector operations.

Initialization, vector addition and scalar multiplication create new vectors:

```
>>> v1 = vector3([0, 1, 2])
>>> v2 = vector3([3, 4, 5])
>>> v1 + v2
vector3([3.0, 5.0, 7.0])
>>> v1 - v2
vector3([-3.0, -3.0, -3.0])
>>> 3 * v1
vector3([0.0, 3.0, 6.0])
>>> v1 * 3
vector3([0.0, 3.0, 6.0])
```

Vector attributes can be set and read. Vectors can be tested for exact or approximate equality with == and `isclose` method.

```
>>> v2.z = 0
>>> v2
vector3([3.0, 4.0, 0.0])
>>> v2.r = 10
>>> v2 == vector3([6, 8, 0])
True
>>> v2.theta = 0
>>> v2.isclose(vector3([0, 0, 10]))
True
>>> from math import pi
>>> v2.phi = 0
>>> v2.theta = pi/2.
>>> v2.isclose(vector3([10, 0, 0]))
True
```

class vector3(*v*)
3-dimensional vector with Cartesian and spherical coordinates.

Create vector3 from Cartesian coordinates.

v should be a container of size 3 (will be transformed to a list of floats).

Attributes

vector3 has usual vector attributes: *x*, *y*, *z* and spherical coordinates *r*, *phi*, *theta*.

They are connected through this formula:

$$x = r * \cos(\phi) * \sin(\theta),$$
$$y = r * \sin(\phi) * \sin(\theta),$$
$$z = r * \cos(\theta),$$

$\phi \in [0, 2\pi], \theta \in [0, \pi]$.

ϕ and $\phi + 2\pi$ are equal.

Cartesian coordinates can be obtained and set through indices starting from 0 (v.x = v[0]). In this respect, *vector3* behaves as a container of length 3.

Only Cartesian coordinates are stored internally (spherical coordinates are recomputed each time).

Attributes can be got and set using subscript or a function set*, get*. For example:

```
>>> v = vector3([1, 0, 0])
>>> v.x = 0
>>> x = v.getx()
>>> v.setx(x+1)
>>> v
vector3([1.0, 0.0, 0.0])
```

r^2 and $\cos\theta$ can be obtained with methods *getr2()* and *getcostheta()*.

Comparisons

For elementwise comparison of two vectors one can use '==' and '!=' operators. Because of rounding errors, this can often show two same vectors as different. In general, it is recommended to use approximate comparison with `isclose` method.

Comparisons like '>', '<=' are all prohibited: if one tries to use these operators, `LenaTypeError` is raised.

Truth testing

vector3 is non-zero if its magnitude (*r*) is not 0.

Vector operations

3-dimensional vectors can be added and subtracted, multiplied or divided by a scalar. Multiplication by a scalar can be written from any side of the vector (c*v or v*c). A vector can also be negated (*-v*).

For other vector operations see methods below.

classmethod fromspherical(*r*, *phi*, *theta*)

Construct vector3 from spherical coordinates.

r is magnitude, *phi* is azimuth angle from 0 to $2 * \pi$, *theta* is polar angle from 0 (z = 1) to π (z = -1).

```
>>> from math import pi
>>> vector3.fromspherical(1, 0, 0)
vector3([0.0, 0.0, 1.0])
>>> vector3.fromspherical(1, 0, pi).isclose(vector3([0, 0, -1]))
True
>>> vector3([1, 0, 0]).isclose(vector3.fromspherical(1, 0, pi/2))
True
>>> vector3.fromspherical(1, pi, 0).isclose(vector3([0.0, 0.0, 1.0]))
True
>>> vector3.fromspherical(1, pi/2, pi/2).isclose(vector3([0.0, 1.0, 0.0]))
True
```

angle (*B*)

The angle between self and *B*, in radians.

```
>>> v1 = vector3([0, 3, 4])
>>> v2 = vector3([0, 3, 4])
>>> v1.angle(v2)
0.0
>>> v2 = vector3([0, -4, 3])
>>> from math import degrees
>>> degrees(v1.angle(v2))
90.0
>>> v2 = vector3([0, -30, -40])
>>> degrees(v1.angle(v2))
180.0
```

cosine (*B*)

Cosine of the angle between self and *B*.

```
>>> v1 = vector3([0, 3, 4])
>>> v2 = vector3([0, 3, 4])
>>> v1.cosine(v2)
1.0
>>> v2 = vector3([0, -4, 3])
>>> v1.cosine(v2)
0.0
>>> v2 = vector3([0, -30, -40])
>>> v1.cosine(v2)
-1.0
```

cross (*B*)

The cross product between self and *B*, $A \times B$.

```
>>> v1 = vector3([0, 3, 4])
>>> v2 = vector3([0, 1, 0])
>>> v1.cross(v2)
vector3([-4.0, 0.0, 0.0])
```

dot (*B*)

The scalar product between self and *B*, $A \cdot B$.

classmethod fromspherical (*r*, *phi*, *theta*)

Construct vector3 from spherical coordinates.

r is magnitude, *phi* is azimuth angle from 0 to $2 * \pi$, *theta* is polar angle from 0 (z = 1) to π (z = -1).

```
>>> from math import pi
>>> vector3.fromspherical(1, 0, 0)
vector3([0.0, 0.0, 1.0])
>>> vector3.fromspherical(1, 0, pi).isclose(vector3([0, 0, -1]))
True
>>> vector3([1, 0, 0]).isclose(vector3.fromspherical(1, 0, pi/2))
True
>>> vector3.fromspherical(1, pi, 0).isclose(vector3([0.0, 0.0, 1.0]))
True
>>> vector3.fromspherical(1, pi/2, pi/2).isclose(vector3([0.0, 1.0, 0.0]))
True
```

isclose (*B*, *rel_tol=1e-09*, *abs_tol=0.0*)

Test whether two vectors are approximately equal.

Parameter semantics is the same as for the general *isclose*.

```
>>> v1 = vector3([0, 1, 2])
>>> v1.isclose(vector3([1e-11, 1, 2]))
True
```

norm()

$A/|A|$, a unit vector in the direction of self.

```
>>> v1 = vector3([0, 3, 4])
>>> n1 = v1.norm()
>>> v1n = vector3([0, 0.6, 0.8])
>>> (n1 - v1n)._mag() < 1e-6
True
```

proj(*B*)

The vector projection of self along B.

A.proj(B) = $(A \cdot norm(B))norm(B)$.

```
>>> v1 = vector3([0, 3, 4])
>>> v2 = vector3([0, 2, 0])
>>> v1.proj(v2)
vector3([0.0, 3.0, 0.0])
```

rotate(*theta*, *B*)

Rotate self around *B* through angle *theta*.

From the position where *B* points towards us, the rotation is counterclockwise (the right hand rule).

```
>>> v1 = vector3([1, 1, 1.0])
>>> v2 = vector3([0, 1, 0.0])
>>> from math import pi
>>> vrot = v1.rotate(pi/2, v2)
>>> vrot.isclose(vector3([1.0, 1.0, -1.0]))
True
```

scalar_proj(*B*)

The scalar projection of self along B.

A.scalar_proj(B) = $A \cdot norm(B)$.

```
>>> v1 = vector3([0, 3, 4])
>>> v2 = vector3([0, 2, 0])
>>> v1.scalar_proj(v2)
3.0
```

2.5 Output

Output:

HistToCSV(**kwargs)	Deprecated.
PDFToPNG([format, timeoutsec])	Convert PDF to image format (by default PNG).
ToCSV([separator, header, duplicate_last_bin])	Convert data to CSV text.
Writer([output_directory, output_filename])	Write text data to filesystem.

LaTeX utilities:

LaTeXToPDF([verbose, create_command])	Run pdflatex binary for LaTeX files.
RenderLaTeX([select_template, ...])	Create LaTeX from templates and data.

Make filename:

format_context(format_str, *args, **kwargs)	Create a function, which formats a given string using a context.
MakeFilename(*args, **kwargs)	Make file names for data from the flow.

2.5.1 Output

class PDFToPNG(*format='png'*, *timeoutsec=60*)

Convert PDF to image format (by default PNG).

Initialize output *format*.

timeoutsec is time (in seconds) for subprocess timeout (used only in Python 3). If the timeout expires, the child process will be killed and waited for. The TimeoutExpired exception will be re-raised after the child process has terminated.

This class uses `pdftoppm` binary internally. Pdftoppm can be given other output formats as an option (see `man pdftoppm`), for example *jpeg* or *tiff*.

run(*flow*)

Convert PDF files to *format*.

PDF files are recognized via *context.output.filetype*. Their paths are assumed to be data part of the value (may contain trailing ".pdf").

Data yielded is the resulting file name. Context is updated with filetype = *format*.

Other values are passed unchanged.

class Writer(*output_directory=''*, *output_filename='output'*)

Write text data to filesystem.

output_directory is the base output directory. It can be further appended by the incoming data. Non-existing directories are created.

output_filename is the name for unnamed data. Use it to write only one file.

If no arguments are given, the default is to write to "output.txt" in the current directory (rewritten for every new value) (unless different extensions are provided through the context). It is recommended to create filename explicitly using `MakeFilename`. The default writer's output file can be useful in case of errors, when explicit file name didn't work.

run(*flow*)

Write incoming data to file system.

Only strings (and unicode in Python 2) are written. To be written, data must have "output" dictionary in context and *context["output"]["writer"]* not set to `False`. Other values pass unchanged.

Full name of the file to be written (*filepath*) has the form self.output_directory/dirname/filename.fileext, where dirname, filename and file extension are searched in *context["output"]*. If *filename* is missing, Writer's default filename is used. If *fileext* is missing, then *filetype* is used; if it is also absent, the default file extension is "txt". It is recommended to provide only *fileext* in context, unless it differs with *filetype*.

File name with full path is yielded as data. Context.output is updated with *fileext* and *filename* (in case they were not present), and *filepath*, where *filename* is its base part (without output directory and extension) and *filepath* is the complete path.

If context.output.filename is present, but empty, `LenaRuntimeError` is raised.

class ToCSV(*separator=', '*, *header=None*, *duplicate_last_bin=True*)

Convert data to CSV text.

These objects are converted:

- `Histogram` (implemented only for 1- and 2-dimensional histograms).
- any object (including `Graph`) with *to_csv* method.

separator delimits values in the output text,

header is a string which becomes the first line of the output,

If *duplicate_last_bin* is `True`, contents of the last bin will be written in the end twice. This may be useful for graphical representation: if last bin is from 9 to 10, then the plot may end on 9, while this parameter allows to write bin content at 10, creating the last horizontal step.

run(*flow*)

Convert values from *flow* to CSV text.

Context.output is updated with {"filetype": "csv"}. All not converted data is yielded unchanged.

If *data* has *to_csv* method, it must accept keyword arguments *separator* and *header* and return text.

If *context.output.to_csv* is False, the value is skipped.

Data is yielded as a whole CSV block. To generate CSV line by line, use `hist1d_to_csv()` and `hist2d_to_csv()`.

hist1d_to_csv(*hist*, *header=None*, *separator=', '*, *duplicate_last_bin=True*)

Yield CSV-formatted strings for a one-dimensional histogram.

hist2d_to_csv(*hist*, *header=None*, *separator=', '*, *duplicate_last_bin=True*)

Yield CSV-formatted strings for a two-dimensional histogram.

class HistToCSV(***kwargs*)

Deprecated. Use `ToCSV` instead.

2.5.2 LaTeX

class LaTeXToPDF(*verbose=1*, *create_command=None*)

Run pdflatex binary for LaTeX files.

It runs in parallel (separate process is spawned for each job) and non-interactively.

Initialize object.

verbose = 0 means no output messages. 1 prints pdflatex error messages. More than 1 prints pdflatex stdout.

If you need to run pdflatex (or other executable) with different parameters, provide its command.

create_command is a function which accepts texfile_name, outfilename, output_directory, context (in this order) and returns a list made of the command and its arguments.

Default command is:

["pdflatex", "-halt-on-error", "-interaction", "batchmode", "-output-directory", output_directory, texfile_name]

run (*flow*)
Convert all incoming LaTeX files to pdf.

class RenderLaTeX (*select_template=''*, *template_path='.'*, *select_data=None*)
Create LaTeX from templates and data.

select_template is a string or a callable. If a string, it is the name of the template to be used (unless *context.output.template* overwrites that). If *select_template* is a callable, it must accept a value from the flow and return template name. If *select_template* is an empty string (default) and no template could be found in the context, `LenaRuntimeError` is raised.

template_path is the path for templates (used in jinja2.FileSystemLoader). By default, it is the current directory.

select_data is a callable to choose data to be rendered. It should accept a value from flow and return boolean. If it is not provided, by default CSV files are selected.

run (*flow*)
Render values from *flow* to LaTeX.

If no *select_data* was initialized, values with *context.output.filetype* equal to "csv" are selected by default.

Rendered LaTeX text is yielded in the data part of the tuple (no write to filesystem occurs). *context.output.filetype* updates to "tex".

Not selected values pass unchanged.

2.5.3 Make filename

class MakeFilename (**args*, ***kwargs*)
Make file names for data from the flow.

`MakeFilename` can be initialized using a single string, a Sequence or from keyword arguments.

A single string is a file name without extension (but it can contain a relative path).

Otherwise, all positional arguments will make a Sequence.

By default, values with *context.output* already containing *filename* are skipped. This can be changed using a keyword argument *overwrite*.

Other allowed keywords are *make_filename*, *make_dirname*, *make_fileext*. Their values must be a tuple, which will initialize a context formatter, or a callable (as returned by format_context). The first item of the tuple is format string, the rest are positional and keyword arguments taken from context during *run* (see *`format_context()`*).

run (*flow*)
Add output parameters to context from the *flow*.

If *`MakeFilename`* works as a Sequence, it transforms all *flow*. In general it should only add values for filename, fileext or dirname in context.output. It is recommended that if context already contains the field, that is not changed. Place more specific formatters first in the sequence.

If *`MakeFilename`* was initialized with keyword arguments, then only those values are transformed, which have no corresponding fields (*filename*, *fileext* and *dirname*) in *context.output* and for which the current context from *flow* could be formatted (contains all necessary keys for the format string).

Note that Sequence takes values with data, while keyword methods take and update only context.

format_context (*format_str*, **args*, ***kwargs*)
Create a function, which formats a given string using a context.

format_str is an ordinary Python format string. *args* are positional and *kwargs* are keyword arguments.

When calling *format_context*, arguments are bound and a new function is returned. When called with a context, its keys are extracted and formatted in *format_str*.

Positional arguments in the *format_str* correspond to *args*, which must be keys in the context. Keys used as positional arguments may be nested (e.g. format_context("{}", "x.y")).

Keyword arguments *kwargs* connect arguments between *format_str* and context. Example:

```
>>> f = format_context("{y}", y="x.y")
>>> f({"x": {"y": 10}})
'10'
```

All keywords in the *format_str* must have corresponding *kwargs*.

Keyword and positional arguments can be mixed. Example:

```
>>> f = format_context("{}_{x}_{y}", "x", x="x", y="y")
>>> f({"x": 1, "y": 2})
'1_1_2'
>>>
```

If no *args* or *kwargs* are given, *kwargs* are extracted from *format_str*. It must contain all non-empty replacement fields, and only simplest formatting without attribute lookup. Example:

```
>>> f = format_context("{x}")
>>> f({"x": 10})
'10'
```

If *format_str* is not a string, `LenaTypeError` is raised. All other errors are raised only during formatting. If context doesn't contain the needed key, `LenaKeyError` is raised. Note that string formatting can also raise a `KeyError` or an `IndexError`, so it is recommended to test your formatters before using them.

2.6 Structures

Histograms:

`Histogram`(edges[, bins, make_bins, . . .])	Multidimensional histogram.
`NumpyHistogram`	

Graph:

`Graph`([points, scale, sort])	Function at given points.

Histogram functions:

`check_edges_increasing`(edges)	Assure that multidimensional *edges* are increasing.
`get_bin_on_value_1d`(val, arr)	Return index for value in one-dimensional array.
`get_bin_on_value`(arg, edges)	Get the bin index for *arg* in a multidimensional array *edges*.
`get_bin_on_index`(index, bins)	Return bin corresponding to multidimensional *index*.
`iter_bins`(bins)	Iterate on *bins*.
`init_bins`(edges[, value, deepcopy])	Initialize cells of the form *edges* with the given *value*.
`integral`(bins, edges)	Compute integral (scale for a histogram).

Continued on next page

Table 20 – continued from previous page

make_hist_context(hist, context)	Update *context* with the context of a `Histogram` *hist*.
unify_1_md(bins, edges)	Unify 1- and multidimensional bins and edges.

2.6.1 Histograms

class Histogram(*edges*, *bins=None*, *make_bins=None*, *initial_value=0*)

Multidimensional histogram.

Arbitrary dimension, variable bin size and a weight function during *fill()* are supported. Lower bin edge is included, upper edge is excluded. Underflow and overflow values are skipped. Bin content type is defined during the initialization.

Examples:

```
>>> # two-dimensional histogram
>>> hist = Histogram([[0, 1, 2], [0, 1, 2]])
>>> hist.fill([0, 1])
>>> hist.bins
[[0, 1], [0, 0]]
>>> values = [[0, 0], [1, 0], [1, 1]]
>>> # use fill method
>>> for v in values:
...     hist.fill(v)
>>> hist.bins
[[1, 1], [1, 1]]
>>> # use as a Lena FillCompute element
>>> # (yielded only after fully computed)
>>> hseq = lena.core.Sequence(hist)
>>> h, context = next(hseq.run(values))
>>> print(h.bins)
[[2, 1], [2, 2]]
```

edges is a sequence of one-dimensional arrays, each containing strictly increasing bin edges. If *edges*' subarrays are not increasing or any of them has length less than 2, `LenaValueError` is raised.

Histogram bins by default are initialized with *initial_value*. It can be any object, which supports addition of a *weight* during *fill* (but that is not necessary if you don't plan to fill the histogram). If the *initial_value* is compound and requires special copying, create initial bins yourself (see `init_bins()`).

Histogram may be created from existing *bins* and *edges*. In this case a simple check of the shape of *bins* is done. If that is incorrect, `LenaValueError` is raised.

make_bins is a function without arguments, which creates new bins (it will be called during `__init__()` and *reset()*). *initial_value* in this case is ignored, but bin check is being done. If both *bins* and *make_bins* are provided, `LenaTypeError` is raised.

Attributes

`Histogram.edges` is a list of edges on each dimension. Edges mark the borders of the bin. Edges along each dimension is a one-dimensional list, and the multidimensional bin is the result of all intersections of one-dimensional edges. For example, 3-dimensional histogram has edges of the form *[x_edges, y_edges, z_edges]*, and the 0th bin has the borders *((x[0], x[1]), (y[0], y[1]), (z[0], z[1]))*.

Index in the edges is a tuple, where a given position corresponds to a dimension, and the content at that position to the bin along that dimension. For example, index *(0, 1, 3)* corresponds to the bin with lower edges *(x[0], y[1], z[3])*.

`Histogram.bins` is a list of nested lists. Same index as for edges can be used to get bin content: bin at *(0, 1, 3)* can be obtained as *bins[0][1][3]*. Most nested arrays correspond to highest (further from x) coordinates. For example, for a 3-dimensional histogram bins equal to *[[[1, 1], [0, 0]], [[0, 0], [0, 0]]]* mean that the only filled bins are those where x and y indices are 0, and z index is 0 and 1.

`dim` is the dimension of a histogram (length of its *edges* for a multidimensional histogram).

Programmer's note

one- and multidimensional histograms have different *bins* and *edges* format. To be unified, 1-dimensional edges should be nested in a list (like *[[1, 2, 3]]*). Instead, they are simply the x-edges list, because it is more intuitive and one-dimensional histograms are used more often. To unify the interface for bins and edges in your code, use `unify_1_md()` function.

compute()

Yield this histogram with context.

fill(*value*, *weight=1*)

Fill histogram with *value* with the given *weight*.

Value can be a *(data, context)* pair. Values outside the histogram edges are ignored.

reset()

Reset the histogram.

Current context is reset to an empty dict. Bins are reinitialized with the *initial_value* or with *make_bins* (depending on the initialization).

If bins were set explicitly during the initialization, `LenaRuntimeError` is raised.

scale(*other=None*, *recompute=False*)

Compute or set scale (integral of the histogram).

If *other* is None, return scale of this histogram. If its scale was not computed before, it is computed and stored for subsequent use (unless explicitly asked to *recompute*).

If a float *other* is provided, rescale to *other*. A new histogram with the scale equal to *other* is returned, the original histogram remains unchanged.

Histograms with scale equal to zero can't be rescaled. `LenaValueError` is raised if one tries to do that.

2.6.2 Graph

class Graph(*points=None*, *scale=None*, *sort=True*)

Function at given points.

Graph can be set during the initialization and during `fill()`. It can be rescaled (producing a new graph).

One can get graph points as `Graph.points` attribute. They will be sorted each time before return if *sort* was set to True. An attempt to change points (use `Graph.points` on the left of '=') will raise Python's `AttributeError`.

Warning: *Graph* does not reduce data. All filled values will be stored in it. To reduce data, use histograms.

points is an array of *(coordinate, value)* tuples.

context will be added to graph context. If it contains "scale", `scale()` method will be available. Otherwise, if "scale" is contained in the context during `fill()`, it will be used. In this case it is assumed that this scale is same for all values (only the last filled context is checked). Context from flow takes precedence over the initialized one.

Graph coordinates are sorted by default. This is usually needed to plot graphs of functions. If you need to keep the order of insertion, set *sort* to False.

By default, sorting is done using standard Python lists and functions. You can disable *sort* and provide your own sorting container for *points*. Some implementations are compared here. Note that a rescaled graph uses a default list.

fill(*value*)
Fill the graph with *value*.

Value can be a *(data, context)* tuple. *Data* part must be a *(coordinates, value)* pair, where both coordinates and value are also tuples. For example, *value* can contain the principal number and the precision.

points
Get graph points (read only).

request()
Yield graph with context.

If *sort* was initialized True, graph points will be sorted. If flow contained *scale* it the context, it is set now.

reset()
Reset points to an empty list and current context to an empty dict.

scale(*other=None*)
Get or set the scale.

Graph's scale comes from an external source. For example, if the graph was computed from a function, this may be its integral passed via context during `fill()`. Once the scale is set, it is stored in the graph. If one attempts to use scale which was not set, `LenaAttributeError` is raised.

If *other* is None, return the scale.

If a `float` *other* is provided, rescale to *other*. A new graph with the scale equal to *other* is returned, the original one remains unchanged. Note that in this case its *points* will be a simple list and new graph *sort* parameter will be True.

Graphs with scale equal to zero can't be rescaled. Attempts to do that raise `LenaValueError`.

to_csv(*separator=', ', header=None*)
Convert graph's points to CSV.

separator delimits values, default is a comma.

header, if not `None`, is the first string of the output (new line is added automatically).

Since a graph can be multidimensional, for each point first its coordinate is converted to string (separated by *separator*), than each part of its value.

To convert `Graph` to CSV inside a Lena sequence, use `ToCSV`.

2.6.3 Histogram functions

Functions for histograms.

These functions are used for low-level work with histograms and their contents. They are not needed for normal usage.

check_edges_increasing(*edges*)

Assure that multidimensional *edges* are increasing.

If length of *edges* or its subarray is less than 2 or if some subarray of *edges* contains not strictly increasing values, `LenaValueError` is raised.

get_bin_on_index(*index*, *bins*)

Return bin corresponding to multidimensional *index*.

index can be a number or a list/tuple. If *index* length is less than dimension of *bins*, a subarray of *bins* is returned.

In case of an index error, `LenaIndexError` is raised.

Example:

```
>>> from lena.structures import Histogram, get_bin_on_index
>>> hist = Histogram([0, 1], [0])
>>> get_bin_on_index(0, hist.bins)
0
>>> get_bin_on_index((0, 1), [[0, 1], [0, 0]])
1
>>> get_bin_on_index(0, [[0, 1], [0, 0]])
[0, 1]
```

get_bin_on_value(*arg*, *edges*)

Get the bin index for *arg* in a multidimensional array *edges*.

arg is a 1-dimensional array of numbers (or a number for 1-dimensional *edges*), and corresponds to a point in N-dimensional space.

edges is an array of N-1 dimensional arrays (lists or tuples) of numbers. Each 1-dimensional subarray consists of increasing numbers.

arg and *edges* must have the same length (otherwise `LenaValueError` is raised). *arg* and *edges* must be iterable and support *len()*.

Return list of indices in *edges* corresponding to *arg*.

If any coordinate is out of its corresponding edge range, its index will be `-1` for underflow or `len(edge)-1` for overflow.

Examples:

```
>>> from lena.structures import get_bin_on_value
>>> edges = [[1, 2, 3], [1, 3.5]]
>>> get_bin_on_value((1.5, 2), edges)
[0, 0]
>>> get_bin_on_value((1.5, 0), edges)
[0, -1]
>>> # the upper edge is excluded
>>> get_bin_on_value((3, 2), edges)
[2, 0]
>>> # one-dimensional edges
>>> edges = [1, 2, 3]
>>> get_bin_on_value(2, edges)
[1]
```

get_bin_on_value_1d(*val*, *arr*)

Return index for value in one-dimensional array.

arr must contain strictly increasing values (not necessarily equidistant), it is not checked.

"Linear binary search" is used, that is our array search by default assumes the array to be split on equidistant steps.

Example:

```
>>> from lena.structures import get_bin_on_value_1d
>>> arr = [0, 1, 4, 5, 7, 10]
>>> get_bin_on_value_1d(0, arr)
0
>>> get_bin_on_value_1d(4.5, arr)
2
>>> # upper range is excluded
>>> get_bin_on_value_1d(10, arr)
5
>>> # underflow
>>> get_bin_on_value_1d(-10, arr)
-1
```

hist_to_graph(*hist*, *context*, *make_graph_value=None*, *bin_coord='left'*)
Convert a *Histogram* *hist* to a Graph.

context becomes graph's context. For example, it can contain a scale.

make_graph_value is a function to set graph point's value. By default it is bin content. This option could be used to create graph error bars. *make_graph_value* must accept bin content and bin context as positional arguments.

bin_coord signifies which will be the coordinate of a graph's point created from histogram's bin. Can be "left" (default), "right" and "middle".

Return the resulting graph.

init_bins(*edges*, *value=0*, *deepcopy=False*)
Initialize cells of the form *edges* with the given *value*.

Return bins filled with copies of *value*.

Value must be copyable, usual numbers will suit. If the value is mutable, use *deepcopy* = True (or the content of cells will be identical).

Examples:

```
>>> edges = [[0, 1], [0, 1]]
>>> # one cell
>>> init_bins(edges)
[[0]]
>>> # no need to use floats,
>>> # because integers will automatically be cast to floats
>>> # when used together
>>> init_bins(edges, 0.0)
[[0.0]]
>>> init_bins([[0, 1, 2], [0, 1, 2]])
[[0, 0], [0, 0]]
>>> init_bins([0, 1, 2])
[0, 0]
```

integral(*bins*, *edges*)
Compute integral (scale for a histogram).

bins contain values, and *edges* form the mesh for the integration. Their format is defined in Histogram description.

iter_bins(*bins*)
Iterate on *bins*. Yield *(index, bin content)*.

Edges with higher index are iterated first (that is z, then y, then x for a 3-dimensional histogram).

make_hist_context(*hist*, *context*)
Update *context* with the context of a `Histogram` *hist*.

Deep copy of updated context is returned.

unify_1_md(*bins*, *edges*)
Unify 1- and multidimensional bins and edges.

Return a tuple of *(bins, edges)*. Bins and multidimensional *edges* return unchanged, while one-dimensional *edges* are inserted into a list.

2.7 Variables

Variables:

`Combine`(*args, **kwargs)	Combine variables into a tuple.
`Compose`(*args, **kwargs)	Composition of variables.
`Variable`(name, getter, **kwargs)	Function of data with context.

2.7.1 Variables

Variables are functions to transform data and add context.

A variable can represent a particle type, a coordinate, etc. They transform raw input data into Lena data with context. Variables have name and may have other attributes like LaTeX name, dimension or unit.

Variables can be composed using `Compose`, which corresponds to function composition.

Variables can be combined into multidimensional variables using `Combine`.

Examples:

```
>>> from lena.variables import Variable, Compose
>>> # data is pairs of (positron, neutron) coordinates
>>> data = [((1.05, 0.98, 0.8), (1.1, 1.1, 1.3))]
>>> x = Variable(
...    "x", lambda coord: coord[0], type="coordinate"
... )
>>> neutron = Variable(
...    "neutron", latex_name="n",
...    getter=lambda double_ev: double_ev[1], type="particle"
... )
>>> x_n = Compose(neutron, x)
>>> x_n(data[0])[0]
1.1
>>> x_n(data[0])[1] == {
...     'variable': {
...         'name': 'neutron_x', 'particle': 'neutron',
...         'latex_name': 'x_{n}', 'coordinate': 'x', 'type': 'coordinate',
...         'compose': {
...             'type': 'particle', 'latex_name': 'n',
```

(continues on next page)

(continued from previous page)

```
...             'name': 'neutron', 'particle': 'neutron'
...         },
...     }
... }
True
```

Combine and *Compose* are subclasses of a *Variable*.

class Combine(**args*, ***kwargs*)
Combine variables into a tuple.

Combine(var1, var2, . . .)(value) is *((var1.getter(value), var2.getter(value), . . .), context)*.

args are the variables to be combined.

Keyword arguments are passed to *Variable*'s __init__. For example, *name* is the name of the combined variable. If not provided, it is its variables' names joined with '_'.

context.variable is updated with *combine*, which is a tuple of each variable's context.

Attributes:

dim is the number of variables.

All *args* must be *Variables* and there must be at least one of them, otherwise `LenaTypeError` is raised.

__getitem__(*index*)
Get variable at the given *index*.

class Compose(**args*, ***kwargs*)
Composition of variables.

args are the variables to be composed.

Keyword arguments:

name is the name of the composed variable. If that is missing, it is composed from variables names joined with underscore.

latex_name is LaTeX name of the composed variable. If that is missing and if there are only two variables, it is composed from variables' names (or their LaTeX names if present) as a subscript in the reverse order *(latex2_{latex1})*.

context.variable.compose contains contexts of the composed variables (the first composed variable is most nested).

If any keyword argument is a callable, it is used to create the corresponding variable attribute. In this case, all variables must have this attribute, and the callable is applied to the list of these attributes. If any attribute is missing, `LenaAttributeError` is raised. This can be used to create composed attributes other than *latex_name*.

If there are no variables or if *kwargs* contain *getter*, `LenaTypeError` is raised.

class Variable(*name*, *getter*, ***kwargs*)
Function of data with context.

name is variable's name.

getter is the python function (not a *Variable*) that performs the actual transformation of data. It must accept data and return data without context.

Other variable's attributes can be passed as keyword arguments. Examples include *latex_name*, *unit* (like *cm* or *keV*), *range*, etc.

type is the type of the variable. It depends on your application, examples are 'coordinate' or 'particle_type'. It has a special meaning: if present, its value is added to variable's context as a key with variable's name (see example for this module). Thus variable type's data is preserved during composition of different types.

Attributes

getter is the function that does the actual data transformation.

var_context is the dictionary of attributes of the variable, which is added to *context.variable* during `__call__()`.

All public attributes of a variable can be accessed using dot notation (for example, *var.var_context["latex_name"]* can be simply *var.latex_name*). `AttributeError` is raised if the attribute is missing.

If *getter* is a `Variable` or is not callable, `LenaTypeError` is raised.

`__call__`(*value*)

Transform a *value*.

Data part of the value is transformed by the *getter*. *Context.variable* is updated with the context of this variable (or created if missing).

If context already contained *variable*, it is preserved as *context.variable.compose* subcontext.

Return *(data, context)*.

`get`(*key*, *default=None*)

Return the attribute *key* if present, else default.

Key can be a dot-separated string, a list or a dictionary (see `lena.context.get_recursively()`).

If default is not given, it defaults to `None`, so that this method never raises a *KeyError*.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

CHAPTER 4

Installation

4.1 From pip

Lena core modules depend only on Python standard libraries. Other python extensions can be installed from pip:

```
pip install lena
# if you plan to render LaTeX templates
pip install jinja2
```

4.2 From github

```
git clone https://github.com/ynikitenko/lena
# most of requirements are for development only
pip install -r lena/requirements.txt
```

Replace <*path-to-lena*> with the actual path to the directory *lena* and add

```
export PYTHONPATH=$PYTHONPATH:<path-to-lena>
```

to your profile (e.g. `.profile` or `.bashrc` on Linux).

4.3 Additional programs

To fully use all available tools, you may need the following programs:

- *pdflatex* to produce pdf files from LaTeX.
- *pgfplots* and *TikZ* to produce LaTeX plots.
- *pdftoppm* to convert pdf files to png.

They are not necessary if you don't need to make plots or want to provide your own tools for that.

CHAPTER 5

Documentation

To get started, read the *Tutorial*.

Complete documentation on Lena classes and specific topics can be found in the *Reference*.

CHAPTER 6

License

Lena is free software licensed under Apache software license (version 2). You can use it freely for your data analysis, read its source code and modify it.

It is intended to help people in their data analysis, but we don't take responsibility if something goes wrong.

CHAPTER 7

Alternatives

Ruffus is a Computation Pipeline library for python used in science and bioinformatics. It connects program components by writing and reading files.

Python Module Index

l

lena.context.context, 29
lena.core, 32
lena.core.adapters, 35
lena.core.exceptions, 37
lena.flow, 39
lena.flow.functions, 40
lena.flow.group_plots, 41
lena.flow.iterators, 43
lena.flow.split_into_bins, 43
lena.math.elements, 47
lena.math.meshes, 46
lena.math.utils, 46
lena.math.vector3, 48
lena.output, 52
lena.output.make_filename, 54
lena.structures, 56
lena.structures.graph, 57
lena.structures.hist_functions, 58
lena.variables.variable, 61

Index

Symbols

__call__() (*Call method*), 35
__call__() (*Chain method*), 43
__call__() (*Context method*), 30
__call__() (*CountFrom method*), 43
__call__() (*Selector method*), 42
__call__() (*Source method*), 33
__call__() (*SourceEl method*), 37
__call__() (*Variable method*), 63
__getitem__() (*Combine method*), 62

A

alter_sequence() (*Cache static method*), 40
angle() (*vector3 method*), 49

C

Cache (*class in lena.flow*), 39
cache_exists() (*Cache method*), 40
Call (*class in lena.core.adapters*), 35
cell_to_string() (*in module lena.flow.split_into_bins*), 45
Chain (*class in lena.flow.iterators*), 43
check_edges_increasing() (*in module lena.structures.hist_functions*), 58
clear() (*GroupBy method*), 41
clip() (*in module lena.math.utils*), 46
Combine (*class in lena.variables.variable*), 62
Compose (*class in lena.variables.variable*), 62
compute() (*FillCompute method*), 35
compute() (*FillComputeSeq method*), 33
compute() (*Histogram method*), 57
compute() (*Mean method*), 47
compute() (*SplitIntoBins method*), 44
compute() (*Sum method*), 47
Context (*class in lena.context.context*), 29
cosine() (*vector3 method*), 50
CountFrom (*class in lena.flow.iterators*), 43
cross() (*vector3 method*), 50

D

difference() (*in module lena.context.functions*), 30
dot() (*vector3 method*), 50
drop_cache() (*Cache method*), 40
DropContext (*class in lena.flow*), 40

E

End (*class in lena.flow*), 40

F

fill() (*FillCompute method*), 35
fill() (*FillComputeSeq method*), 33
fill() (*FillRequest method*), 36
fill() (*FillRequestSeq method*), 34
fill() (*Graph method*), 58
fill() (*Histogram method*), 57
fill() (*Mean method*), 47
fill() (*SplitIntoBins method*), 44
fill() (*Sum method*), 48
fill_into() (*FillInto method*), 36
fill_into() (*ISlice method*), 43
FillCompute (*class in lena.core.adapters*), 35
FillComputeSeq (*class in lena.core*), 33
FillInto (*class in lena.core.adapters*), 35
FillRequest (*class in lena.core.adapters*), 36
FillRequestSeq (*class in lena.core*), 33
format_context() (*in module lena.output.make_filename*), 54
fromspherical() (*lena.math.vector3.vector3 class method*), 49, 50

G

get() (*Variable method*), 63
get_bin_on_index() (*in module lena.structures.hist_functions*), 59
get_bin_on_value() (*in module lena.structures.hist_functions*), 59
get_bin_on_value_1d() (*in module lena.structures.hist_functions*), 59

get_context() (in module lena.flow.functions), 40
get_data() (in module lena.flow.functions), 40
get_data_context() (in module lena.flow.functions), 41
get_example_bin() (in module lena.flow.split_into_bins), 45
get_recursively() (in module lena.context.functions), 30
Graph (class in lena.structures.graph), 57
GroupBy (class in lena.flow), 41
GroupPlots (class in lena.flow), 41
GroupScale (class in lena.flow), 42

H

hist1d_to_csv() (in module lena.output), 53
hist2d_to_csv() (in module lena.output), 53
hist_to_graph() (in module lena.structures.hist_functions), 60
Histogram (class in lena.structures), 56
HistToCSV (class in lena.output), 53

I

init_bins() (in module lena.structures.hist_functions), 60
integral() (in module lena.structures.hist_functions), 60
intersection() (in module lena.context.functions), 30
isclose() (in module lena.math.utils), 47
isclose() (vector3 method), 50
ISlice (class in lena.flow.iterators), 43
iter_bins() (in module lena.structures.hist_functions), 60

L

LaTeXToPDF (class in lena.output), 53
lena.context.context (module), 29
lena.core (module), 32
lena.core.adapters (module), 35
lena.core.exceptions (module), 37
lena.flow (module), 39
lena.flow.functions (module), 40
lena.flow.group_plots (module), 41
lena.flow.iterators (module), 43
lena.flow.split_into_bins (module), 43
lena.math.elements (module), 47
lena.math.meshes (module), 46
lena.math.utils (module), 46
lena.math.vector3 (module), 48
lena.output (module), 52
lena.output.make_filename (module), 54
lena.structures (module), 56
lena.structures.graph (module), 57
lena.structures.hist_functions (module), 58
lena.variables.variable (module), 61
LenaAttributeError, 37
LenaEnvironmentError, 37
LenaException, 37
LenaIndexError, 38
LenaKeyError, 38
LenaNotImplementedError, 38
LenaRuntimeError, 38
LenaStopFill, 38
LenaTypeError, 38
LenaValueError, 38
LenaZeroDivisionError, 38

M

make_hist_context() (in module lena.structures.hist_functions), 61
MakeFilename (class in lena.output.make_filename), 54
md_map() (in module lena.math.meshes), 46
Mean (class in lena.math.elements), 47
mesh() (in module lena.math.meshes), 46

N

norm() (vector3 method), 51

P

PDFToPNG (class in lena.output), 52
points (Graph attribute), 58
Print (class in lena.flow), 40
proj() (vector3 method), 51

R

ReduceBinContent (class in lena.flow.split_into_bins), 43
RenderLaTeX (class in lena.output), 54
request() (FillRequest method), 36
request() (FillRequestSeq method), 34
request() (Graph method), 58
reset() (FillRequest method), 36
reset() (FillRequestSeq method), 34
reset() (Graph method), 58
reset() (Histogram method), 57
reset() (Mean method), 47
reset() (Sum method), 48
rotate() (vector3 method), 51
Run (class in lena.core.adapters), 37
run() (Cache method), 40
run() (DropContext method), 40
run() (End method), 40
run() (FillRequest method), 36
run() (GroupPlots method), 42
run() (ISlice method), 43

run() (*LaTeXToPDF method*), 53
run() (*MakeFilename method*), 54
run() (*PDFToPNG method*), 52
run() (*ReduceBinContent method*), 44
run() (*RenderLaTeX method*), 54
run() (*Run method*), 37
run() (*Sequence method*), 33
run() (*Split method*), 34
run() (*ToCSV method*), 53
run() (*Writer method*), 52

S

scalar_proj() (*vector3 method*), 51
scale() (*Graph method*), 58
scale() (*GroupScale method*), 42
scale() (*Histogram method*), 57
Selector (*class in lena.flow*), 42
seq_map() (*in module lena.flow.functions*), 41
Sequence (*class in lena.core*), 32
Source (*class in lena.core*), 33
SourceEl (*class in lena.core.adapters*), 37
Split (*class in lena.core*), 34
SplitIntoBins (*class in lena.flow.split_into_bins*), 44
str_to_dict() (*in module lena.context.functions*), 31
Sum (*class in lena.math.elements*), 47

T

to_csv() (*Graph method*), 58
ToCSV (*class in lena.output*), 53
TransformBins (*class in lena.flow.split_into_bins*), 44

U

unify_1_md() (*in module lena.structures.hist_functions*), 61
update() (*GroupBy method*), 41
update_nested() (*in module lena.context.functions*), 31
update_recursively() (*in module lena.context.functions*), 31

V

Variable (*class in lena.variables.variable*), 62
vector3 (*class in lena.math.vector3*), 48

W

Writer (*class in lena.output*), 52