Lena Documentation

Release 0.6-beta

Yaroslav Nikitenko

Apr 16, 2024

CONTENTS:

Lena is an architectural framework for data analysis. It is written in a popular programming language Python and works with Python versions 2, 3 and PyPy (2 and 3).

Lena features from programming point of view include:

- simple and powerful programming language.
- modularity, weak coupling. Algorithms can be easily added, replaced or reused.
- 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 (to check the code). Results of heavy calculations can be easily saved.
- 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.
- 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 positions in the detector).
- analysis consistency. When we run several algorithms for same data or reuse an algorithm, we are confident that we use same data and algorithm.
- algorithms can be combined into a more complex analysis.

Lena was created in experimental neutrino physics and is named after a great Siberian river.

CHAPTER ONE

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

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 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
```

The first line imports a *Lena* class *Sequence*. 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, Slice

s = Sequence(
    lambda i: pow(-1, i) * (2 * i + 1),
)
spi = Source(
    CountFrom(0),
    s,
    Slice(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*).

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 a *Slice* element. *Slice* and *CountFrom* are similar to *islice* and *count* functions from Python's standard library module *itertools*. *Slice* 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*.

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

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}
```

(continues on next page)

(continued from previous page)

```
\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 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.
```

(continues on next page)

[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.

(continued from previous page)

```
        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

```
import os

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

from read_data import ReadData


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

(continues on next page)

(continued from previous page)

```
        lambda dt: (dt[0][0], dt[1]),
        Histogram(mesh((-10, 10), 10)),
        ToCSV(),
        MakeFilename("x"),
        Write("output"),
        RenderLaTeX("histogram_1d.tex"),
        Write("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*.

Write 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(),
    Slice(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.

Slice, 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

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(
    ReadData(),
    Split([
        (
            lambda vec: vec[0],
            Histogram(mesh((-10, 10), 10)),
            ToCSV(),
            Write("output", "x"),
        ),
        (
            lambda vec: vec[1],
            Histogram(mesh((-10, 10), 10)),
            ToCSV(),
            Write("output", "y"),
        ),
    ]),
    RenderLaTeX("histogram_1d.tex", "templates"),
    Write("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 *Write* (so that they could be distinguished in the following flow).

Write has two initialization parameters: the default directory and the default file name. Write 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 *Write*'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 Write objects write CSV data to *output/x.csv* and *output/y.csv*.

For each file written, *Write* 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 *writes* to do that. However, we can move *ToCSV* and *Write* outside the *Split* (and make our code one line shorter):

Listing 3: tutorial/2_split/main2.py

```
from lena.output import MakeFilename
s = Sequence(
    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(),
    Write("output"),
    # ... as earlier ...
)
```

Element *MakeFilename* adds file name to *context.output*. *Write* 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")
    write = Write("output")
    s = Sequence(
        ReadData(),
        Split([
            (
                Variable("x", lambda vec: vec[0]),
                Histogram(mesh((-10, 10), 10)),
            ),
            (
                Variable("y", lambda vec: vec[1]),
```

(continues on next page)

(continued from previous page)

```
                Histogram(mesh((-10, 10), 10)),
            ),
            (
                Variable("z", lambda vec: vec[2]),
                Histogram(mesh((-10, 10), 10)),
            ),
        ]),
        MakeFilename("{{variable.name}}"),
        ToCSV(),
        write,
        RenderLaTeX("histogram_1d.tex", "templates"),
        write,
        LaTeXToPDF(),
        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 *Writes* 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}

\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

```
import os

import lena.context
import lena.flow
from lena.core import Sequence, Split, Source
from lena.structures import Histogram
```

(continues on next page)

(continued from previous page)

```
from lena.math import mesh
from lena.output import ToCSV, Write, 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",
    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")
    write = Write("output")
    s = Sequence(
        ReadDoubleEvents(),
        Split(
            coordinates_1d
            +
            [(
                particle,
                Combine(x, y, name="xy"),
```

(continues on next page)

(continued from previous page)

```
                Histogram(mesh(((-10, 10), (-10, 10)), (10, 10))),
                MakeFilename("{{variable.particle}}/{{variable.name}}"),
             )
             for particle in (positron, neutron)
            ]
        ),
        MakeFilename("{{variable.particle}}/{{variable.coordinate}}"),
        ToCSV(),
        write,
        RenderLaTeX(select_template, template_dir="templates"),
        write,
        LaTeXToPDF(),
        PDFToPNG(),
    )
    results = s.run([data_file])
    for res in results:
        print(res)


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*:

```
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")*),

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

- 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 ...
```

(continues on next page)

(continued from previous page)

```
            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 (*Write* 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].

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*

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

[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

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.

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?

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*.

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
...
>>> 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 = {}
```

(continues on next page)

(continued from previous page)

```
    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"):
    write = lena.output.Write(output_dir)
    s = lena.core.Sequence(
        lena.output.ToCSV(),
        write,
        lena.context.Context(),
        lena.output.RenderLaTeX(),  # initialize properly here
        write,
        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

The author is unaware of a simple for a user way to stop a function and resume it at the given point. Inform the author if you know better answers to any of these exercises.

Mikhail Zelenyi gives this explanation (translated from Russian):

There are two types of models: push and pull. If you have a sequence, then in the case of a *push* model the calculations are initiated by the first member of the sequence, which pushes data further. In this case a fork can be done easily, just at a certain moment it pushes data not into one sequence, but into two.

In the case of a *pull* model the calculations are initiated by the last member of the sequence. Consequently, if we want to branch the sequence, we need to think what to do: to start only when all consumers asked, to use a buffer, or to start with one consumer and to push data into the others conforming to the *push* model.

1.3.2 Part 2

Ex. 1

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

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

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

(continues on next page)

(continued from previous page)

```
        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

TWO

REFERENCE

2.1 Context

Elements:

`Context`([d, formatter])	Dictionary with easy-to-read formatting.
`UpdateContext`(subcontext, update[, value, ...])	Update context of passing values.

Functions:

`contains`(d, s)	Check that a dictionary *d* contains a subdictionary defined by a string *s*.
`difference`(d1, d2[, level])	Return a dictionary with items from *d1* not contained in *d2*.
`format_context`(format_str)	Create a function that formats a context using the given string.
`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[, value])	Create a dictionary from a dot-separated string *s*.
`str_to_list`(s)	Like `str_to_dict()`, but return a flat list.
`to_string`(d)	Convert a dictionary *d* to a string.
`update_nested`(key, d, other)	Update *d[key]* with the *other* dictionary preserving data.
`update_recursively`(d, other[, value])	Update dictionary *d* with items from *other* dictionary.

2.1.1 Elements

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

Bases: `dict`

Dictionary with easy-to-read formatting.

`Context` provides a better representation for context. Example:

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

(continues on next page)

(continued from previous page)

```
    "2": {
        "3": 4
    }
}
```

Initialize from a dictionary *d* (empty by default).

Representation is defined by the *formatter*. That must be a callable accepting a dictionary and returning a string. The default is `json.dumps`.

All public attributes of a `Context` can be retrieved or set using dot notation (for example, *context["data_path"]* is equal to *context.data_path*). Only one level of nesting is accessible using dot notation.

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

If the attribute to be retrieved is missing, `LenaAttributeError` is raised. An attempt to access a private attribute raises `AttributeError`.

`__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`.

When a `Context` is used as a sequence element, its initialization argument *d* has no effect on the produced values.

class **UpdateContext**(*subcontext*, *update*, *value=False*, *default=<object object>*, *skip_on_missing=False*, *raise_on_missing=False*, *recursively=True*)

Update context of passing values.

subcontext is a string representing the part of context to be updated (for example, *"output.plot"*). *subcontext* must be non-empty.

update will become the value of *subcontext* during `__call__()`. It can be one of three different types:

- a simple value (not a string),
- a context formatting string,
- a context value (a string in curly braces).

A context formatting string is any string with arguments enclosed in double braces (for example, *"{{variable.type}}_{{variable.name}}"*). Its argument values will be filled from context during `__call__()`. If a formatting argument is missing in context, it will be substituted with an empty string.

To set *update* to a value from context (not a string), the keyword argument *value* must be set to `True` and the *update* format string must be a non-empty single expression in double braces (*"{{variable.compose}}"*).

If *update* corresponds to a context value and a formatting argument is missing in the context, `LenaKeyError` will be raised unless a *default* is set. In this case *default* will be used for the update value.

If *update* is a context formatting string, *default* keyword argument can't be used. To set a default value other than an empty string, use a jinja2 filter. For example, if *update* is *"{{variable.name|default('x')}}"*, then *update* will be set to "x" both if *context.variable.name* is missing and if *context.variable* is missing itself.

Other variants to deal with missing context values are:

- to skip update (don't change the context), set by *skip_on_missing*, or

- to raise `LenaKeyError` (set by *raise_on_missing*).

Only one of *default*, *skip_on_missing* or *raise_on_missing* can be set, otherwise `LenaValueError` is raised. None of these options can be used if *update* is a simple value.

If *recursively* is `True` (default), not overwritten existing values of *subcontext* are preserved. Otherwise, all existing values of *subcontext* (at its lowest level) are removed. See also `update_recursively()`.

Example:

```
>>> from lena.context import UpdateContext
>>> make_scatter = UpdateContext("output.plot", {"scatter": True})
>>> # call directly
>>> make_scatter(((0, 0), {}))
((0, 0), {'output': {'plot': {'scatter': True}}})
>>> # or use in a sequence
```

If *subcontext* is not a string, `LenaTypeError` is raised. If it is empty, `LenaValueError` is raised. If *value* is `True`, braces can be only the first two and the last two symbols of *update*, otherwise `LenaValueError` is raised.

`__call__`(*value*)

Update *value*'s context.

If the *value* is updated, *subcontext* is always created (also if the *value* contains no context).

`LenaKeyError` is raised if *raise_on_missing* is `True` and the update argument is missing in *value*'s context.

2.1.2 Functions

`contains`(*d*, *s*)

Check that a dictionary *d* contains a subdictionary defined by a string *s*.

True if *d* contains a subdictionary that is represented by *s*. Dots in *s* mean nested subdictionaries. A string without dots means a key in *d*.

Example:

```
>>> d = {'fit': {'coordinate': 'x'}}
>>> contains(d, "fit")
True
>>> contains(d, "fit.coordinate.x")
True
>>> contains(d, "fit.coordinate.y")
False
```

If the most nested element of *d* to be compared with *s* is not a string, its string representation is used for comparison. See also `str_to_dict()`.

`difference`(*d1*, *d2*, *level=-1*)

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

level sets the maximum depth of recursion. For infinite recursion, set that to -1. For level 1, if a key is present both in *d1* and *d2* but has different values, it is included into the difference. See `intersection()` for more details.

d1 and *d2* remain unchanged. However, *d1* or some of its subdictionaries may be returned directly. Make a deep copy of the result when appropriate.

New in version 0.5: add keyword argument *level*.

format_context(*format_str*)

Create a function that formats a context using the given string.

It is recommended to use jinja2.Template. Use this function only if you don't have jinja2.

format_str is a Python format string with double braces instead of single ones. It must contain all non-empty replacement fields, and only simplest formatting without attribute lookup. Example:

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

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*.

Keys can be nested using a dot, for example:

```
>>> f = format_context("{{x.y}}_{{z}}")
>>> f({"x": {"y": 10}, "z": 1})
'10_1'
```

This function does not work with unbalanced braces. If a simple check fails, `LenaValueError` is raised. 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 `ValueError`, so it is recommended to test your formatters before using them.

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.

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'
```

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* is not a string, a dict or a list, `LenaTypeError` is raised. If *keys* is a dictionary with more than one key at some level, `LenaValueError` is raised.

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*, *value=<object object>*)

Create a dictionary from a dot-separated string *s*.

If the *value* is provided, it becomes the value of the deepest key represented by *s*.

Dots represent nested dictionaries. If *s* is non-empty and *value* is not provided, then *s* must have at least two dot-separated parts (*"a.b"*), otherwise `LenaValueError` is raised. If a *value* is provided, *s* must be non-empty.

If *s* is empty, an empty dictionary is returned.

Examples:

```
>>> str_to_dict("a.b.c d")
{'a': {'b': 'c d'}}
>>> str_to_dict("output.changed", True)
{'output': {'changed': True}}
```

str_to_list(*s*)

Like `str_to_dict()`, but return a flat list.

If the string *s* is empty, an empty list is returned. This is different from *str.split*: the latter would return a list with one empty string. Contrarily to `str_to_dict()`, this function allows an arbitrary number of dots in *s* (or none).

to_string(*d*)

Convert a dictionary *d* to a string.

Example:

```
>>> d = {"a": 1, "b": {"c": 3}}
>>> to_string(d)
'{"a":1,"b":{"c":3}}'
```

d can have nested subdictionaries, lists and other JSON-serializable items. *d* keys are sorted.

Note: The returned representation is terse and can be used for hashing (though more optimal solutions for that may exist). *d* can be not only a dictionary, but for example to hash a list one can simply convert it to a tuple. Use `Context` for a human-friendlier formatting. Use `json.dumps` for more flexibility.

If an item is unserializable (for example, *d* contains a *set*), `LenaValueError` is raised.

New in version 0.6.

update_nested(*key*, *d*, *other*)

Update *d[key]* with the *other* dictionary preserving data.

If *d* doesn't contain the *key*, it is updated with *{key: other}*. If *d* contains the *key*, *d[key]* is inserted into *other[key]* (so that it is not overriden). If *other* contains *key* (and possibly more nested *key*-s), then *d[key]* is inserted into the deepest level of *other.key.key*... Finally, *d[key]* becomes *other*.

Example:

```
>>> context = {"variable": {"name": "x"}}
>>> new_var_context = {"name": "n"}
>>> update_nested("variable", context, copy.deepcopy(new_var_context))
>>> context == {'variable': {'name': 'n', 'variable': {'name': 'x'}}}
True
>>>
>>> update_nested("variable", context, {"name": "top"})
>>> context == {
...    'variable': {'name': 'top',
...                  'variable': {'name': 'n', 'variable': {'name': 'x'}}}
... }
True
```

other is modified in general. Create that on the fly or use *copy.deepcopy* when appropriate.

Recursive dictionaries (containing references to themselves) are strongly discouraged and meaningless when nesting. If *other[key]* is recursive, `LenaValueError` may be raised.

update_recursively(*d*, *other*, *value=<object object>*)

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 and the *value* to a dictionary. A *value* argument is allowed only when *other* is a string, otherwise `LenaValueError` is raised.

Existing values are updated recursively, that is including nested subdictionaries. 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
```

2.2 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)	Deprecated since version 0.6.
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[, bufsize, reset, ...])	Deprecated since version 0.6.
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	Incorrect type.
LenaValueError	Wrong value.

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 that transforms the incoming flow.

If this `Sequence` is empty, the flow passes unaltered, but 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. It can be an an object with a generator function *__call__()* or an iterable. Following arguments (if present) form a sequence of elements, each accepting computational flow from the previous element.

```
>>> from lena.flow import CountFrom, Slice
>>> s = Source(CountFrom(), Slice(5))
>>> # iterate in a cycle
>>> for i in s():
...     if i == 5:
...         break
...     print(i, end=" ")
0 1 2 3 4
>>> # if called twice, results depend on the generator
>>> list(s()) == list(range(5, 10))
True
```

For a *sequence* that 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*)

Deprecated since version 0.6: inside a `Split` element this sequence is a subtype of a simple `Sequence`.

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 elements that follow it.

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* or *reset*. See `FillRequest` for more information on them. By default *bufsize* is *1*.

If *FillRequest* element was not found, the sequences could not be correctly initialized, or unknown keyword arguments were received, `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. 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`.

copy_buf sets whether the buffer should be copied during `run()`. This is important if different sequences can change input data and thus 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__*). Common type is not implemented for *Call* element.

In case of wrong initialization arguments, *LenaTypeError* or *LenaValueError* is raised.

__call__()

Each initialization sequence generates flow. After its flow is empty, next sequence is called, etc.

This method is available only if each self sequence is a *Source*, otherwise runtime *LenaAttributeError* is raised.

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*, *bufsize=1*, *reset=None*, *buffer_input=None*, *buffer_output=None*, *yield_on_remainder=False*, *fill='fill'*, *request='request'*, *reset_name='reset'*)

Deprecated since version 0.6: inside `Split` this element can be implemented by a simple `Run` element.

Adapter for a *FillRequest* element.

A *FillRequest* element slices the flow during *fill* and yields results for each chunk during *request*. It can also call *reset* after each *request*.

Names for actual *fill*, *request* and *reset* methods can be provided during initialization (the latter is set through *reset_name*).

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

FillRequest can also be initialized from a *Run* element. In that case *el* is not required to have *fill*, *compute* or *reset* methods (and *FillRequest* will not have such missing methods). *FillRequest* implements *run* method that splits the flow into subslices of *bufsize* values and feeds them to the *run* method of *el*.

Since we require no less than *bufsize* values (except *yield_on_remainder* is `True`), we need to store either *bufsize* values of the incoming flow or all values produced by *el.run* or *el.request* for each slice. This is set by *buffer_input* or *buffer_output*. One and only one of them must be `True`. For example, if the element receives file names and produces data from them, it would be wise to buffer input. If the element receives much data and produces a histogram, one should buffer output.

If a keyword argument *reset* is `True`, *el* must have a method *reset_name*, and in this case `reset()` is called after each `request()` (including those during `run()`). In general, *Run* elements have no *reset* methods, but for *FillCompute* elements *reset* must be set explicitly.

If *yield_on_remainder* is `True`, then the output will be yielded even if the element was filled less than *bufsize* times (but at least once). In that case no internal buffers are used during `run()` and corresponding attributes are not checked.

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.

Changed in version 0.5: add keyword arguments *yield_on_remainder*, *buffer_input*, *buffer_output*, *reset_name*. Require explicit *reset* for *FillCompute* elements.

fill(*value*)

Fill *el* with *value*.

If more than *bufsize* values were filled, incoming values are stored in a buffer (if *buffer_input* is `True`) or, otherwise, the output of *el.request* is stored in a buffer, until it is requested.

request()

Yield results (if they are available) and possibly reset.

If input or output buffers were filled, all their contents are processed and yielded.

reset()

Reset *el* (ignoring the initialization setting).

run(*flow*)

Process the *flow* slice by slice.

fill each value from a subslice of *flow* of *bufsize* length, then yield results from *request*. Repeat until the *flow* is exhausted.

If *fill* was not called even once (*flow* was empty), nothing is yielded, because *bufsize* values were not obtained (in contrast to *FillCompute*, for which output for an empty flow is reasonable). The last slice may contain less than *bufsize* values. If there were any and if *yield_on_remainder* is `True`, *request* will be called for that.

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 values 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 be callable or iterable, or contain a callable method *call*.

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: `LenaException`, `AttributeError`

exception LenaEnvironmentError

Bases: `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: `LenaException`, `IndexError`

exception LenaKeyError

Bases: *LenaException*, `KeyError`

exception LenaNotImplementedError

Bases: *LenaException*, `NotImplementedError`

exception LenaRuntimeError

Bases: *LenaException*, `RuntimeError`

Raised when an error does not belong to other categories.

exception LenaStopFill

Bases: *LenaException*

Signal that no more fill is accepted.

Analogous to StopIteration, but control flow is reversed.

exception LenaTypeError

Bases: *LenaException*, `TypeError`

Incorrect type.

Typically used during initialization of Lena elements. Use *LenaValueError* for errors from values from the flow.

exception LenaValueError

Bases: *LenaException*, `ValueError`

Wrong value.

It is also used for values from the flow, even when they have a wrong type.

exception LenaZeroDivisionError

Bases: *LenaException*, `ZeroDivisionError`

2.3 Flow

Elements:

Cache(filename[, recompute, method, protocol])	Cache the flow passing through.
Count([name, count])	Count items that pass through.
DropContext(*args)	Sequence that transforms *(data, context)* flow so that only *data* remains in the inner sequence.
End()	Stop sequence here.
Filter(selector)	Filter values from flow.
Print([before, sep, end, transform])	Print values passing through.
Progress([name, format])	Print progress (how much data was processed and remains).
RunIf(select, *args)	Run a sequence only for selected values.

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, merge])	Group values.
`GroupPlots`(group_by[, select, transform, ...])	Deprecated since version 0.6.
`group_plots`(group)	Return data parts of the *group* and set context["group"] to their intersection.
`GroupScale`(scale_to[, allow_zero_scale, ...])	Scale a group of data.
`MapGroup`(*seq, **map_scalars)	Apply a sequence to groups.
`Selector`(selector[, raise_on_error])	A boolean function on values.
`And`(selectors[, raise_on_error])	And-test of multiple selectors.
`Or`(selectors[, raise_on_error])	Or-test of multiple selectors.
`Not`(selector[, raise_on_error])	Negate a selector.

Iterators:

`Chain`(*iterables)	Chain generators.
`CountFrom`([start, step])	Generate numbers from *start* to infinity, with *step* between values.
`ISlice`(*args, **kwargs)	Deprecated since version 0.4.
`Reverse`()	Reverse the flow (yield values from last to first).
`Slice`(*args)	Slice data flow from *start* to *stop* with *step*.

Split into bins:

Since Lena 0.5 moved to *Structures*.

2.3.1 Elements

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

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

Cache the flow passing through.

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

Example:

```
s = Source(
        ReadFiles(),
        ReadEvents(),
        MakeHistograms(),
```

(continues on next page)

(continued from previous page)

```
        Cache("histograms.pkl"),
        MakeStats(),
        Cache("stats.pkl"),
    )
```

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

Only pickleable objects can be cached (otherwise a *pickle.PickleError* will be 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. It can be given *.pkl* extension.

If *recompute* is `True`, an existing cache will always be overwritten. This option is typically used if one wants to define cache behaviour from the command line.

method can be *pickle* or *cPickle* (faster pickle). For Python 3 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.

If *recompute* was `True` during the initialization, pretend that cache does not exist (return `False`).

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 Count(*name='count'*, *count=0*)

Count items that pass through.

Example:

```
>>> flow = [0, 1, 2]
>>> c = Count("my_counter")
>>> list(c.run(iter(flow))) == [
...     0, 1, (2, {'my_counter': 3})
... ]
True
```

name is this counter's name (added to context). One can use the default name if *Count* is filled, but it is recommended to provide a meaningful name in a *Run* element.

count is the initial counter. It is added to all countings. It is set to 0 during `reset()`.

name and *count* are public attributes.

`compute``()`

Yield *(count, context)*.

context is taken from the last filled value and is updated with *{self.name: self.count}*.

`fill``(`*value*`)`

Increase *count* and set current context from *value*.

`fill_into``(`*element*, *value*`)`

Fill *element* with *value* and increase *count*.

value context is updated with *{self.name: self.count}*.

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

`reset``()`

Set *count* to zero. Clear current context.

`run``(`*flow*`)`

Yield incoming values and increase *count*.

After the flow is exhausted, update last value's context with *{self.name: self.count}*.

If the *flow* was empty, nothing is yielded (so *count* can be zero only from `compute()`).

`class` **`DropContext`**`(`**args*`)`

Sequence that transforms *(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` **`Filter`**`(`*selector*`)`

Filter values from flow.

selector is a boolean function. If it returns `True`, the value passes `Filter`. If *selector* is not callable, it is converted to a `Selector`. If the conversion could not be done, `LenaTypeError` is raised.

Note: `Filter` appeared in Lena only in version 0.4. There may be better alternatives to using this element:

- don't produce values that you will discard later. If you want to select data from a specific file, read only that file.
- use a custom class. *SelectPosition("border")* is more readable and maintainable than a `Filter` with many conditions, and it is also more *cohesive* if you group several options like "center" or "top" in a single place. If you make a selection, it can be useful to add information about that to the *context* (and `Filter` does not do that).

This doesn't mean that we recommend against this class: sometimes it can be quick and explicit, and if one's class name provides absolutely no clue what it does, a general `Filter` would be more readable.

New in version 0.4.

fill_into(*element*, *value*)

Fill *value* into an *element* if *selector(value)* is `True`.

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

run(*flow*)

Yield values from the *flow* for which the *selector* is `True`.

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).

__call__(*value*)

Print and return *value*.

class **Progress**(*name=''*, *format=''*)

Print progress (how much data was processed and remains).

name, if set, customizes the output with the collective name of values being processed (for example, "events").

format is a formatting string for the output. It will be passed keyword arguments *percent*, *index*, *total* and *name*.

Use `Progress` before a large processing. For example, if you have files with much data, put this element after generating file names, but before reading files. To print indices without reading the whole flow, use `CountFrom` and `Print`.

Progress is estimated based on the number of items processed by this element. It does not take into account the creation of final plots or the difference in the processing time for different values.

Warning: To measure progress, the whole flow is consumed.

run(*flow*)

Consume the *flow*, then yield values one by one and print progress.

class **RunIf**(*select*, **args*)

Run a sequence only for selected values.

Note: In general, different flows are transformed to common data types (like histograms). In some complicated analyses (like in `SplitIntoBins`) there can appear values of very different types, for which additional transformation must be run. Use this element in such cases.

RunIf is similar to `Filter`, but the latter can be used as a `FillInto` element inside `Split`.

RunIf with a selector *select* (let us call its opposite *not_select*) is equivalent to

```
Split(
    [
        (
            select,
            Sequence(*args)
        ),
        not_select
        # not selected values pass unchanged
    ],
    bufsize=1,
    copy_buf=False
)
```

and can be considered "syntactic sugar". Use `Split` for more flexibility.

---

select is a function that accepts a value (maybe with context) and returns a boolean. It is converted to a `Selector`. See its specifications for available options.

args are an arbitrary number of elements that will be run for selected values. They are joined into a `Sequence`.

New in version 0.4.

run(*flow*)

Run the sequence for selected values from the *flow*.

Warning: *RunIf* disrupts the flow: it feeds values to the sequence one by one, and yields the results. If the sequence depends on the complete flow (for example, yields the maximum element), this will be incorrect. The flow after *RunIf* is not disrupted.

Not selected values pass unchanged.

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 its members to the following elements, but have to transform the plots inside *GroupPlots*. We can also scale (normalize) all plots to one using *GroupScale*.

Example from a real analysis:

```
Sequence(
    # ... read data and produce histograms ...
    MakeFilename(dirname="background/{{run_number}}"),
    UpdateContext("output.plot.name", "{{variable.name}}",
                  raise_on_missing=True),
    lena.flow.GroupPlots(
        group_by="variable.coordinate",
        # Select either histograms (data) or Graphs (fit),
        # but only having "variable.coordinate" in context
        select=("variable.coordinate", [histogram, Graph]),
        # scale to data
        scale=Not("fit"),
        transform=(
            ToCSV(),
            # scaled plots will be written to separate files
            MakeFilename(
                "{{output.filename}}_scaled",
                overwrite=True,
            ),
            UpdateContext("output.plot.name", "{{variable.name}}",
                          raise_on_missing=True),
            write,
            # Several prints were used during this code creation
            # Print(transform=lambda val: val[1]["plot"]["name"]),
        ),
        # make both single and combined plots of coordinates
        yield_selected=True,
    ),
    # create file names for combined plots
    MakeFilename("combined_{{variable.coordinate}}"),
    # non-combined plots will still need file names
```

(continues on next page)

(continued from previous page)

```
    MakeFilename("{{variable.name}}"),
    lena.output.ToCSV(),
    write,
    lena.context.Context(),
    # here our jinja template renders a group as a list of items
    lena.output.RenderLaTeX(template_dir=TEMPLATE_DIR,
                            select_template=select_template),
    # we have a single template, no more groups are present
    write,
    lena.output.LaTeXToPDF(),
)
```

class GroupBy(*group_by=''*, *merge=''*)

Group values.

Data is added during `fill()`. Groups dictionary is available as `groups` attribute. `groups` is a mapping of *keys* (defined by *group_by* and *merge*) to lists of items with the same key.

group_by is a function that returns distinct hashable results for values from different groups. It can be also a dot-separated formatting string. In that case only the context part of the value is used (see `context.format_context`). *group_by* can be a tuple of strings or callables. In that case the hash value will be combined from each part of the tuple. A tuple may be used when not all parts of context can be always rendered (that would lead to an error or an empty string if they were combined into one formatting string).

Changed in version 0.6: *group_by* is no longer a function.

New in version 0.6: *merge* allows ignoring keys.

clear()

Deprecated since version 0.6: use the standard `reset()` method.

compute()

Yield values groupped by distinct keys one by one.

Each group is a tuple of filled values having the same key.

fill(*val*)

Find the corresponding group and fill it with *val*.

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

If a formatting key was not found for *val* (or if no values for a tuple *group_by* could produce keys) `LenaValueError` is raised.

reset()

Remove all groups.

update(*val*)

Deprecated since version 0.6: use the standard `fill()` method.

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

Deprecated since version 0.6: use `GroupBy`, `group_plots()` and other relevant elements.

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

Deprecated since version 0.5: use `RunIf` instead of *select*.

Plots are grouped by *group_by*, which returns different keys for different groups. It can be a function of a value or a formatting string for its context (see `GroupBy`). Example: *group_by="{{value.variable.name}}_{{variable.name}}"*.

transform is a sequence that processes individual plots before yielding. Example: `transform=(ToCSV(), write)`. *transform* is called after *scale*.

Deprecated since version 0.5: use `MapGroup` instead of *transform*.

scale 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* 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`.

yield_selected defines whether selected items should be yielded during `run()`. By default it is `False`: if we used a variable in a combined plot, we don't create a separate plot of that.

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. Otherwise, it is yielded.

After the *flow* is finished, groups are yielded. Groups are lists of items, which have same keys returned from *group_by*. Each group's context (including empty one) is inserted into a list in *context.group*. If any element's *context.output.changed* is `True`, the final *context.output.changed* is set to `True` (and to `False` otherwise). The resulting context is updated with the intersection of groups' contexts.

If *scale* 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.

group_plots(*group*)

Return data parts of the *group* and set context["group"] to their intersection.

If any of values has been changed, *context.output.changed* of the group is set to `True`.

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.

__call__(*group*)

Scale the group. See `scale_to()` for details.

If *group* is not iterable, `LenaValueError` is raised.

class **MapGroup**(**seq*, ***map_scalars*)

Apply a sequence to groups.

Arguments *seq* must form a *Sequence*.

Set a keyword argument *map_scalars* to `False` to ignore scalar values (those that are not groups). Other keyword arguments raise `LenaTypeError`.

New in version 0.5.

run(*flow*)

Map *seq* to every group from *flow*.

A value represents a group if its context has a key *group* and its data part is iterable (for example, a list of values). If length of data is different from the length of *context.group*, `LenaRuntimeError` is raised.

seq must produce an equal number of results for each item of group, or `LenaRuntimeError` is raised. These results are yielded in groups one by one.

Common changes of group context update common context (that of the value). *context.output.changed* is set appropriately.

`class` **Selector**(*selector*, *raise_on_error=True*)

A boolean function on values.

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 it is.

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

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.

raise_on_error is a boolean that sets whether in case of an exception the selector raises that exception or returns `False`. If *selector* is a container, *raise_on_error* will be used recursively during the initialization of its items.

__call__(*value*)

Check whether *value* is selected.

If an exception occurs and *raise_on_error* is `False`, the result is `False`. This could be used while testing potentially non-existing attributes or arbitrary contexts. However, this is not recommended, since it covers too many errors and some of them should be raised explicitly.

`class` **And**(*selectors*, *raise_on_error=True*)

Bases: `Selector`

And-test of multiple selectors.

selectors is a tuple of items, each of which is a `Selector` or will be converted to that.

raise_on_error has the same meaning as in `Selector`, and will be applied to each newly initialized subselector.

__call__(*val*)

Check whether *value* is selected.

If an exception occurs and *raise_on_error* is `False`, the result is `False`. This could be used while testing potentially non-existing attributes or arbitrary contexts. However, this is not recommended, since it covers too many errors and some of them should be raised explicitly.

`class` **Or**(*selectors*, *raise_on_error=True*)

Bases: `Selector`

Or-test of multiple selectors.

selectors is a list of items, each of which is a `Selector` or will be converted to that. Evaluation is short-circuit, that is if a selector was true, further ones are not applied.

raise_on_error has the same meaning as in `Selector`, and will be applied to each newly initialized subselector.

`__call__`(*val*)

Check whether *value* is selected.

If an exception occurs and *raise_on_error* is `False`, the result is `False`. This could be used while testing potentially non-existing attributes or arbitrary contexts. However, this is not recommended, since it covers too many errors and some of them should be raised explicitly.

class `Not`(*selector*, *raise_on_error=True*)

Bases: `Selector`

Negate a selector.

selector is converted to `Selector`.

raise_on_error has the same meaning as in `Selector`.

`__call__`(*value*)

Negate the result of the *selector*.

If *raise_on_error* is `False`, then this is a full negation (including the case of an error encountered in the *selector*). If *raise_on_error* is `True`, then any occurred exception will be re-raised here.

2.3.4 Iterators

Iterators allow to transform a data flow or create a new one.

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*.

`ISlice`(**args*, ***kwargs*)

Deprecated since version 0.4: use `Slice`.

class `Reverse`

Reverse the flow (yield values from last to first).

Warning: This element will consume the whole flow.

run(*flow*)

Consume the *flow* and yield values in reverse order.

class Slice(**args*)

Slice data flow from *start* to *stop* with *step*.

Initialization:

`Slice` (*stop*)

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

Similar to `itertools.islice()` or `range()`. Negative indices for *start* and *stop* are supported during `run()`.

Examples:

```
>>> Slice(1000)
```

analyse only one thousand first events (no other values from flow are generated). Use it for quick checks of data on small subsamples.

```
>>> Slice(-1)
```

yields all elements from the flow except the last one.

```
>>> Slice(1, -1)
```

yields all elements from the flow except the first and the last one.

Note that in case of negative indices it is necessary to store abs(start) or abs(stop) values in memory. For example, to discard the last 200 elements one has to a) read the whole flow, b) store 200 elements during each iteration.

It is not possible to use negative indices with `fill_into()`, because it doesn't control the flow and doesn't know when it is finished. To obtain a negative step, use a composition with `Reverse`.

fill_into(*element*, *value*)

Fill *element* with *value*.

Values are filled in the order defined by *(start, stop, step)*. *Element* must have a `fill(value)` method.

When the filling should stop, `LenaStopFill` is raised (`Split` handles this normally). Sometimes for *step* more than one `LenaStopFill` will be raised before reaching *stop* elements. Early exceptions are an optimization and don't affect the correctness of this method.

run(*flow*)

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

2.4 Input

ROOT readers:

`ReadROOTFile`([types, keys, selector])	Read ROOT files from flow.
`ReadROOTTree`([leaves, get_entries])	Read ROOT trees from flow.

2.4.1 ROOT readers

To use these classes, ROOT must be installed.

`class ReadROOTFile`(*types=None*, *keys=None*, *selector=None*)

Read ROOT files from flow.

Keyword arguments specify which objects should be read from ROOT files.

types sets the list of possible objects types.

keys specifies a list of allowed objects' names. Only simple keys are currently allowed (no regular expressions).

If both *types* and *keys* are provided, then objects that satisfy any of *types* or *keys* are read.

selector is a general function that accepts an object from a ROOT file and returns a boolean. If *selector* is given, both *types* and *keys* must be omitted, or `LenaTypeError` is raised.

`run`(*flow*)

Read ROOT files from *flow* and yield objects they contain.

For file to be read, data part of the value must be a string (file's path) and *context.input.read_root_file* must not be *False*. Other values pass unchanged. After all entries from the file are yielded, it is closed.

context.input.root_file_path is updated with the path to the ROOT file.

Warning: After a ROOT file is closed, all its contained objects are destroyed. Make all processing within one flow: don't save yielded values to a list, or save copies of them.

`class ReadROOTTree`(*leaves=None*, *get_entries=None*)

Read ROOT trees from flow.

Trees can be read in two ways.

In the first variant, *leaves* is a list of strings that enables to read the specified tree leaves. Only branches containing the leaves are read. To get a leaf from a specific branch, add it to the leaf's name with a slash, e.g. *"my_branch/my_leaf"*. Tree entries are yielded as named tuples with fields named after *leaves*.

A leaf can contain a branch name prepended

In the second variant, *get_entries* is a function that accepts a ROOT tree and yields its entries.

Exactly one of *leaves* or *get_entries* (not both) must be provided, otherwise `LenaTypeError` is raised.

Note: To collect the resulting values (not use them on the fly), make copies of them in *get_entries* (e.g. use *copy.deepcopy*). Otherwise all items will be the last value read.

`run`(*flow*)

Read ROOT trees from *flow* and yield their contents.

context.input.root_tree_name is updated with the name of the current tree.

The tree must have one and only one branch corresponding to each leaf, otherwise `LenaRuntimeError` is raised. To read leaves with the same name in several branches, specify branch names for them.

2.5 Math

Functions of multidimensional arguments:

flatten(array)	Flatten an *array* of arbitrary dimension.
mesh(ranges, nbins)	Generate equally spaced mesh of *nbins* cells in the given range.
md_map(f, *arrays)	Multidimensional map.
refine_mesh(arr, refinement)	Refine (subdivide) one-dimensional mesh *arr*.

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:

DSum([total])	Calculate an accurate floating point sum using decimals.
Mean([sum_seq, pass_on_empty])	Calculate the arithmetic mean (average) of input values.
Sum([total])	Calculate the sum of input values.
Vectorize(seq[, dim, construct])	Apply an algorithm to a vector component-wise.

3-dimensional vector:

vector3(x, y, z)	3-dimensional vector with Cartesian, spherical and cylindrical coordinates.

2.5.1 Functions of multidimensional arguments

`flatten`(*array*)

Flatten an *array* of arbitrary dimension.

array must be list or a tuple (can be nested). Depth-first flattening is used.

Return an iterator over the flattened array.

Examples:

```
>>> arr = [1, 2, 3]
>>> list(flatten(arr)) == arr
True
>>> arr = [[1, 2, 3, [4]], 5, [[6]], 7]
>>> list(flatten(arr))
[1, 2, 3, 4, 5, 6, 7]
>>> arr = [[1, 2, [3], 4], 5, [[6]], 7]
>>> list(flatten(arr))
[1, 2, 3, 4, 5, 6, 7]
```

`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*, **arrays*)

Multidimensional map.

Return function *f* mapped to contents of multidimensional *arrays*. *f* is a function of that many arguments as the number of arrays.

An item of *arrays* must be a list of (possibly nested) lists. Its contents remain unchanged. Returned array has same dimensions as those of the initial ones (they are all assumed equal). If any of *arrays* 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]]
>>> # multiple arrays
>>> md_map(lambda x, y: x+y, [0, 1], [2, 3])
[2, 4]
```

refine_mesh(*arr*, *refinement*)

Refine (subdivide) one-dimensional mesh *arr*.

refinement is the number of subdivisions. It must be not less than 1.

Note that to create a new mesh may be faster. Use this function only for convenience.

2.5.2 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*.

2.5.3 Elements

Elements for mathematical calculations.

class DSum(*total=0*)

Calculate an accurate floating point sum using decimals.

total is the initial value of the sum.

See also:

Use `Sum` for quick and precise sums of integer numbers.

compute()

Yield the calculated sum as *float*.

If the current context is not empty, yield *(sum, context)*. Otherwise yield only the *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 the sum to 0.

Context is reset to {}.

class **Mean**(*sum_seq=None*, *pass_on_empty=False*)

Calculate the arithmetic mean (average) of input values.

sum_seq is the algorithm to calculate the sum. If it is not provided, ordinary Python summation is used. Otherwise it is converted to a *FillCompute* sequence.

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

compute()

Calculate the mean and yield.

If the current context is not empty, yield *(mean, context)*. Otherwise yield only *mean*. If the *sum_seq* yields several values, they are all yielded, but only the first is divided by number of events (considered the mean value).

If no values were filled (count is zero), the 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 (considered empty if missing) is yielded in the output.

reset()

Reset sum, count and context.

Sum is reset zero (or the *reset* method of *sum_seq* is called), count to zero and context to {}.

class **Sum**(*total=0*)

Calculate the sum of input values.

total is the initial value of the sum.

See also:

Use `DSum` for exact floating summation.

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 total and context.

total is reset to 0 (not the starting number) and context to {}.

class Vectorize(*seq*, *dim=-1*, *construct=None*)

Apply an algorithm to a vector component-wise.

seq must be a *FillCompute* element or sequence.

dim is the dimension of the input data (and of the constructed structure). *seq* may also be a list of sequences, in that case *dim* may be omitted.

construct allows one to create an arbitrary object (by default the resulting values are tuples of dimension *dim*).

`compute`()

Yield results from *compute()* for each component grouped together.

If *compute* for different components yield different number of results, the longest output is yielded (the others are padded with `None`).

If the resulting value can't be converted to the type of the first value (or *construct* couldn't be used), a `tuple` is yielded.

`fill`(*val*)

Fill sequences for each component of the data vector.

2.5.4 3-dimensional vector

vector3 is a 3-dimensional vector. It supports spherical and cylindrical 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, 5, 7)
>>> v1 - v2
vector3(-3, -3, -3)
>>> 3 * v1
vector3(0, 3, 6)
>>> v1 * 3
vector3(0, 3, 6)
```

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, 4, 0)
>>> v2.r = 10
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

Vector components are floats in general. Other values can be used as well, if that makes sense for the operations used (types are not tested during the initialization). For example, vectors in the examples above have integer coordinates, which will become floats if we multiply them by floats, divide or maybe rotate. For usual vector additions or subtractions, though, their coordinates will remain integer.

class vector3(*x*, *y*, *z*)

3-dimensional vector with Cartesian, spherical and cylindrical coordinates.

Create a vector from Cartesian coordinates *x*, *y*, *z*.

Attributes

vector3 has usual vector attributes *x*, *y*, *z*, spherical coordinates *r*, *phi*, *theta* and cylindrical ones *rho* and *rho2* (*rho*^2 = *x*^2 + *y*^2).

Spherical and Cartesian coordinates are connected by this formula:

$$
\begin{aligned}
x &= r * \cos(\phi) * \sin(\theta),\\
y &= r * \sin(\phi) * \sin(\theta),\\
z &= r * \cos(\theta),
\end{aligned}
$$

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

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 and other 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)
```

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.

`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)
```

`dot`(*B*)

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

`classmethod from_spherical`(*r*, *phi*, *theta*)

Construct a new *vector3* from spherical coordinates.

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

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

Changed in version 0.6: Renamed from *fromspherical*.

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).r < 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)
>>> v2 = vector3(0, 1, 0)
>>> from math import pi
>>> vrot = v1.rotate(pi/2, v2)
>>> vrot.isclose(vector3(1, 1, -1))
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
```

2.6 Meta

Elements:

SetContext(key, value)	Set static context for this sequence.
StoreContext([name, verbose])	Store static context.
UpdateContextFromStatic()	Update runtime context with the static one.

2.6.1 Elements

class SetContext(*key*, *value*)

Set static context for this sequence.

Static context does not automatically update runtime context. Use *UpdateContextFromStatic* for that.

Static context can be used during the initialisation phase to set output directories, *Cache* names, etc. There is no way to update static context from runtime one.

key is a string representing a (possibly nested) dictionary key. *value* is its value. See *str_to_dict()* for details.

class StoreContext(*name=''*, *verbose=False*)

Store static context. Use for debugging.

name and *verbose* affect output and representation.

class UpdateContextFromStatic

Update runtime context with the static one.

Note that for runtime context later elements update previous values, but for static context it is the opposite (external and previous elements take precedence).

2.7 Output

Output:

`MakeFilename`([filename, dirname, fileext, ...])	Make file name, file extension and directory name.
`PDFToPNG`([format, overwrite, verbose, ...])	Convert PDF to image format (by default PNG).
`iterable_to_table`(iterable[, format_, ...])	Create a table from an *iterable*.
`ToCSV`([separator, header, duplicate_last_bin])	Convert data to CSV text.
`Write`(output_directory[, output_filename, ...])	Write text data to filesystem.
`Writer`(*args, **kwargs)	Deprecated since version 0.4.

LaTeX utilities:

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

2.7.1 Output

class MakeFilename(*filename=None*, *dirname=None*, *fileext=None*, *prefix=None*, *suffix=None*, *overwrite=False*)

Make file name, file extension and directory name.

filename is a string, which will be used as a file name without extension (but it can contain a relative path). The string can contain formatting arguments enclosed in double braces. These arguments will be filled from context during `__call__()`. Example:

MakeFilename("{{variable.type}}/{{variable.name}}")

dirname and *fileext* set directory name and file extension. They are treated similarly to *filename* in most aspects.

It is possible to "postpone" file name creation, but to provide a part of a future file name through *prefix* or *suffix*. They will be appended to file name during its creation. Existing file names are not affected. It is not allowed to use *prefix* or *suffix* if *filename* argument is given.

For example, if one creates logarithmic plots, but complete file names will be made later, one may use *Make-Filename(suffix="_log")*.

All these arguments must be strings, otherwise `LenaTypeError` is raised. They may all contain formatting arguments.

By default, values with *context.output* already containing *filename*, *dirname* or *fileext* are not updated (pass unaltered). This can be changed using a keyword argument *overwrite*. For more options, use `lena.context.UpdateContext`.

At least one argument must be present, or `LenaTypeError` will be raised.

__call__(*value*)

Add *output* keys to the *value*'s context.

Formatting context is retrieved from static context and from the context part of the *value*. The run-time context has higher precedence.

filename, *dirname*, *fileext*, if initialized, set respectively *context.output.{filename,dirname,fileext}* (if they didn't exist).

If this elements sets file name and if context contains *output.prefix* or *output.suffix*, they are prepended to or appended after the file name. After that they are removed from *context.output*.

If this element adds a prefix or a suffix and they exist in the context, then *prefix* is prepended before the existing prefix, and *suffix* is appended after the existing suffix, unless *overwrite* is set to `True`: in that case they are overwritten. *prefix* and *suffix* always update their existing keys in the context if they could be formatted (which is different for attributes like *filename*).

If current context can't be formatted (doesn't contain all necessary keys for the format string), a key is not updated.

class PDFToPNG(*format='png'*, *overwrite=False*, *verbose=True*, *timeoutsec=60*)

Convert PDF to image format (by default PNG).

Set output *format* (by default *png*).

If the resulting file already exists and the *pdf* is unchanged (which is checked through *context.output.changed*), conversion is not repeated. To convert all pdfs to images, set *overwrite* to `True` (by default it is `False`).

To disable printing messages during `run()`, set *verbose* to `False`.

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 element uses `pdftoppm` binary internally. `pdftoppm` can use other output formats, for example *jpeg* or *tiff*. See `pdftoppm` manual for more details.

run(*flow*)

Convert PDF files to *format*.

PDF files are recognized via *context.output.filetype*. Their paths are assumed to be the data part of the value.

Data yielded is the resulting file name. Context is updated with *output.filetype* set to *format*.

Other values are passed unchanged.

iterable_to_table(*iterable*, *format_=None*, *header=''*, *header_fields=()*, *row_start=''*, *row_end=''*, *row_separator=','*, *footer=''*)

Create a table from an *iterable*.

The resulting table is yielded line by line. If the *header* or *footer* is empty, it is not yielded.

*format_* controls the output of individual cells in a row. By default, it uses standard Python representation. For finer control, one should provide a sequence of formatting options for each column. For floating values it is recommended to output only a finite appropriate number of digits, because this would allow the output to be immutable between calls despite technical reasons. Default formatting allows an arbitrary number of columns in each cell. For tables to be well-formed, substitute missing values in the *iterable* for some placeholder like "", *None*, etc.

Each row is prepended with *row_start* and appended with *row_end*. If it consists of several columns, they are joined by *row_separator*. Separators between rows can be added while iterating the result.

This function can be used to convert structures to different formats: *csv*, *html*, *xml*, etc.

Examples:

```
>>> angles = [(3.1415*i/4, 180*i/4) for i in range(1, 5)]
>>> format_ = ("{:.2f}", "{:.0f}")
>>> header_fields = ("rad", "deg")
>>>
>>> csv_rows = iterable_to_table(
...     angles, format_=format_,
...     header="{},{}", header_fields=header_fields,
...     row_separator=",",
```

(continues on next page)

(continued from previous page)

```
... )
>>> print("\n".join(csv_rows))
rad,deg
0.79,45
1.57,90
2.36,135
3.14,180
>>>
>>> html_rows = iterable_to_table(
...     angles, format_=format_,
...     header="<table>\n" + " "*4 + "<tr><td>{}</td><td>{}</td></tr>",
...     header_fields=header_fields,
...     row_start=" "*4 + "<tr><td>", row_end="</td></tr>",
...     row_separator="</td><td>",
...     footer="</table>"
... )
>>> print("\n".join(html_rows))
<table>
    <tr><td>rad</td><td>deg</td></tr>
    <tr><td>0.79</td><td>45</td></tr>
    <tr><td>1.57</td><td>90</td></tr>
    <tr><td>2.36</td><td>135</td></tr>
    <tr><td>3.14</td><td>180</td></tr>
</table>
>>>
```

For more complex formatting use templates (see `RenderLaTeX`).

New in version 0.5.

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

Convert data to CSV text.

Can be converted:

- `histogram` (implemented only for 1- and 2-dimensional histograms).
- any iterable object (including `graph`).

separator delimits values in the output text. The result is yielded as one string starting from *header*.

If *duplicate_last_bin* is `True`, then for histograms 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"}. If a data structure has a method *_update_context(context)*, it also updates the current context during the transform. All not converted data is yielded unchanged. If *output.duplicate_last_bin* is present in context, it takes precedence over this element's value. To force the common behaviour, one can manually update context before this element.

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()`, `hist2d_to_csv()` or `iterable_to_table()`.

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 **Write**(*output_directory*, *output_filename='output'*, *verbose=True*, *existing_unchanged=False*, *overwrite=False*)

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 is useful in case of errors, when explicit file name didn't work.

verbose sets whether additional information should be printed on the screen. *verbose* set to `False` disables runtime messages.

existing_unchanged and *overwrite* are used during `run()` to change the handling of existing files. These options are mutually exclusive: their simultaneous use raises `LenaValueError`.

run(*flow*)

Only strings (and *unicode* in Python 2) and objects with a method *write* are written. Method *write* must accept a string with output file path as an argument. If *context["output"]["write"]* is set to `False`, a value will not be written. Not written 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 *fileext* are searched in *context["output"]*. If *filename* is missing, *Write*'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 usually enough to provide *fileext*.

If the resulting file exists and its content is the same as the incoming data, file is not overwritten (unless it was produced with an object's method *write*, which doesn't allow to learn whether the file has changed). If *existing_unchanged* is `True`, existing file contents are not checked (they are assumed to be not changed). If *overwrite* is `True`, file contents are not checked, and all data is assumed to be changed. If a file was written, then *output.changed* is set to `True`, otherwise, if it was not set before, it is set to `False`. If in that case *output.changed* existed, it retains its previous value.

Example: suppose you have a sequence *(Histogram, ToCSV, Write, RenderLaTeX, Write, LaTeXToPDF)*. If both histogram representation and LaTeX template exist and are unchanged, the second *Write* signals *context.output.changed=False*, and LaTeXToPDF doesn't regenerate the plot. If LaTeX template was unchanged, but the previous context from the first *Write* signals *context.output.changed=True*, then in the second *Write* template is not rewritten, but *context.output.changed* remains `True`. On the second run, even if we check file contents, the program will run faster for unchanged files even for `Write`, because read speed is typically higher than write speed.

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 data is equal to *context.output.filepath*, this means that the file was already written by another *Write*, and the value is skipped (yielded unchanged).

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

class **Writer**(**args*, ***kwargs*)

Deprecated since version 0.4: use `Write`.

2.7.2 LaTeX

class **LaTeXToPDF**(*overwrite=False*, *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.

overwrite sets whether existing unchanged pdfs shall be overwritten during `run()`.

verbose = 0 allows no output messages. 1 prints `pdflatex` command and output in case of errors. More than 1 prints all `pdflatex` output.

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”, “errorstopmode”,
“-output-directory”, output_directory, texfile_name]

run(*flow*)

Convert all incoming LaTeX files to pdf.

A *value* from *flow* corresponds to a TeX file if its *context.output.filetype* is *“tex”*. Other values pass unchanged.

If the resulting pdf file exists and *context.output.changed* is set to `False`, pdf rendering is not run. If *context.output.changed* is not set, then modification times for *.tex* and *.pdf* files are compared: if the template *.tex* is newer, it is reprocessed. Set the initialization argument *overwrite* to `True` to always recreate pdfs. All non-existent files are always created.

class **RenderLaTeX**(*select_template=''*, *template_dir='.'*, *select_data=None*, *environment=None*, *from_data=False*, *verbose=0*)

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_dir is the path to the directory with templates (used by 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. By default CSV files are selected (see `run()`).

environment allows user-defined initialisation of jinja Environment. One can use that to add custom filters, tests, global functions, etc. In that case one must set *template_dir* for that environment manually. Example user initialisation:

```python
import jinja2
from lena.output import RenderLaTeX, jinja_syntax_latex

# import user settings, filters and globals


def render_latex():
    """Construct RenderLaTeX to be used in analysis sequences."""
```

(continues on next page)

(continued from previous page)

```
    loader = jinja2.FileSystemLoader(TEMPLATE_PATH)
    environment = jinja2.Environment(
        loader=loader,
        **jinja_syntax_latex
    )
    environment.filters.update(FILTERS)
    environment.globals.update(GLOBALS)
    return RenderLaTeX(
        select_template=select_template,
        environment=environment
    )
```

Usually template context is stored in the context part of values. Sometimes, however, the data part contains the needed information (for example, during creation of tables). Set *from_data* to `True` to render the data part.

verbose controls the verbosity of output. If it is 1, selected values are printed during `run()`. If it is 2 or higher, not selected values are printed as well.

run(*flow*)

Render values from *flow* to LaTeX.

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

Rendered LaTeX text is yielded as the data part of the tuple (use `Write` to write that to the filesystem). *context.output.filetype* updates to *"tex"*.

Not selected values pass unchanged.

2.8 Structures

Histograms:

`histogram`(edges[, bins, initial_value])	A multidimensional histogram.
`Histogram`(edges[, bins, make_bins, ...])	An element to produce histograms.

`NumpyHistogram`(*args, **kwargs)	Create a histogram using a 1-dimensional *numpy.histogram*.

Graph:

`graph`(coords[, field_names, scale])	Numeric arrays of equal size.
`Graph`([points, context, scale, sort])	Deprecated since version 0.5.

`root_graph_errors`(graph[, type_code])	2-dimensional ROOT graph with errors.
`ROOTGraphErrors`()	Element to convert graphs to `root_graph_errors`.

`HistToGraph`(make_value[, get_coordinate, ...])	Transform a `histogram` to a `graph`.

Split into bins:

IterateBins([create_edges_str, select_bins])	Iterate bins of histograms.
MapBins(seq[, select_bins, get_example_bin, ...])	Transform bin content of histograms.
SplitIntoBins(seq, arg_var, edges)	Split analysis into groups defined by bins.

Histogram functions:

HistCell(edges, bin, index)	A namedtuple with fields *edges, bin, index*.
cell_to_string(cell_edges[, var_context, ...])	Transform cell edges into a string.
check_edges_increasing(edges)	Assure that multidimensional *edges* are increasing.
get_bin_edges(index, edges)	Return edges of the bin for the given *edges* of a histogram.
get_bin_on_index(index, bins)	Return bin corresponding to multidimensional *index*.
get_bin_on_value(arg, edges)	Get the bin index for *arg* in a multidimensional array *edges*.
get_bin_on_value_1d(val, arr)	Return index for value in one-dimensional array.
get_example_bin(struct)	Return bin with zero index on each axis of the histogram bins.
hist_to_graph(hist[, make_value, ...])	Convert a *histogram* to a *graph*.
init_bins(edges[, value, deepcopy])	Initialize cells of the form *edges* with the given *value*.
integral(bins, edges)	Compute integral (scale for a histogram).
iter_bins(bins)	Iterate on *bins*.
iter_bins_with_edges(bins, edges)	Generate *(bin content, bin edges)* pairs.
iter_cells(hist[, ranges, coord_ranges])	Iterate cells of a histogram *hist*, possibly in a subrange.
make_hist_context(hist, context)	Update a deep copy of *context* with the context of a *histogram hist*.
unify_1_md(bins, edges)	Unify 1- and multidimensional bins and edges.

2.8.1 Histograms

class **histogram**(*edges*, *bins=None*, *initial_value=0*)

A multidimensional histogram.

Arbitrary dimension, variable bin size and weights are supported. Lower bin edge is included, upper edge is excluded. Underflow and overflow values are skipped. Bin content can be of arbitrary type, which is defined during initialization.

Examples:

```
>>> # a 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]]
>>> # fill the histogram with values
>>> for v in values:
...     hist.fill(v)
>>> hist.bins
[[1, 1], [1, 1]]
```

edges is a sequence of one-dimensional arrays, each containing strictly increasing bin edges.

Histogram's bins by default are initialized with *initial_value*. It can be any object that supports addition with *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()`).

A histogram can be created from existing *bins* and *edges*. In this case a simple check of the shape of *bins* is done (raising `LenaValueError` if failed).

Attributes

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

`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).

`n_out_of_range` is the number of entries filled outside the range of the histogram.

`overflow` and `underflow` for a one-dimensional histogram are numbers of events above the highest (respectively, below the lowest) edges range. `n_out_of_range` is equal to the sum of `overflow` and `underflow` in that case. All these attributes are rescaled together with histogram bins during `set_nevents()` and `scale()`. For multidimensional histograms overflows and underflows are rarely used, and for efficiency reasons they are counted only for the last coordinate.

If subarrays of *edges* are not increasing or if any of them has length less than 2, `LenaValueError` is raised.

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.

`__eq__`(*other*)

Two histograms are equal, if and only if they have equal bins, edges and number of events outside of range.

If *other* is not a `histogram`, return `False`.

Note that floating numbers should be compared approximately (using `math.isclose()`).

`add`(*other*, *weight=1*)

Add a histogram *other* to this one.

For each bin, the corresponding bin of *other* is added. It can be multiplied with *weight*. For example, to subtract *other*, use *weight* -1.

Histograms must have the same edges. Note that floating numbers should be compared approximately (using `math.isclose()`).

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

Fill histogram at *coord* with the given *weight*.

Coordinates outside the histogram edges are ignored.

get_nevents(*include_out_of_range=False*)

Return number of entries in the histogram.

If the histogram was filled N times, return N. If the histogram was filled with weights w_i, return the sum of w_i. Values filled outside the histogram range are not counted unless *include_out_of_range* is `True`.

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*). Note that after changing (filling) the histogram one must explicitly recompute the scale if it was computed before.

If a float *other* is provided, rescale self to *other*.

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

set_nevents(*nevents*, *include_out_of_range=False*)

Scale histogram bins to contain *nevents*.

include_out_of_range adds `n_out_of_range` to the estimated number of entries to be rescaled. For example, suppose we know the estimated number of events for the signal and the background, and our histograms have range encompassing only a part of data. Then if we want to plot these two histograms together scaled to the real number of events, we should take into account the efficiencies of each histogram, that is set *include_out_of_range* to `True`. On the other hand, let us have two spectra in the given range and the data containing both of them. We fit the signals to the data and get their relative contributions in that region. After that we scale the histograms to those numbers of events with *include_out_of_range* set to `False` (default). In both examples `n_out_of_range` is scaled together with the histogram bins.

Rescaling a histogram with zero entries raises a `LenaValueError`.

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

An element to produce histograms.

edges, *bins* and *initial_value* have the same meaning as during creation of a `histogram`.

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

compute()

Yield histogram with context.

fill(*value*)

Fill the histogram with *value*.

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).

class NumpyHistogram(**args*, ***kwargs*)

Create a histogram using a 1-dimensional *numpy.histogram*.

The result of *compute* is a Lena `histogram`, but it is calculated using *numpy* histogram, and all its initialization arguments are passed to *numpy*.

Examples

With *NumpyHistogram()* bins are automatically derived from data.

With *NumpyHistogram(bins=list(range(0, 5)), density=True)* bins are set explicitly.

Warning: as *numpy* histogram is computed from an existing array, all values are stored in the internal data structure during *fill*, which may take much memory.

Use **args* and ***kwargs* for *numpy.histogram* initialization.

Default *bins* keyword argument is *auto*.

A keyword argument *reset* specifies the exact behaviour of *request*.

fill(*val*)

Add data to the internal storage.

request()

Compute the final histogram.

Return `histogram` with context.

If *reset* was set during the initialization, *reset* method is called.

reset()

Reset data and context.

Remove all data for this histogram and set current context to {}.

2.8.2 Graph

class graph(*coords*, *field_names=('x', 'y')*, *scale=None*)

Numeric arrays of equal size.

This structure generally corresponds to the graph of a function and represents arrays of coordinates and the function values of arbitrary dimensions.

coords is a list of one-dimensional coordinate and value sequences (usually lists). There is little to no distinction between them, and “values” can also be called “coordinates”.

field_names provide the meaning of these arrays. For example, a 3-dimensional graph could be distinguished from a 2-dimensional graph with errors by its fields (“x”, “y”, “z”) versus (“x”, “y”, “error_y”). Field names don’t affect drawing graphs: for that `Variable`-s should be used. Default field names, provided for the most used 2-dimensional graphs, are “x” and “y”.

field_names can be a string separated by whitespace and/or commas or a tuple of strings, such as (“x”, “y”). *field_names* must have as many elements as *coords* and each field name must be unique. Otherwise field names are arbitrary. Error fields must go after all other coordinates. Name of a coordinate error is “error_” appended by coordinate name. Further error details are appended after '_'. They could be arbitrary depending on the problem: "low", "high", "low_90%_cl", etc. Example: ("E", "time", "error_E_low", "error_time").

scale of the graph is a kind of its norm. It could be the integral of the function or its other property. A scale of a normalised probability density function would be one. An initialized *scale* is required if one needs to renormalise the graph in `scale()` (for example, to plot it with other graphs).

Coordinates of a function graph would usually be arrays of increasing values, which is not required here. Neither is it checked that coordinates indeed contain one-dimensional numeric values. However, non-standard graphs will likely lead to errors during plotting and will require more programmer's work and caution, so use them only if you understand what you are doing.

A graph can be iterated yielding tuples of numbers for each point.

Attributes

`coords` is a list of one-dimensional lists of coordinates.

`field_names`

`dim` is the dimension of the graph, that is of all its coordinates without errors.

In case of incorrect initialization arguments, `LenaTypeError` or `LenaValueError` is raised.

New in version 0.5.

`scale`(*other=None*)

Get or set the scale of the graph.

If *other* is `None`, return the scale of this graph.

If a numeric *other* is provided, rescale to that value. If the graph has unknown or zero scale, rescaling that will raise `LenaValueError`.

To get meaningful results, graph's fields are used. Only the last coordinate is rescaled. For example, if the graph has *x* and *y* coordinates, then *y* will be rescaled, and for a 3-dimensional graph *z* will be rescaled. All errors are rescaled together with their coordinate.

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

Deprecated since version 0.5: use `graph`. This class may be used in the future, but with a changed interface.

Function at given coordinates (arbitraty dimensions).

Graph points can be set during the initialization and during `fill()`. It can be rescaled (producing a new `Graph`). A point is a tuple of *(coordinate, value)*, where both *coordinate* and *value* can be tuples of numbers. *Coordinate* corresponds to a point in N-dimensional space, while *value* is some function's value at this point (the function can take a value in M-dimensional space). Coordinate and value dimensions must be the same for all points.

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`.

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

context is the same as the most recent context during *fill*. Use it to provide a context when initializing a `Graph` from existing points.

scale sets the scale of the graph. It is used during plotting if rescaling is needed.

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.

Note that `Graph` does not reduce data. All filled values will be stored in it. To reduce data, use histograms.

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 its precision.

property points

Get graph points (read only).

request()

Yield graph with context.

If *sort* was initialized `True`, graph points will be sorted.

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*)

Deprecated since version 0.5: in Lena 0.5 to_csv is not used. Iterables are converted to tables.

Convert graph's points to CSV.

separator delimits values, the default is 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*), then each part of its value.

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

class root_graph_errors(*graph*, *type_code='d'*)

2-dimensional ROOT graph with errors.

This is an adapter for TGraphErrors and contains that graph as a field *root_graph*.

graph is a Lena `graph`.

type_code is the basic numeric type of array values (by default double). 'f' means floating values. See Python module array for more options.

New in version 0.5.

class ROOTGraphErrors

Element to convert graphs to `root_graph_errors`.

__call__(*value*)

Convert data part of the value (which must be a `graph`) to `root_graph_errors`.

New in version 0.5.

class **HistToGraph**(*make_value*, *get_coordinate='left'*, *field_names=('x', 'y')*, *scale=None*)

Transform a `histogram` to a `graph`.

make_value is a `Variable` that creates graph value from the bin value.

get_coordinate defines the coordinate of the graph point. By default, it is the left bin edge. Other allowed values are "right" and "middle".

field_names set field names of resulting graphs.

scale sets scales of resulting graphs. If it is `True`, the scale is computed from the histogram.

See `hist_to_graph()` for details and examples.

Incorrect values for *make_value* or *get_coordinate* raise, respectively, `LenaTypeError` or `LenaValueError`.

run(*flow*)

Iterate the *flow* and transform histograms to graphs.

context.value is updated with *make_value* context. If histogram bins contained context (which is assumed to be the same for all bins), *make_value* context is composed with that.

Not histograms or histograms with *context.histogram.to_graph* set to `False` pass unchanged.

2.8.3 Split into bins

Split analysis into groups defined by bins.

class **IterateBins**(*create_edges_str=None*, *select_bins=None*)

Iterate bins of histograms.

create_edges_str is a callable that creates a string from bin's edges and coordinate names and adds that to the 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()`.

select_bins is a callable used to test bin contents. By default, only those histograms are iterated where bins contain histograms. Use *select_bins* to choose other classes. See `Selector` for examples.

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

run(*flow*)

Yield histogram bins one by one.

For each `histogram` from the *flow*, if its bins pass *select_bins*, they are iterated.

The resulting context is taken from bin's context. Histogram's context is preserved in *context.bins*. *context.bin* is updated with "edges" (with bin edges) and "edges_str" (their representation). If histogram's context contains *variable*, that is used for edges' representation.

Not histograms pass unchanged.

class **MapBins**(*seq*, *select_bins=<function MapBins.<lambda>>*, *get_example_bin=<function get_example_bin>*, *drop_bins_context=True*)

Transform bin content of histograms.

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

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

If *select_bins* applied to histogram bins is `True` (tested on an arbitrary bin), the histogram is transformed. Bin types can be given in a `list` or as a general `Selector`. For example, `select_bins=[lena.math.vector3, list]` selects histograms where bins are vectors or lists. By default all histograms are accepted.

The "arbitrary bin" is returned by a callable *get_example_bin* (by default `get_example_bin()`).

`MapBins` creates histograms that may be plotted, because their bins contain only data without context. If *drop_bins_context* is `False`, context remains in bins. By default, context of all histogram bins is discarded. This discourages compositions of `MapBins`: make compositions of their internal sequences instead.

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

run(*flow*)

Transform histograms from *flow*.

context.value is updated with bin context (if that exists). It is assumed that all bins have the same context (because they were produced by the same sequence), therefore an arbitrary bin is taken and contexts of all other bins are ignored.

Not selected values pass unchanged.

class **SplitIntoBins**(*seq*, *arg_var*, *edges*)

Split analysis into groups defined by bins.

seq is a `FillComputeSeq` sequence (or will be converted to that) that corresponds to the analysis being performed for different bins. Deep copy of *seq* is done for each bin.

arg_var is a `Variable` that takes data and returns value used to compute the bin index. Example of a two-dimensional function: `arg_var = 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)`.

Note: The final histogram may contain vectors, histograms and any other data the analysis produced. To plot them, one can extract vector components with e.g. `MapBins`. If bin contents are histograms, they can be yielded one by one with `IterateBins`.

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)* pair for each *compute()* for all bins.

The `histogram` is created from `edges` with bin contents taken from *compute()* for `bins`. Computational context is preserved in histogram's bins.

`SplitIntoBins` adds context as a subcontext *variable* (corresponding to *arg_var*). This allows unification of `SplitIntoBins` with common analysis using variables (useful when creating plots from one template). Existing context values are preserved.

Note: 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_var(val)* with *val*.

Values outside the `edges` are ignored.

2.8.4 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.

`class HistCell`(*edges*, *bin*, *index*)

A namedtuple with fields *edges, bin, index*.

Create new instance of HistCell(edges, bin, index)

`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.

`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_edges`(*index*, *edges*)

Return edges of the bin for the given *edges* of a histogram.

In one-dimensional case *index* must be an integer and a tuple of *(x_low_edge, x_high_edge)* for that bin is returned.

In a multidimensional case *index* is a container of numeric indices in each dimension. A list of bin edges in each dimension is returned.

`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)
```

(continues on next page)

(continued from previous page)

```
2
>>> # upper range is excluded
>>> get_bin_on_value_1d(10, arr)
5
>>> # underflow
>>> get_bin_on_value_1d(-10, arr)
-1
```

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.

hist_to_graph(*hist*, *make_value=None*, *get_coordinate='left'*, *field_names=('x', 'y')*, *scale=None*)

Convert a `histogram` to a `graph`.

make_value is a function to set the value of a graph's point. By default it is bin content. *make_value* accepts a single value (bin content) without context.

This option could be used to create graph's error bars. For example, to create a graph with errors from a histogram where bins contain a named tuple with fields *mean*, *mean_error* and a context one could use

```
>>> make_value = lambda bin_: (bin_.mean, bin_.mean_error)
```

get_coordinate defines what the coordinate of a graph point created from a histogram bin will be. It can be "left" (default), "right" and "middle".

field_names set field names of the graph. Their number must be the same as the dimension of the result. For a *make_value* above they would be *("x", "y_mean", "y_mean_error")*.

scale becomes the graph's scale (unknown by default). If it is `True`, it uses the histogram scale.

hist must contain only numeric bins (without context) or *make_value* must remove context when creating a numeric graph.

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]]
```

(continues on next page)

(continued from previous page)

```
>>> 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).

`iter_bins_with_edges`(*bins*, *edges*)

Generate *(bin content, bin edges)* pairs.

Bin edges is a tuple, such that its item at index i is *(lower bound, upper bound)* of the bin at i-th coordinate.

Examples:

```
>>> from lena.math import mesh
>>> list(iter_bins_with_edges([0, 1, 2], edges=mesh((0, 3), 3)))
[(0, ((0, 1.0),)), (1, ((1.0, 2.0),)), (2, ((2.0, 3),))]
>>>
>>> # 2-dimensional histogram
>>> list(iter_bins_with_edges(
...     bins=[[2]], edges=mesh(((0, 1), (0, 1)), (1, 1))
... ))
[(2, ((0, 1), (0, 1)))]
```

New in version 0.5: made public.

`iter_cells`(*hist*, *ranges=None*, *coord_ranges=None*)

Iterate cells of a histogram *hist*, possibly in a subrange.

For each bin, yield a *`HistCell`* containing *bin edges, bin content* and *bin index*. The order of iteration is the same as for *`iter_bins()`*.

ranges are the ranges of bin indices to be used for each coordinate (the lower value is included, the upper value is excluded).

coord_ranges set real coordinate ranges based on histogram edges. Obviously, they can be not exactly bin edges. If one of the ranges for the given coordinate is outside the histogram edges, then only existing histogram edges within the range are selected. If the coordinate range is completely outside histogram edges, nothing is yielded. If a lower or upper *coord_range* falls within a bin, this bin is yielded. Note that if a coordinate range falls on a bin edge, the number of generated bins can be unstable because of limited float precision.

ranges and *coord_ranges* are tuples of tuples of limits in corresponding dimensions. For one-dimensional histogram it must be a tuple containing a tuple, for example *((None, None),)*.

`None` as an upper or lower *range* means no limit (*((None, None),)* is equivalent to *((0, len(bins)),)* for a 1-dimensional histogram).

If a *range* index is lower than 0 or higher than possible index, *`LenaValueError`* is raised. If both *coord_ranges* and *ranges* are provided, *`LenaTypeError`* is raised.

make_hist_context(*hist*, *context*)

Update a deep copy of *context* with the context of a *histogram* *hist*.

Deprecated since version 0.5: histogram context is updated automatically during conversion in *ToCSV*. Use histogram._update_context explicitly if needed.

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.

2.9 Variables

Variables:

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

2.9.1 Variables

Variables are functions to transform data adding 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"
... )
>>> positron = Variable(
...     "positron", latex_name="e^+",
...     getter=lambda double_ev: double_ev[0], type="particle"
... )
>>> x_e = Compose(positron, x)
>>> x_e(data[0])[0]
1.05
>>> x_e(data[0])[1] == {
...     'variable': {
...         'name': 'x',
...         'coordinate': {'name': 'x'},
...         'type': 'coordinate',
...         'compose': ['particle', 'coordinate'],
...         'particle': {'name': 'positron', 'latex_name': 'e^+'}
```

(continues on next page)

(continued from previous page)

```
...     }
... }
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 containing 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.

A keyword argument *name* can set the name of the composed variable. If that is missing, it the name of the last variable is used.

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

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

class Variable(*name*, *getter*, *type=''*, ***kwargs*)

Function of data with context.

name is variable's name.

getter is a 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 could be "coordinate" or "particle". It has a special meaning: if present, its value is added to variable's context as a key with the context of this variable (see the example for this module). It is recommended to set the type, otherwise variable's data will be lost after composition of variables.

Attributes

getter

var_context is the dictionary of attributes of the variable. It 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 written as *var.latex_name*). `AttributeError` is raised if an 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 *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.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER FOUR

INSTALLATION

4.1 Minimal

Install the latest official version from PyPI:

```
pip install lena
```

Lena core modules have no dependencies except Python standard libraries.

4.2 Recommended

```
pip install lena jinja2
```

jinja2 is used to create templates for plots. Also install the following programs:

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

These programs can be found in your OS packages. For example, in Fedora Core 29 install them with

```
dnf install texlive-latex texlive-pgfplots poppler-utils
```

pdflatex and *pgfplots* are contained in the standard TeX Live distribution.

4.3 Full

This installation is needed only if you want to extend and develop Lena. Download the full repository (with history) from GitHub and install all development dependencies:

```
git clone https://github.com/ynikitenko/lena
pip install -r lena/requirements.txt
```

Install command line programs from the previous subsection and adjust PYTHONPATH as shown in the next subsection.

4.4 GitHub or PyPI

PyPI contains the last official release, which was tested for more Python versions. GitHub contains the most recent development code for experienced users. Usually it is well tested too, but there is a chance that a newly introduced interface will be changed.

For most users *pip* install should be easier. If for some reasons you can't do that, you can get an archive of an official release from GitHub releases.

pip installs the framework into a system directory, while to install with *git* you need to adjust the PYTHONPATH. Add to your profile (e.g. `.profile` or `.bashrc` on Linux)

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

and replace <*path-to-lena*> with the actual path to the cloned repository.

CHAPTER FIVE

DOCUMENTATION

To get started, read the *Tutorial*.

Complete documentation for Lena modules can be found in the *Reference*.

See Release Notes for changes.

CHAPTER

SIX

LICENSE

Lena is free software released 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 data analysis, but we don't take responsibility if something goes wrong.

CHAPTER SEVEN

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

INDEX