The NEXT experiment analysis framework

J. Generowicz, J.J. Gómez Cadenas, **G. Martínez Lema**

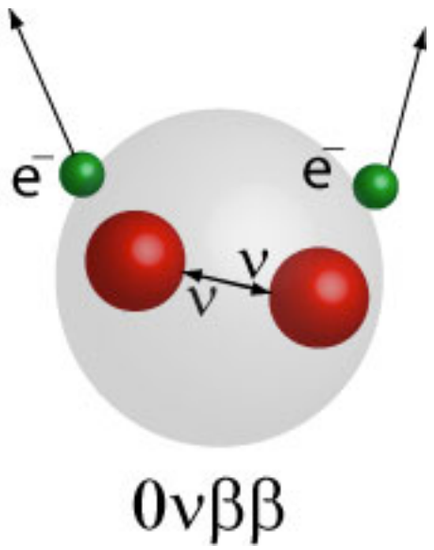
Outline

- Overview of the NEXT experiment
- Software philosophy
- The IC framework
  - Features
  - Structure
  - Dataflow

What is NEXT?

The NEXT experiment

- NEXT stands for **N**eutrino **E**xperiment with a **X**enon **T**PC
- We aim to detect the neutrinoless double beta decay ($\beta\beta^{0\nu}$) in ^{136}Xe

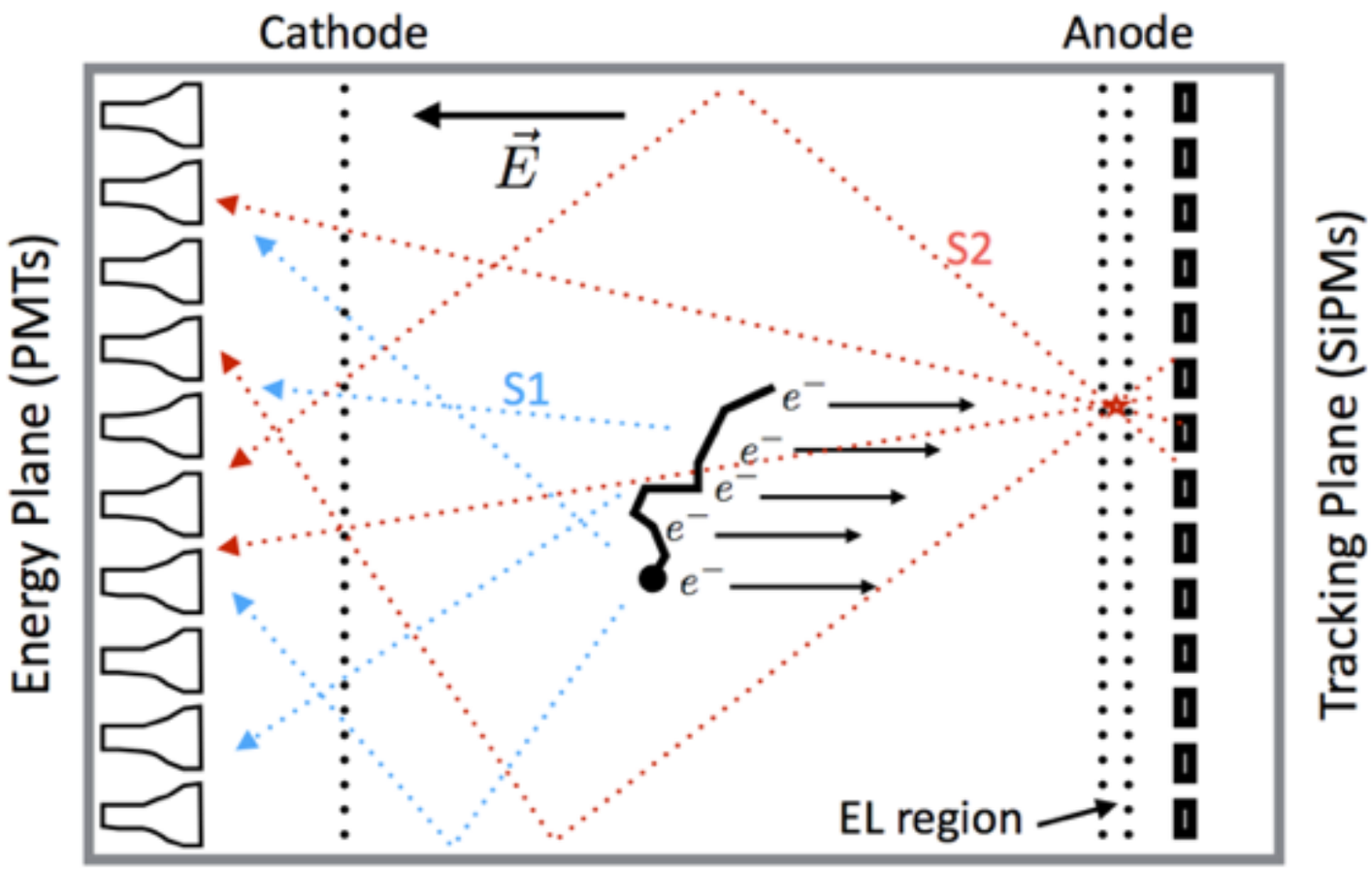


$0\nu\beta\beta$

The NEXT-White detector, first stage of the experiment

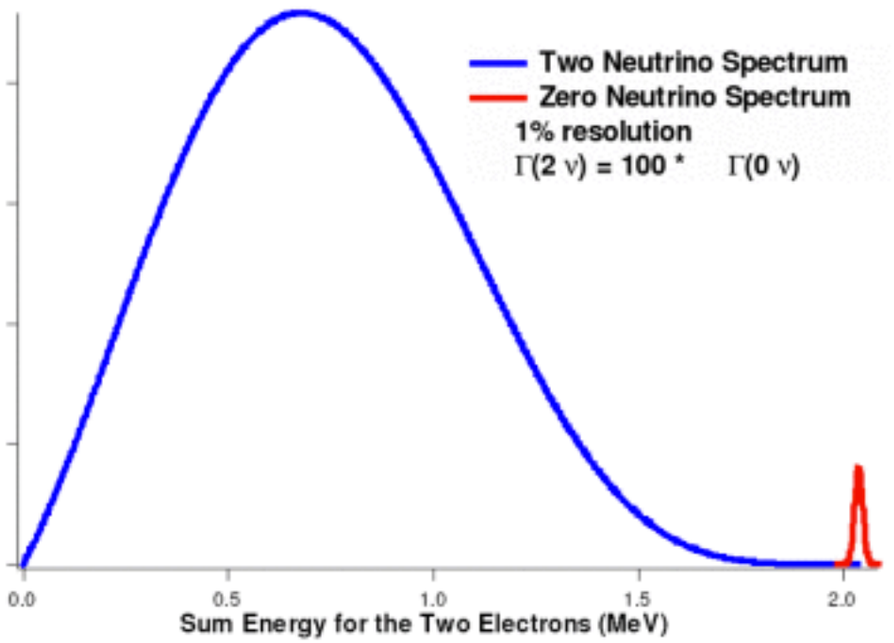
The NEXT experiment

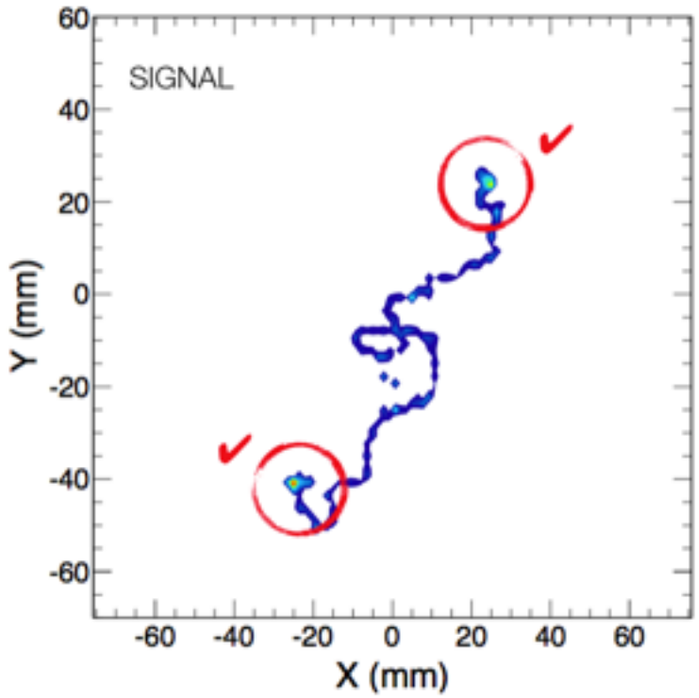
- The detector is a Time Projection Chamber with two sensor planes:
    - PMTs for calorimetry
    - SiPMs for tracking

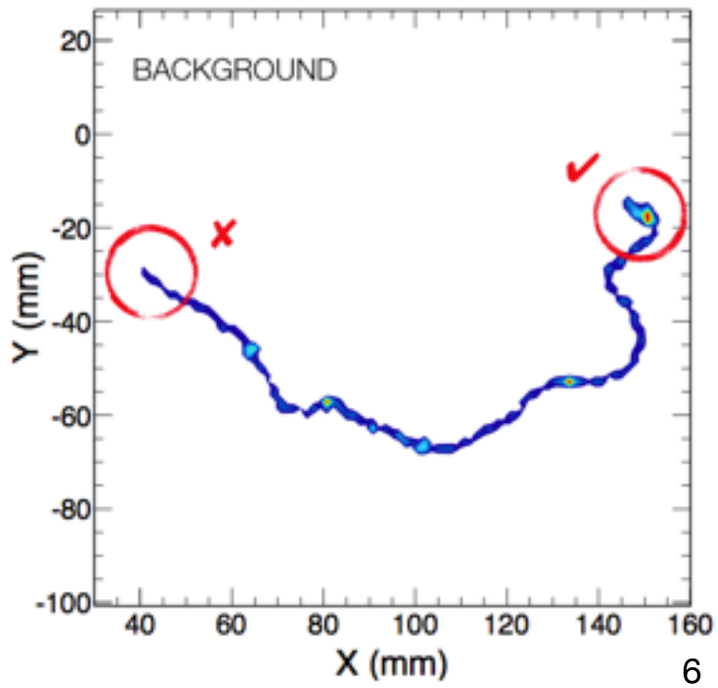


The NEXT experiment

- To detect the ßß$^{0\nu}$, we measure the energy spectrum of the two electrons emitted in the decay
    - The signal (ßß$^{0\nu}$) spectrum is a narrow peak around the decay energy
- To reduce background contamination we perform topological analyses:
    - Signal events have large energy depositions (blobs) at both ends of a thin track
    - Background events are identical, but they contain just one blob







Software philosophy

Our previous experience

- C++-based code
    - Write a lot to do very little: very low productivity
    - Lack of expertise: bad quality code
    - ROOT-oriented
    - No automated test suite: continuous bug fixing
    - Bus number* <= 1
    - Use of personal implementations: reinvention of the wheel

*Bus number: the smallest number of members of your team that would have to be run over by a bus, in order to stall your project

Software philosophy

- Python first
    - High-productivity language
    - The user can easily become a developer
    - CPU-intensive tasks can be handled by the vast range of standard libraries with minimal speed penalty
        - We have found that cython and numba are almost expendable, with few exceptions
- Automated test suite
    - Makes the code more robust
    - Protects future modifications against bugs
    - Useful as documentation
    - Increases the comprehension of the code

Software philosophy

- Do not reinvent the wheel
    - Most of what we need has already been written (better and more efficiently) elsewhere
    - We only need to make it fit our use case
    - Our code relies on standard libraries such as numpy, scipy, pytables, pandas and matplotlib
- Code review by a second developer
    - Bad quality code can waste a lot of human resources
    - Increases the knowledge and understanding of the code among the collaborators: increase of the bus number
    - Untested code cannot make its way to the official software

The IC framework

The IC framework

- IC stands for Invisible Cities and for Italo Calvino, author of this book
- Publicly available repository at https://github.com/nextic/IC

IC core team:

- Jacek Generowicz
- Gonzalo Martínez Lema
- Juan José Gómez Cadenas
- Alejandro Botas
- Paola Ferrario
- Jose María Benlloch
- Ander Simón
- Andrew Laing
- Brais Palmeiro
- Josh Renner
- Jose Angel Hernando Morata

Features

- It is a pure-python framework
- Written in python 3 (https://pythonclock.org/)
- Built around the anaconda ecosystem with few exceptions
- No ROOT
- No C++
- Version control via github (+ magit)
- Largely tested with pytest and hypothesis
- Continuous integration with Travis
- Continuous release

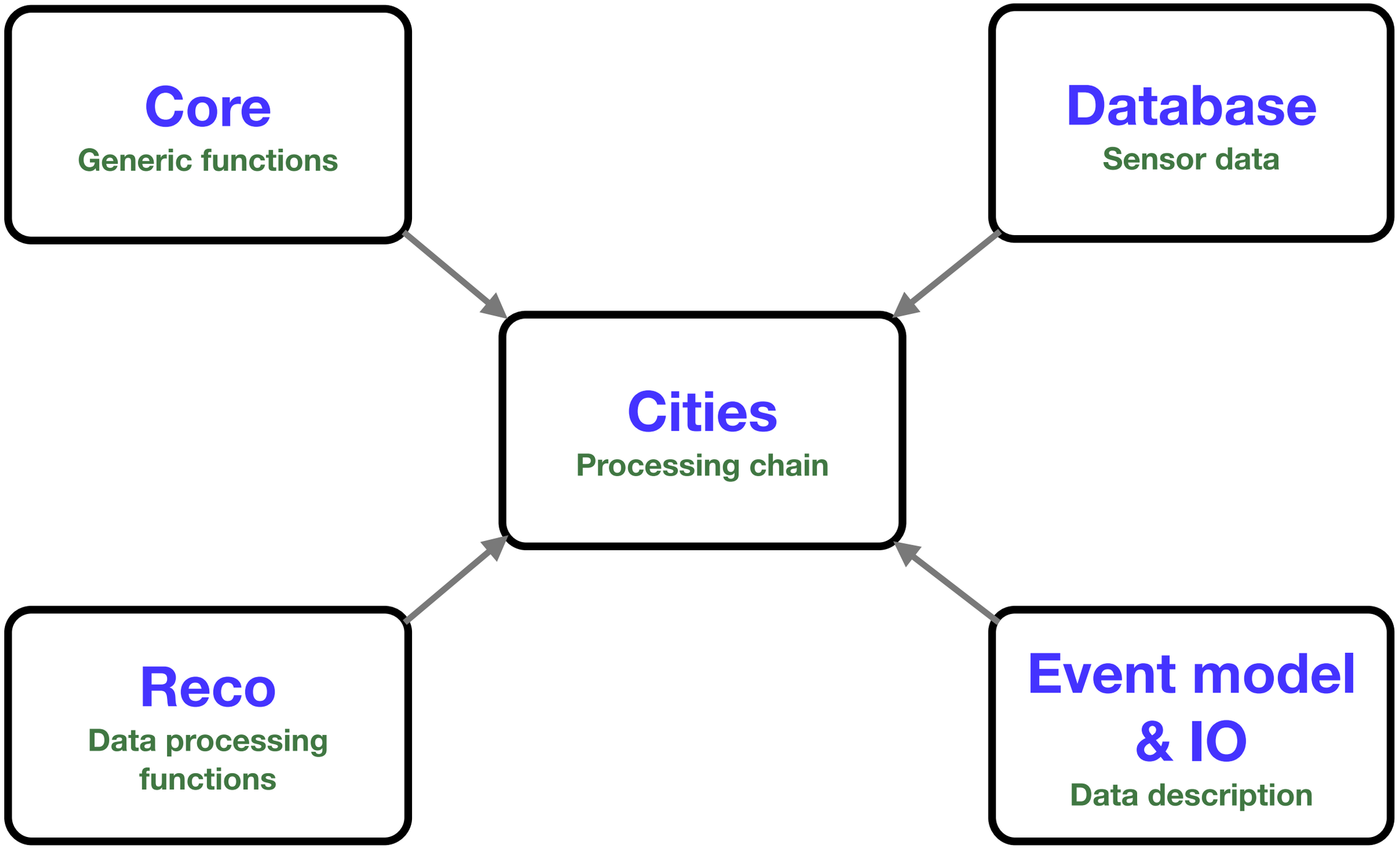
Major dependencies: data storage

- We use the hdf5 library to store our data
    - Flexible and efficient storage
    - Fast I/O operations
    - Optimized memory usage
    - Great python interface with *pytables*
    - Broadly used outside the particle physics community

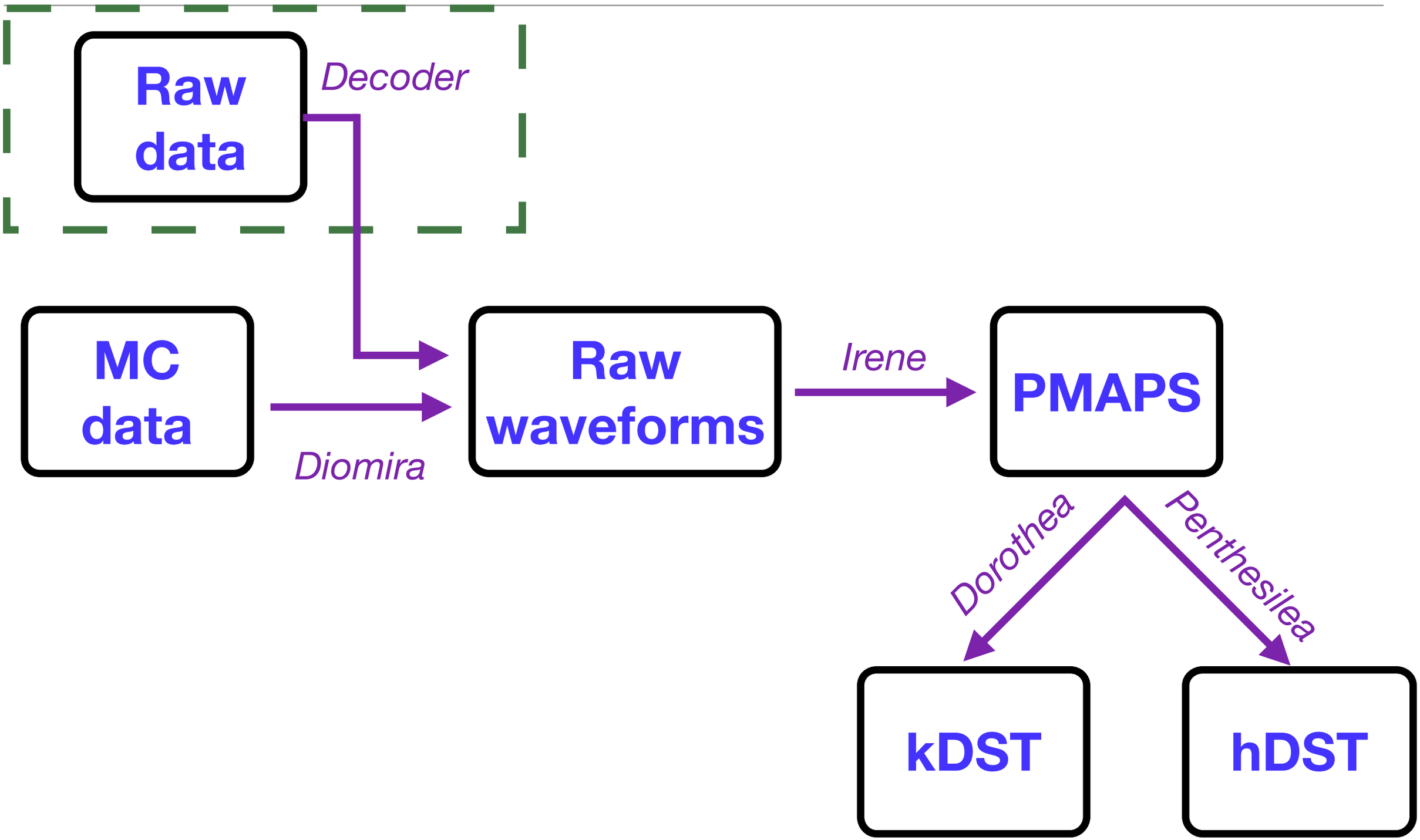
Major dependencies: data analysis

- *scipy* and *pandas* provide most of the tools we need for data analysis
    - Curve fitting
    - Signal processing
    - Pattern recognition
    - Statistical analysis tools
- *matplotlib* provides all the tools for data visualization
- Most importantly
    - All of them work in harmony together

Structure


Core
Generic functions
Database
Sensor data
Cities
Processing chain
Reco
Data processing functions
Event model & IO
Data description

Structure: reconstruction chain (aka cities)



**All the names come from the cities in the book*

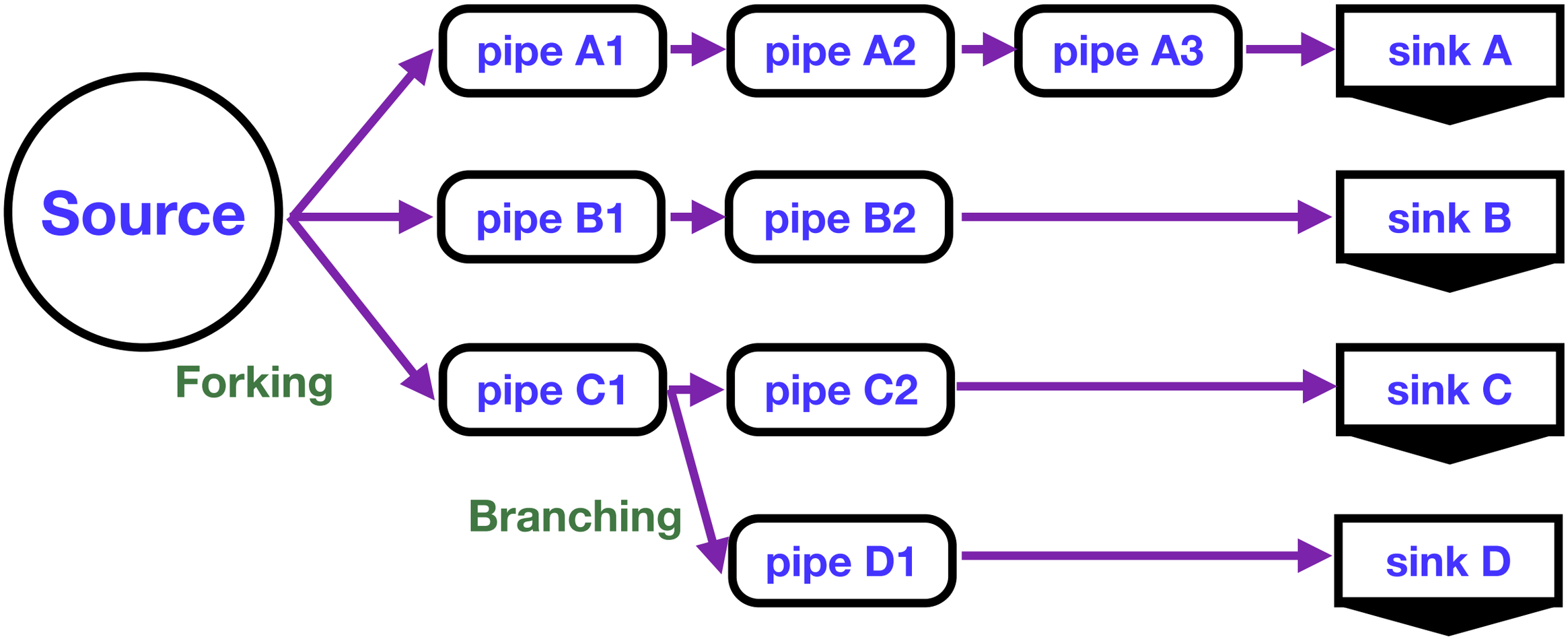
Dataflow

- Coroutine-based structure for data processing
- Functional paradigm
- Pipeline structure
    - Semantically obvious workflow
- Three main components:
    - Sources: feed data into the pipeline
    - Pipelines: transform or filter data
    - Sinks: terminate pipelines. Typically write data to persistent storage or summarize them

Dataflow: features

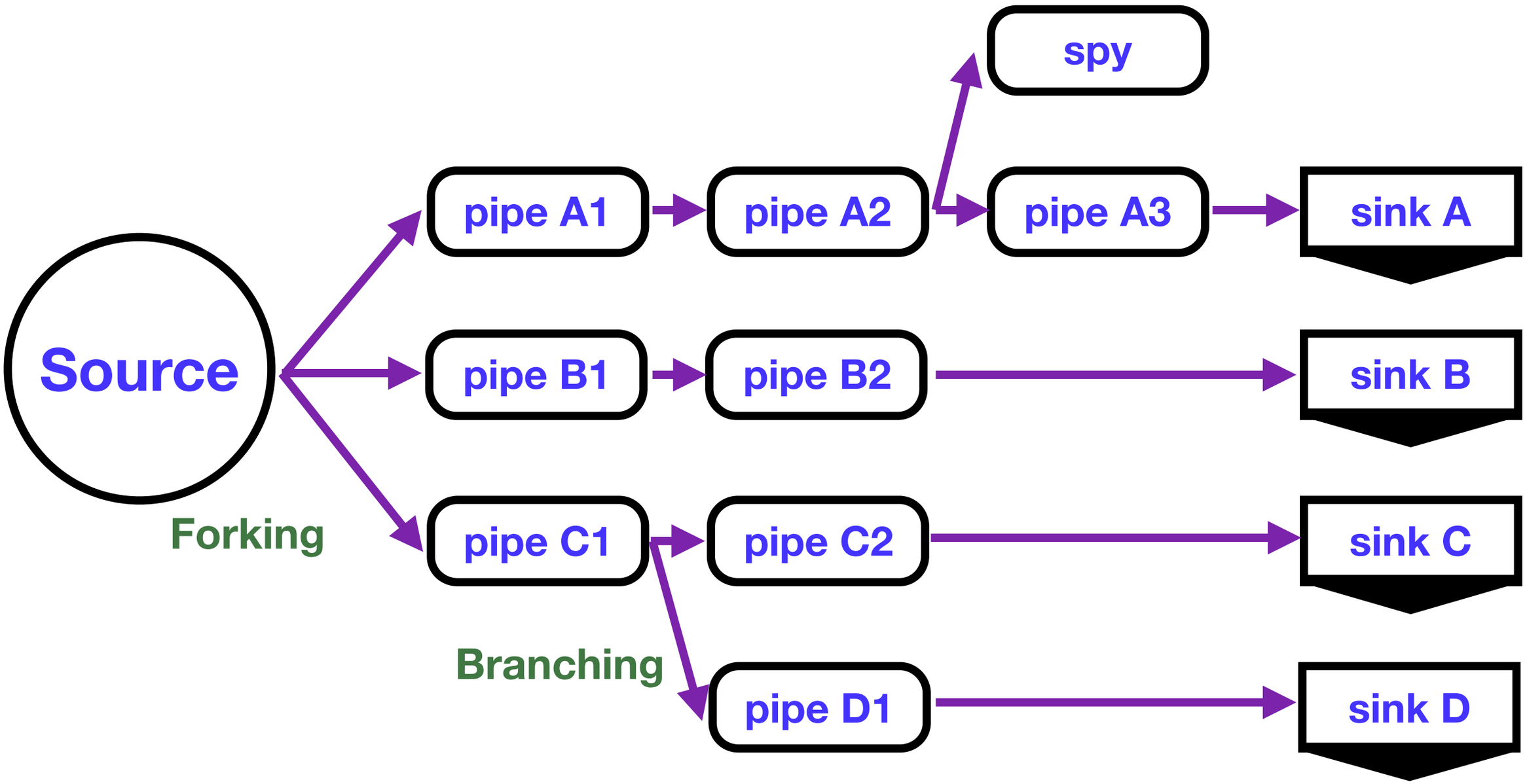
- Split and combine data streams:
    - Forks and branches: replicate the data stream in multiple pipelines
    - Joins: merge pipelines
- Storage of execution variables
    - Futures
- Pluggable structure
    - The components can be added, replaced and removed as needed
    - Small functions can be plugged into the workflow (facilitates testing)
    - Spies: minimalistic branches for data observation
        - Useful for debugging (an example later)

Dataflow: schematics

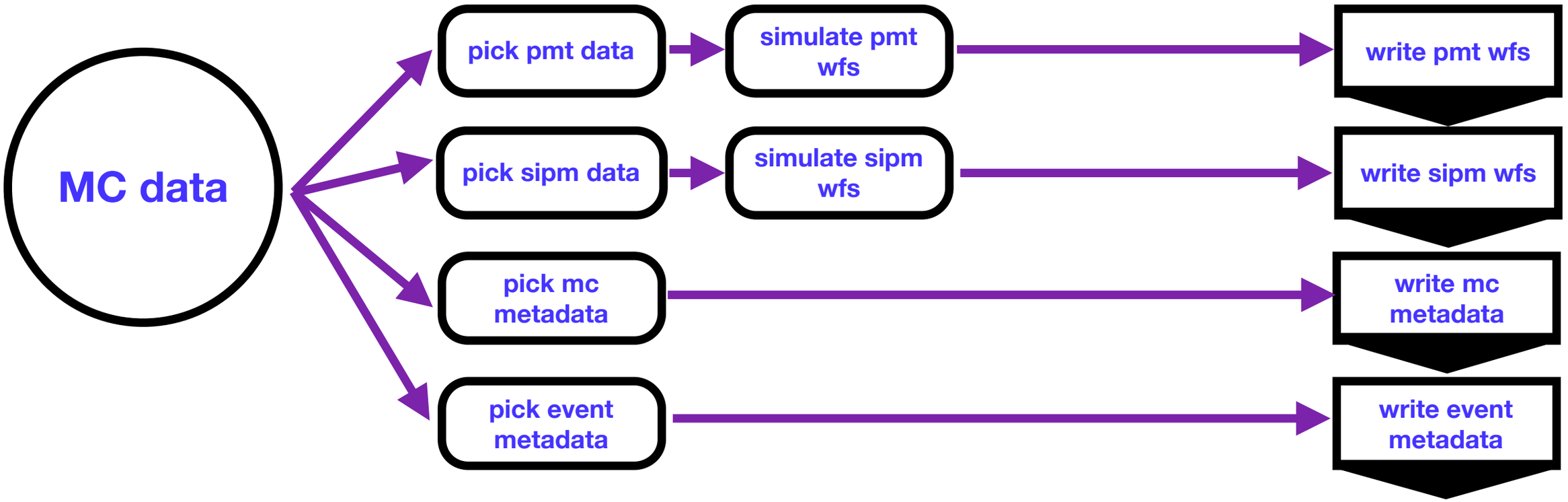


Dataflow: debugging

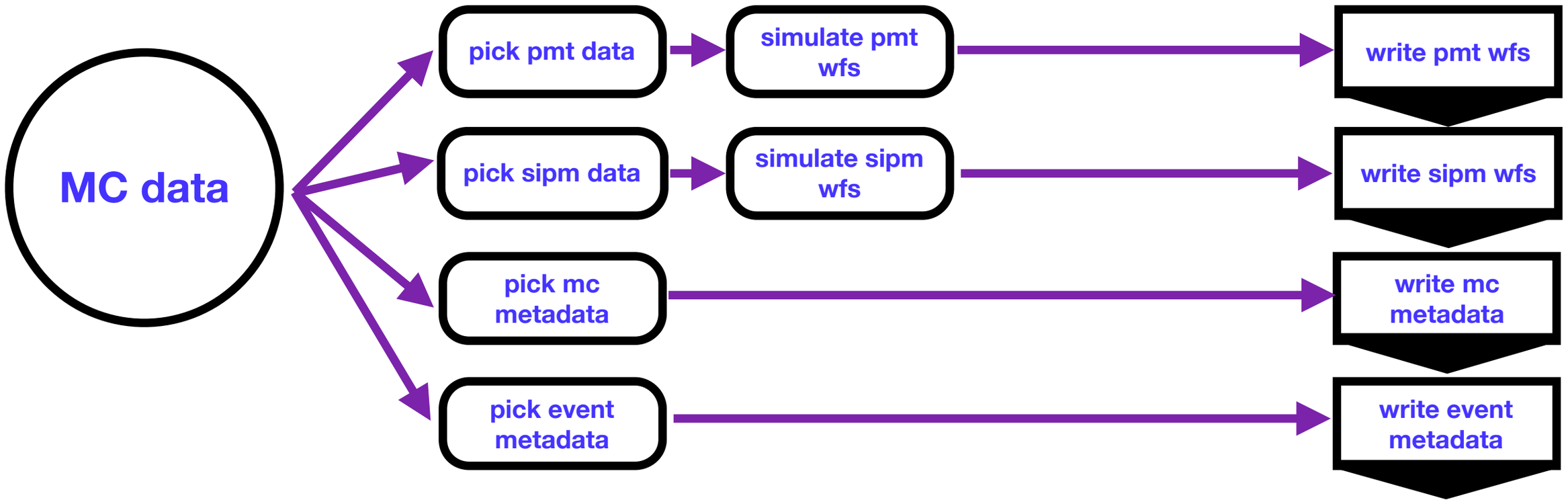
- Spies: designed to obtain information in the middle of a pipeline



Dataflow: an example

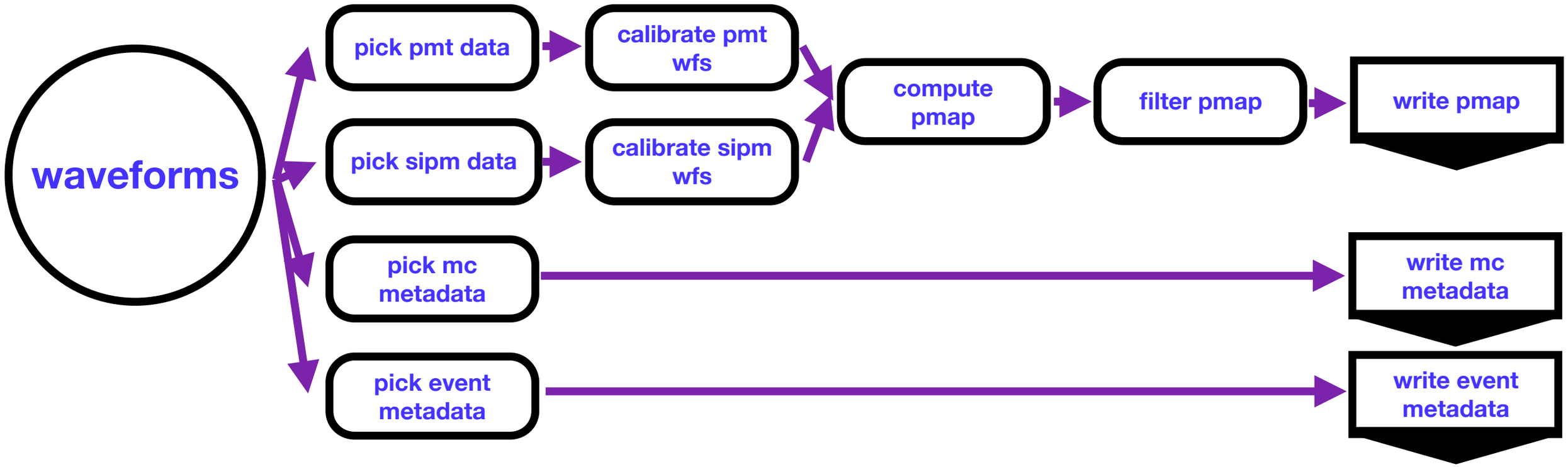


Dataflow: an example




```
return fl.push(
    source = mc_data_reader(files_in),
    pipe   = fl.pipe(fl.slice(*event_range), event_counter.spy, time_execution.spy,
                     fl.fork((  "pmt", simulate_response_pmt , write_pmt           ),
                             ( "sipm", simulate_response_sipm, write_sipm          ),
                             (   "mc",                         write_metadata_mc   ),
                             ("event",                         write_metadata_event)),
    result = dict(events_in  = event_counter .future,
                  total_time = time_execution.future)
        )
```

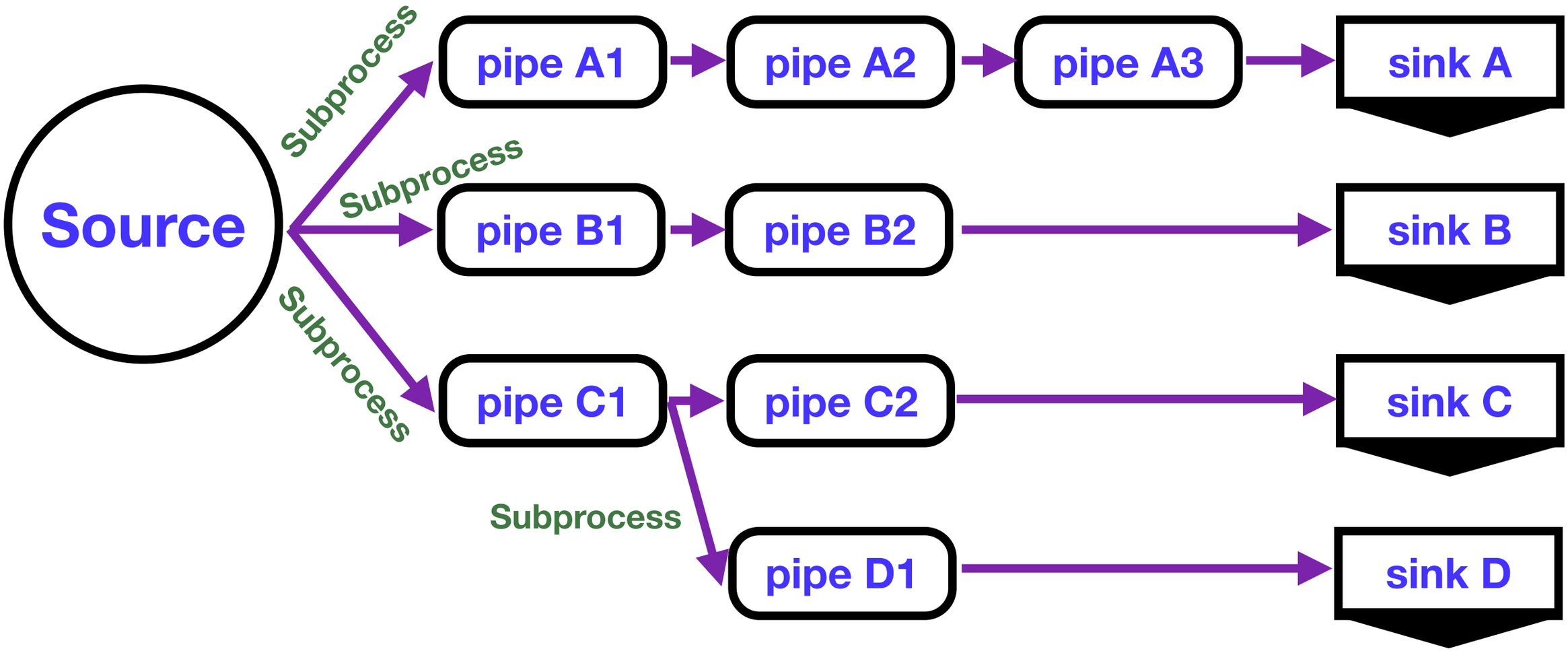
Dataflow: another example




```
return fl.push(
    source = waveform_reader(files_in),
    pipe   = fl.pipe(fl.slice(*event_range), event_in_counter.spy, time_execution.spy,
                     calibrate_pmts ,
                     calibrate_sipms,
                                    compute_pmap,
                                    filter_out_empty_pmaps,
                                    event_out_counter.spy,
                                    fl.fork(( "pmap", write_pmap          ),
                                            (   "mc", write_metadata_mc   ),
                                            ("event", write_metadata_event))),
    result = dict(events_in  = event_in_counter .future,
                  events_out = event_out_counter.future,
                  total_time = time_execution   .future)
       )
```

Dataflow

- An obvious advantage of this scheme is parallelization
  - Asynchronous I/O is needed



Summary

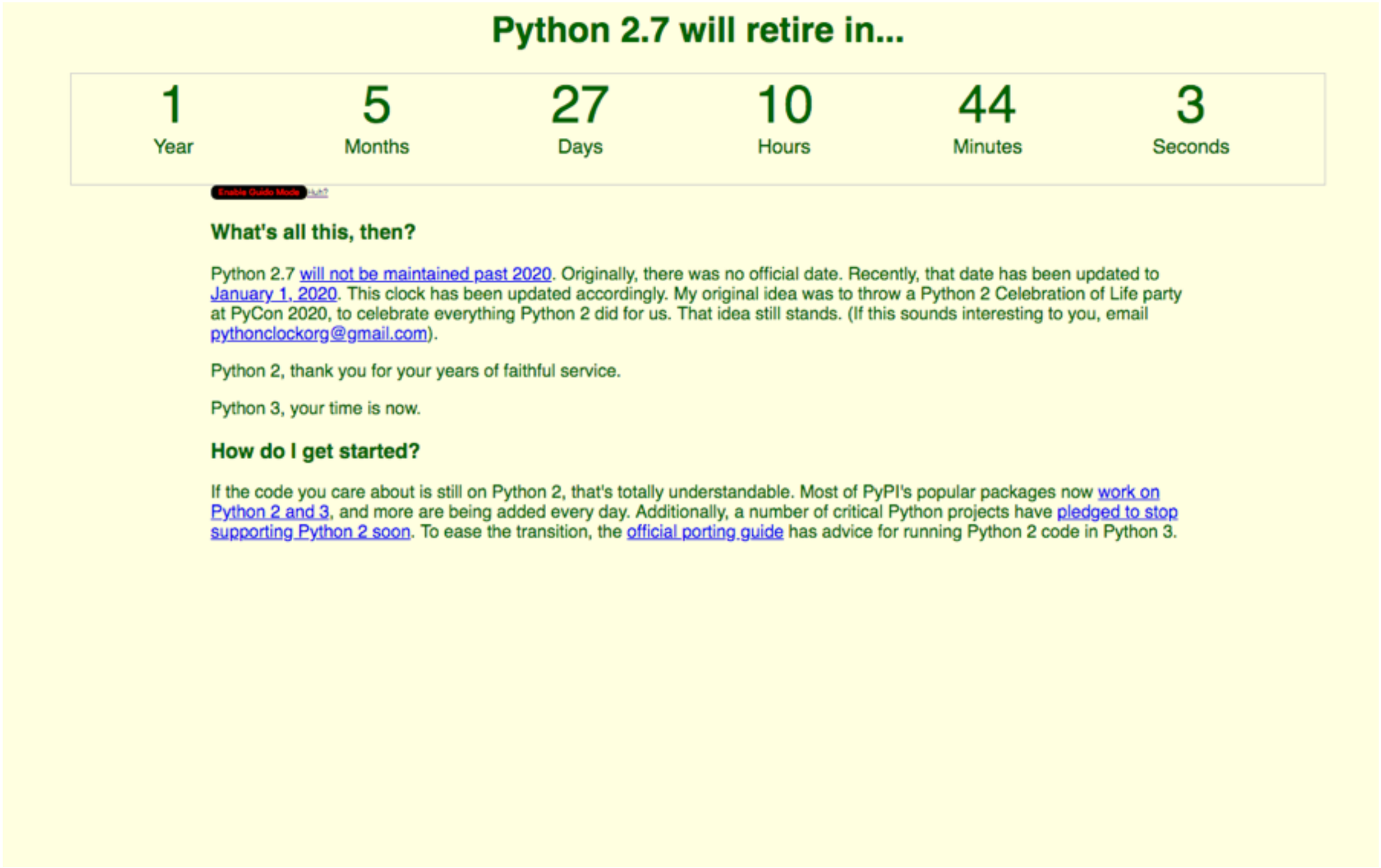
- We have developed a pure-python framework for the NEXT experiment data processing
    - Built around the anaconda ecosystem
    - No C++ or ROOT. Numba/Cython usage found to be marginal
- Exhaustive testing of the code
- Git version control + travis for continuous integration
- We use the *dataflow* scheme
    - Coroutine-based structure
    - Semantically obvious workflow

```
>>> print("\n"*15 + " "*35 + "\033[91m\033[1m" + "Thank you for your attention" + "\033[0m" + "\n"*15)
```

Thank you for your attention

Backup slides

https://pythonclock.org/



Magit

- Magit is a high-level interface to git used from emacs
- It is intuitive and easy to learn and use
- Improves the git use experience by a factor 10^{100}, roughly
- Short, easy-to-remember key combinations for all common operations in git
- Simple, yet formidable visual interface

Magit testimonials

- Thank you Magit for teaching me git.
- Magit profoundly changed my understanding of Git.
- I recommend magit to everyone even if it's the only reason they ever open emacs.
- Every time I use magit: How do I . . . ? It would be really cool if it worked like this. . . Oh! It does!
- Magit is the only git client where I can be faster than on the command line, great stuff.
- Magit being a nice interface to git is the understatement of the year: it's the best interface to git.
- People I work with usually think I'm crazy for using Emacs but everybody is always blown away by magit when they see it.
- This might be the best user interface available to Git anywhere.
- It doesn't just make git easier, or more intuitive, but also makes you a more effective git user.
- Magit made me so much better at using Git, it's ridiculous
- Magit is one of those rare packages which actually help you better understand the tool it provides an interface to.
- Magit allowed me to become very fluid with operations others wouldn't dare considering for their normal workflow while feeling very safe

HDF5 users

- NASA
- Argonne National Laboratory
- Deutsche Bank
- Japan Aerospace Exploration Agency
- Gemini Observatory
- National Oceanographic Data Center
- …

Hypothesis

- https://github.com/HypothesisWorks/hypothesis/tree/master/hypothesis-python
- https://hypothesis.readthedocs.io/en/latest/

Hypothesis

Hypothesis is an advanced testing library for Python. It lets you write tests which are parametrized by a source of examples, and then generates simple and comprehensible examples that make your tests fail. This lets you find more bugs in your code with less work.

e.g.

```
@given(st.lists(
  st.floats(allow_nan=False, allow_infinity=False), min_size=1))
def test_mean(xs):
    assert min(xs) <= mean(xs) <= max(xs)
```

```
Falsifying example: test_mean(
  xs=[1.7976321109618856e+308, 6.102390043022755e+303]
)
```

Hypothesis is extremely practical and advances the state of the art of unit testing by some way. It's easy to use, stable, and powerful. If you're not using Hypothesis to test your project then you're missing out.