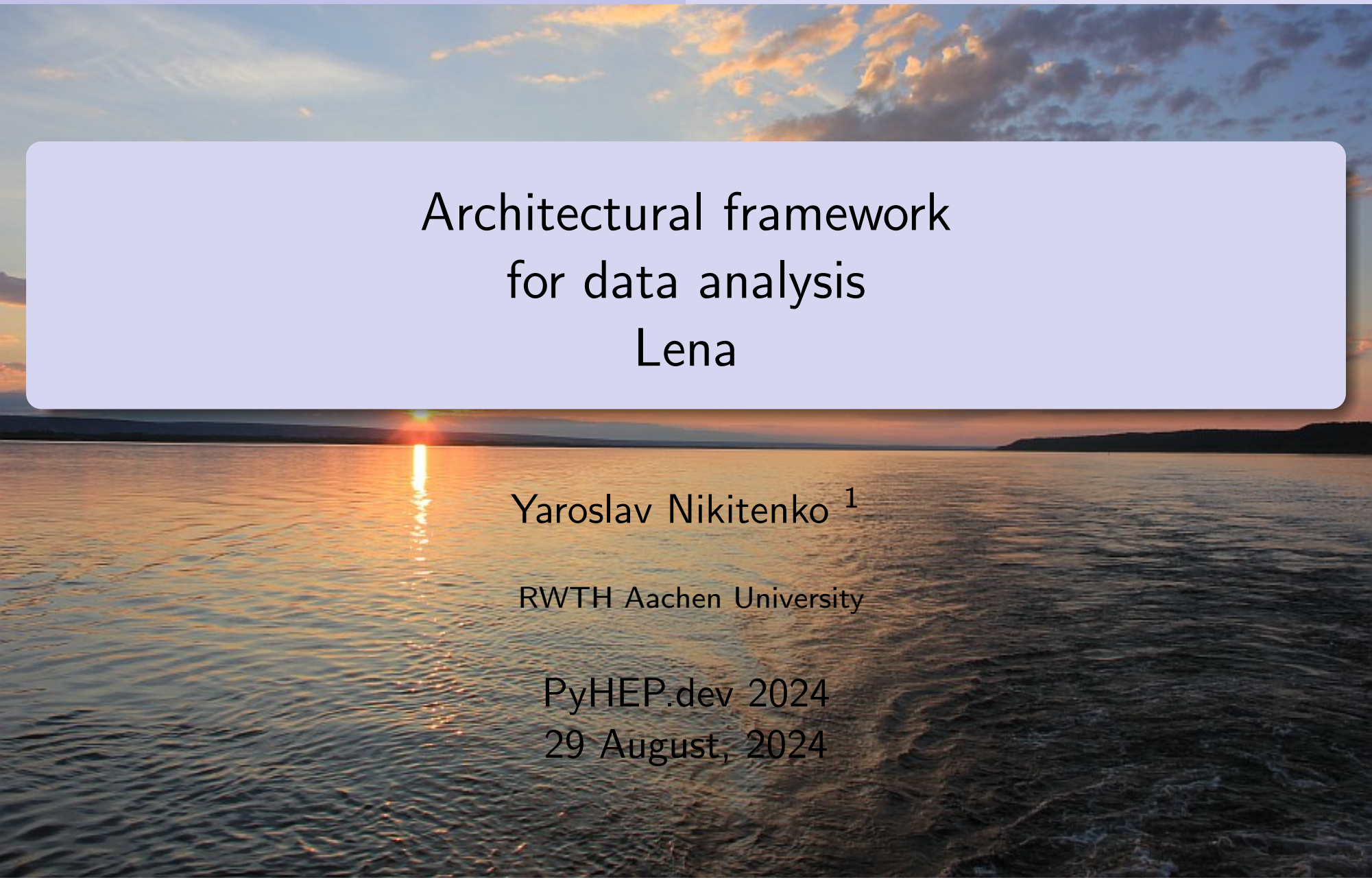


Architectural framework for data analysis Lena

Yaroslav Nikitenko [1]

RWTH Aachen University

PyHEP.dev 2024
29 August, 2024

[1] metst13 (at) gmail.com

Architecture

Architecture defines the most important decisions in the program.
Affects its maintainability and extensibility.

Architectural framework

An *architectural framework* dictates the user how to write their program. Cf. a *library*, which provides you functions that you can use.

Architectural framework

An *architectural framework* dictates the user how to write their program.
Cf. a *library*, which provides you functions that you can use.

Examples of architectural frameworks in Web programming:
Django, Flask,...
High quality. Help many users.

Can we have architectural frameworks in data analysis?

Plan

1. 3 ideas behind Lena
    1. Sequences and elements
    2. Lazy evaluation
    3. Context and template rendering
2. Applying several algorithms in one pass
3. Reusing sequences
4. Meta programming

Sequences and elements

```
>>> from lena.core import Sequence
>>>
>>> s = Sequence(
...     lambda i: pow(-1, i) * (2 * i + 1),
>>> )
>>>
>>> data = [0, 1, 2, 3]
>>>
>>> results = s.run(data)
>>>
>>> list(results)
[1, -3, 5, -7]
```

Sequences and elements

```
import lena.flow
from lena.core import Sequence, Source
s = Sequence(
    lambda i: pow(-1, i) * (2 * i + 1),
)
spi = Source(
    lena.flow.CountFrom(0),  # 0 1 2 3 ...
    s,
    lena.flow.Slice(10**6),  # take the first million
    lambda x: 4./x,
    Sum(),
)
# materialize the results
list(spi())
# [3.14159165359]
```

Lazy evaluation with generators

```
class Sum():

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

Lazy evaluation with generators

```
class Sum():

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

Easily created in Python with *yield*,

Speed: calculations can be finished almost immediately after reading the input data,

Scalability: data size can be larger than memory.

jinja2 template of a PGF/TikZ plot

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

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

PGF/TikZ plots

Structured and *declarative* definition.

```
# compare to an example from Matplotlib User's guide
import matplotlib.pyplot as plt

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```

Imperative definition. Not structured.

PGF/TikZ plots

Structured and *declarative* definition.

Best "publication quality". Native to LaTeX.

Best "presentation quality". Can scale in beamer maintaining font sizes.

A more detailed Lena user's template

```
\BLOCK{ block header }
\documentclass{standalone}
\BLOCK{ endblock header }
\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 }
    },
    % or format xlabel via a filter (function)
    \BLOCK{ endif }
]
```

Template rendering

jinja2 templates support[2]: variables, cycles, branches, template inheritance, ...

Presentation is separated from structure (drawing logic is in the template, analysis logic in the code),

Can adjust one template for many plots,

Template reuse improves quality of many plots,

For that, data is supplemented with *context*/metadata.

[2] Lena templates slightly modify jinja2 templates for LaTeX (instead of HTML).

Data analysis. User class

```
class ReadData():

  def run(self, flow):
      for filename in flow:
          with open(filename, "r") as fil:
              for line in fil:
                  vec = [float(coord)
                         for coord in line.split(',')]
                  yield vec
                  # or a (data, context) pair
                  # yield (vec, {"data": {"data_type": "MC"}}
```

Data analysis

```
from lena.output import HistToCSV, MakeFilename, Write, RenderLaTeX ...
from lena.structures import Histogram

s = Sequence(
    ReadData(),
    lambda vec: vec[0],  # take x component
    Histogram(lena.math.mesh((-10, 10), 10)),
    MakeFilename("x"),  # add {"output": {"filename": "x"}} to
    HistToCSV(),
    Write("output"),  # write text to output/
    RenderLaTeX("histogram_1d.tex"),  # name of jinja templat
    Write("output"),  # write final TeX file
    LaTeXToPDF(),  # run pdflatex
    PDFToPNG(),
)
print(list(s.run(["data.csv"])))
```

Data analysis. Development

```
from lena.flow import Cache, End, Print
s = Sequence(
    Print(),
    ReadData(),
    Print(),
    Slice(1000),  # take first 1000 values
    lambda vec: vec[0], # x
    Histogram(lena.math.mesh((-10, 10), 10)),
    # the histogram will be filled 1 time and cached
    Cache("x_hist.pkl"),
    # End(),
    HistToCSV(),
    MakeFilename("x"),
    Write("output"),
    # ...
)
```

Data analysis. Performance improvement

```
...
HistToCSV(),
Write("output"),
RenderLaTeX("histogram_1d.tex"),  # name of jinja templat
Write("output"),  # write final TeX file
LaTeXToPDF(),  # run pdflatex
PDFToPNG(),
```

If the written file was unchanged, Write adds "output.changed" = False, and the plot is not reprocessed by LaTeXToPDF.

Lena sequences and elements

Elements

Element	method	cardinality
Call	__call__(value)	$1 \to 1$
Run	run(flow)	many $\to$ many
SourceEl	__call__()	$0 \to$ many

Sequences

Lena sequence	Elements	method call
Sequence	(Call/Run, ...)	seq.run(flow)
Source	(SourceEl, Call/Run, ...)	src()

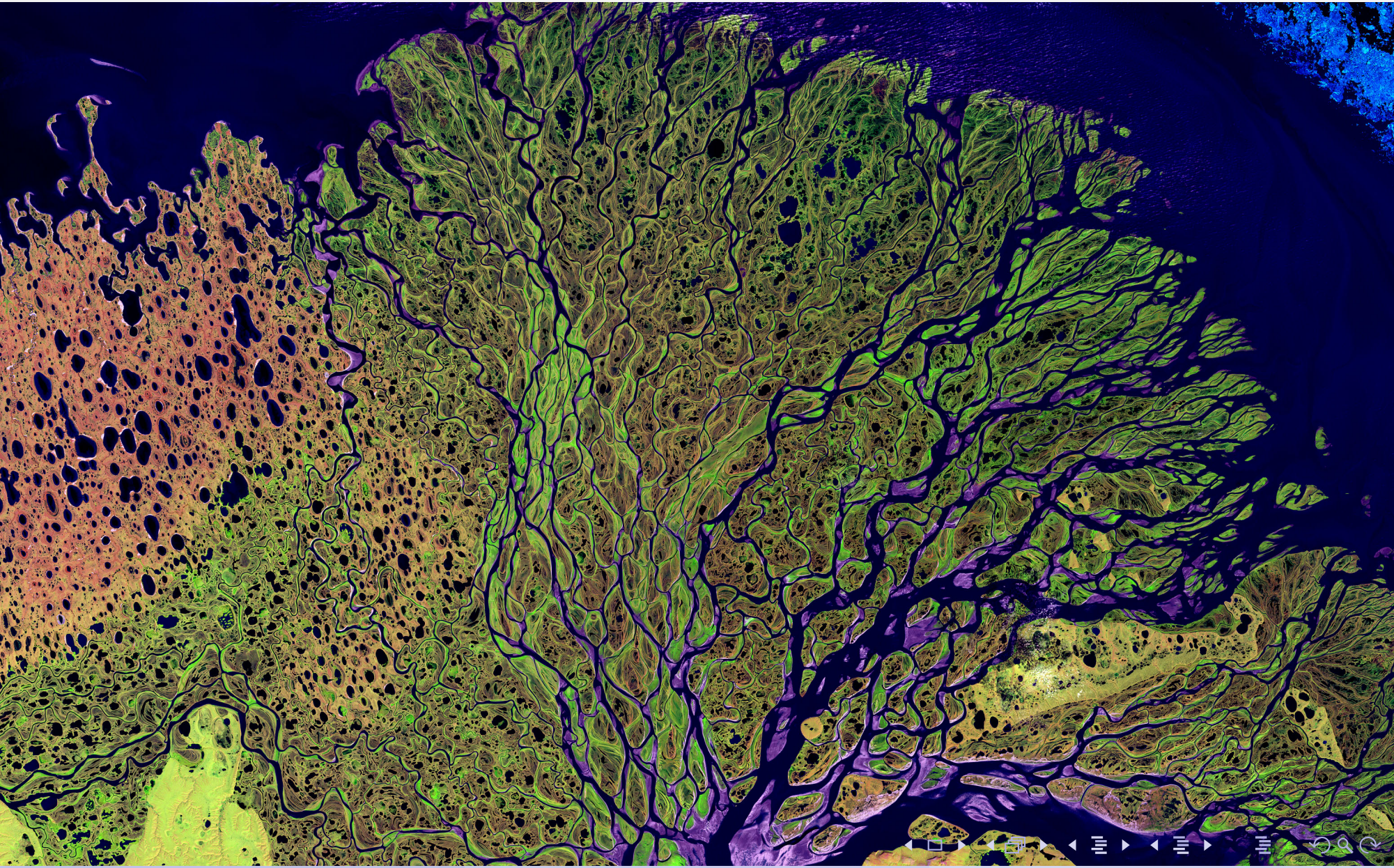
Lena architecture

Sequences provide *loose coupling* of analysis elements.

Data flow is explicit, any place in the sequence can be changed (one doesn't need to edit a separate file of each function), every framework element can be changed,

All control is from Python (with its infrastructure).

Applying several algorithms in one pass



Split

```
from lena.core import Split
s = Source(
    ReadEvents(),
    # yields Event(x, y, z, E,...)
    Split([
        (  # Sequence 1
            lambda ev: ev.x,
            Histogram(mesh((-10, 10), 10)),
            MakeFilename("x"),
        ),
        (  # Sequence 2
            lambda ev: ev.y,
            Histogram(mesh((-10, 10), 10)),
            MakeFilename("y"),
        ),
    ]),
```

Limitations of the Run element

```
class Sum():

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

Limitations of the Run element

```
class Sum():

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

flow will be exhausted before the control will be released from *run*.

FillCompute element

```
class Sum():

    def __init__(self):
        self._sum = 0

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

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

Variable

```
from lena.variables import Variable
s = Sequence(
    ReadEvents(),
    Split([
        (
            lambda ev: ev.x,
            Sum(),
        ),
        (
            Variable("x", lambda ev: ev.x),
            Histogram(mesh((-10, 10), 10)),
        ),
    ]),
    MakeFilename("{variable.name}"),
    HistToCSV(),
    # ...
```

More plots

```
def coordinates():
    return [
        (
            Variable("x", lambda ev: ev.x,
                     unit="m", type="coord"),
            Histogram(mesh((-3, 3), 10)),
        ),
        # y, z
    ]

positron = Variable("e", lambda tup: tup[0],
                    latex_name="$e^+$", type="particle")
neutron  = Variable("n", lambda tup: tup[1],
                    latex_name="n", type="particle")
```

More histograms

```
s = Sequence(
    ReadDoubleEvents(),
    # (PromptEvent, DelayedEvent) pairs
    Split([
        (
            particle,
            Split(coordinates()),
        ),
        for particle in (positron, neutron)
        # use Python constructs in sequences
    ]),
    MakeFilename("{variable.particle}_{variable.coord}"),
    HistToCSV(),
    # ...
)
```

6 histograms

More plots

```
from lena.output import MakeFilename

def add_logy():
    return Split([
        (),
        (
            UpdateContext("output.plot.logy", True),
            lena.output.MakeFilename(suffix="_log"),
        )
    ])

def coordinates():
    return [
        (
            Variable("x", lambda ev: ev.x,
                     unit="m", type="coord"),
            Histogram(mesh((-3, 3), 10)),
            add_logy(),
            add_ymin_free(yield_orig=True),  # similar
            # add_fit, ...
        ),
        # y, z
    ]
```

24 histograms

Transforming sequences

```
# Split data into histogram bins and
# use a Sequence for each bin

split_dz_vs_z = SplitIntoBins(
    # dz analysis
    (
        Variable(
            "dz",
            lambda double_ev: double_ev.displacement.z/1000,
            latex_name=r"{\rm d}z",
            unit="m"
        ),
        ROOTHistogram(ROOT.TH1D("dz_vs_z", "dz vs z",
                                nbins, -0.500, 0.500)),
    ),
    # vs z variable
    arg_var=Variable("z",
                     lambda double_ev: double_ev.first.z/1000.,
                     latex_name="z",
                     unit="m"),
    edges=lena.math.mesh((-z_max, z_max), nbins),
)
```

Transforming sequences

```
# Split data into histogram bins and
# use a Sequence for each bin

split_dz_vs_z = SplitIntoBins(
    # dz analysis
    (
        Variable(
            "dz",
            lambda double_ev: double_ev.displacement.z/1000,
            latex_name=r"{\rm d}z",
            unit="m"
        ),
        ROOTHistogram(ROOT.TH1D("dz_vs_z", "dz vs z",
                                nbins, -0.500, 0.500)),
    ),
    # vs z variable
    arg_var=Variable("z",
                     lambda double_ev: double_ev.first.z/1000.,
                     latex_name="z",
                     unit="m"),
    edges=lena.math.mesh((-z_max, z_max), nbins),
)
```

A FillCompute Sequence must support deep copy.

Transforming sequences

```
def _split_into_zones(head, mid, tail):
    # filter the inner part of the detector
    new_mid = FillComputeSeq(
        # add a new filter to the sequence before mid
        filter_inner,
        deepcopy(mid),
        SetContext("data.full_name",
                   "{{data.full_name}}_" + "inner"),
        *deepcopy(tail)
    )
    return (head, new_mid, [])
```

Use this function to transform the actual analysis sequence,

Transforming sequences

```
seq = double_events_main(settings)
# transform: (head, mid, tail) -> (head, new_mid, new_tail)
new_seq = transform(seq, Split,
                    transform_seq=_split_into_zones)
```

Transforming sequences

```
seq = double_events_main(settings)
# transform: (head, mid, tail) -> (head, new_mid, new_tail)
new_seq = transform(seq, Split,
                    transform_seq=_split_into_zones)
```

Elements and parts of sequences can be substituted and transformed without having to edit the original code.

Merging sequences

Analysis chains with common heads can be merged:
S1 = (e11, e12, ... e1N1)
S2 = (e21, e22, ... e2N1)
if first k elements of S1 and S2 are equal, then
S_merge = (e11, ... e1k, Split([(e1k+1,... e1N1), (e2k+1, ... e2N2)])).
Same for N sequences.

A real example:

```
s1 = make(double_events_main(settings))
s2 = make(reco_bias_zones_main(settings))
sana = merge_heads(s1, s2)
```

allows us to run two different analyses using the same data in one go.

Merging sequences

Analysis chains with common heads can be merged:
S1 = (e11, e12, ... e1N1)
S2 = (e21, e22, ... e2N1)
if first k elements of S1 and S2 are equal, then
S_merge = (e11, ... e1k, Split([(e1k+1,... e1N1), (e2k+1, ... e2N2)])).
Same for N sequences.

A real example:

```
s1 = make(double_events_main(settings))
s2 = make(reco_bias_zones_main(settings))
sana = merge_heads(s1, s2)
```

allows us to run two different analyses using the same data in one go.

Elements must have equality comparisons (*lambdas* won't work).

Conclusions

In order to be *reusable* and *composable*, analysis elements should be *loosely coupled* and have a *common interface*; to facilitate *metaprogramming* one needs *introspectable* code.
Lena assists a user to:

conduct an analysis **from data to final plots/tables**,

decouple numerical analysis from plotting,

use one template for **many plots**,

write **reusable** and **composable** software allowing **introspection**,

make your programs more **structured** and **beautiful**.

Conclusions

Status

Published on `https://github.com/ynikitenko/lena/` and PyPI.

90% test coverage. Release 0.5. The structure stays the same, some elements change.

User elements almost don't change.

meta in development.