Scientific Data Analysis Pipelines -

Push, Pull, React, Or Schedule?

Ami Tavory
Final, Israel

SciPy.in 2012

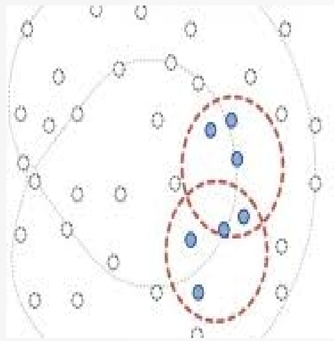
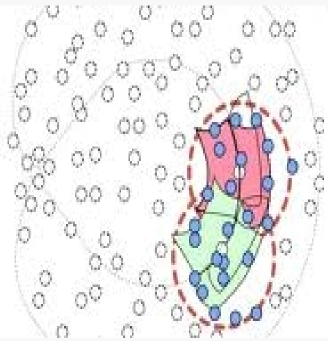
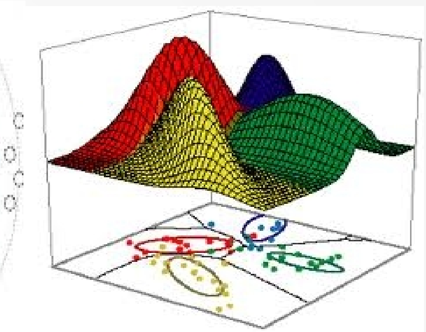
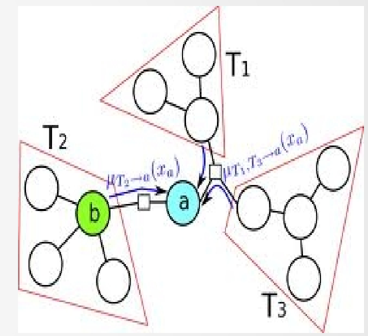
PyConTW 2013

Outline

- Introduction
- Usage Consideration
- Available Language Constructs
- Push, Pull, React, Or Schedule?
- DagPype – A Push-Based Solution
- Conclusions

Complementing
Existing Python Scientific Libraries

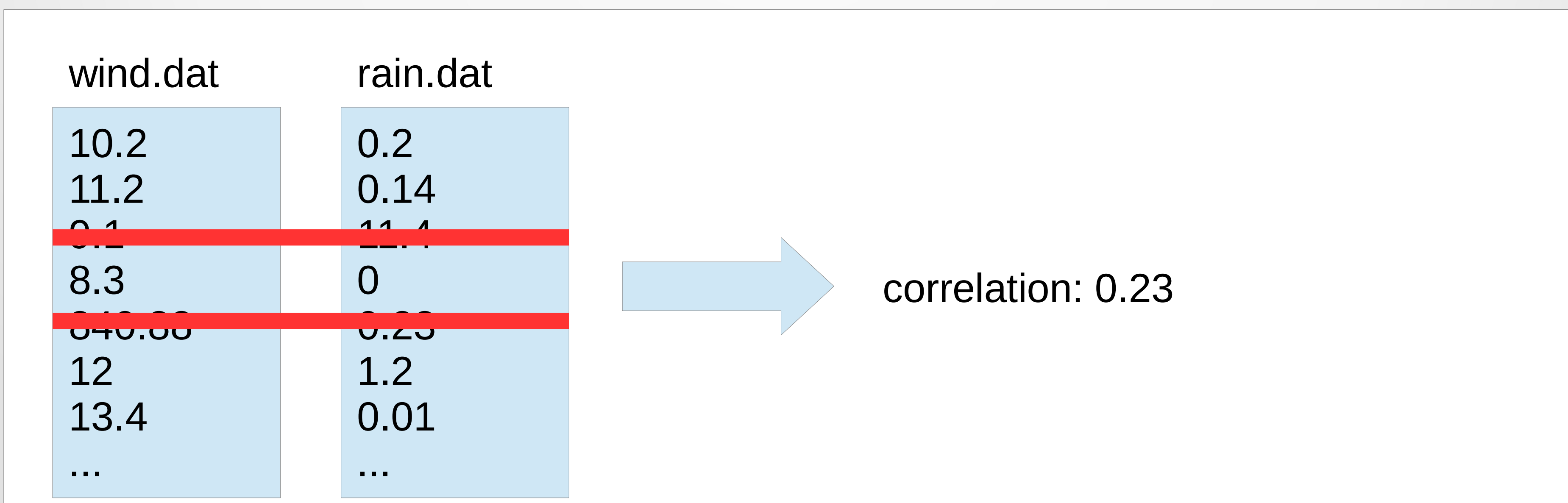
- Once data is assembled, existing Python Scientific libraries are great
  - numpy/scipy, pandas, scikit-learn, mayavi, many more

- Data-analysis processes include several complementary tasks:
  - Data assembly (ugly-format files, outliers, etc.)
  - Extremely quick preliminary analysis
  - Huge-data aggregation (from simple correlation to complex stream algs)

Complementary Tasks: Examples And Attributes

- Examples of common tasks:
  - Find the correlation of the (transformed) content of two files, excluding outliers



Complementary Tasks: Examples And Attributes

- Examples of common tasks:
  - Find the correlation of the contents of two files, excluding outliers
  - Aggregate similar-row CSV columns

meteo.csv

```
day,wind,rain,snow
1,10.2,3.8.4.7
1, 12.2,4.8
1,10.7,6.7
1,9.7,5.2
2,10.4,8.77
2,9,6.2
...
```

[9.97,10,...]

Complementary Tasks: Examples And Attributes

- Examples of common tasks:
  - Find the correlation of the contents of two files, excluding outliers
  - Aggregate similar-row CSV columns
- Common attributes:
  - Complex & flexible combination of relatively simple steps
  - (Existing libraries focus on complex steps)

Pipes

- Pipes are natural for expressing multiple stage connections.
  - This is how we use the shell:

```
$ cat | grep | wc
```

Pipes

- Pipes are natural for expressing multiple stage connections.
- Previous two examples, using extended pipeline syntax:

```
>> wind_rain_corr = stream_vals('wind.txt') + stream_vals('rain.txt') | \
... filt(pre = lambda (wind, rain) : wind < 10 and rain < 10) | \
... corr()

>> day_wind = stream_vals('meteo.csv', ('day', 'wind')) | \
... group( \
...     key = lambda (day, wind, rain, snow) : day,
...     pipe = lambda day : ave_()) | \
... to_array()
```

valid Python expressions – can be used in scripts or even a(n IronPython) prompt.

Plenty Of Reusable Stages

IO

stream_lines (text), stream_vals (csv),
parse_xml (xml),
to_stream (text), vals_to_stream (csv),
np.chunk_stream_vals (csv, numpy), ...

Control

filt, skip, nth, cast, count,
to_list, to_dict, np.to_array,
size_rand_sample, prob_rand_sample
select_inds, grep, trace

Aggregates

min_,max_, sum, count,mean,
stddev, corr, kurtosis

Signal Processing

window_simple_ave, cum_ave, exp_ave,
window_min, window_max, window_quantile
...

Econometrics

...

Plotting

...

Underlying Mechanism?

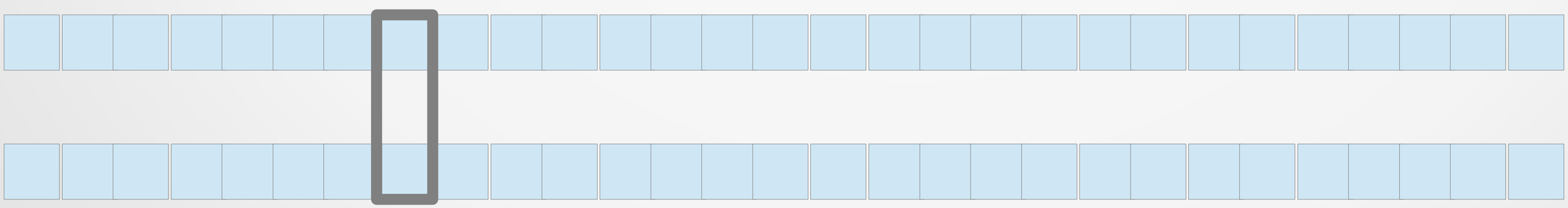
- Which underlying mechanism should drive the pipes?
  - David Beazly (Pycon 08, 09)
- Focus of this talk
- We will:
  - Look at usage considerations
  - Review available language constructs
  - See, using these points, why a coroutine *push*-based solution is necessary

Outline

- Introduction
- Usage Consideration
- Available Language Constructs
- Push, Pull, React, Or Schedule?
- DAGPype – A Push-Based Solution
- Conclusions

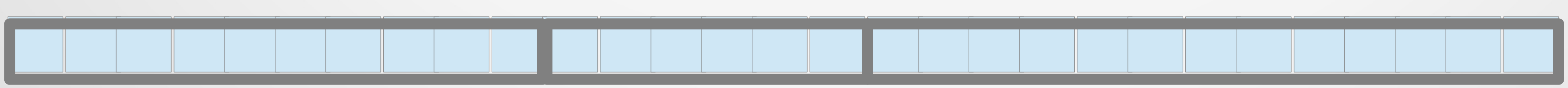
Footprint

- Typical data sets can be huge.
- Each stage should be able to process the data in chunks, not necessarily its entirely.
- Memory use should be proportional to the size of the chunks, not the size of the original dataset.
  - First example (correlation): constant



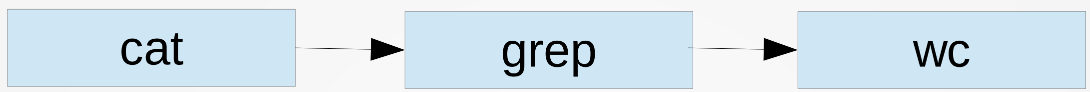
Footprint

- Typical data sets can be huge.
- Each stage should be able to process the data in chunks, not necessarily its entirely.
- Memory use should be proportional to the size of the chunks, not the size of the original dataset.
  - First example (correlation): constant
  - Second example (grouping): proportional to the number of days.

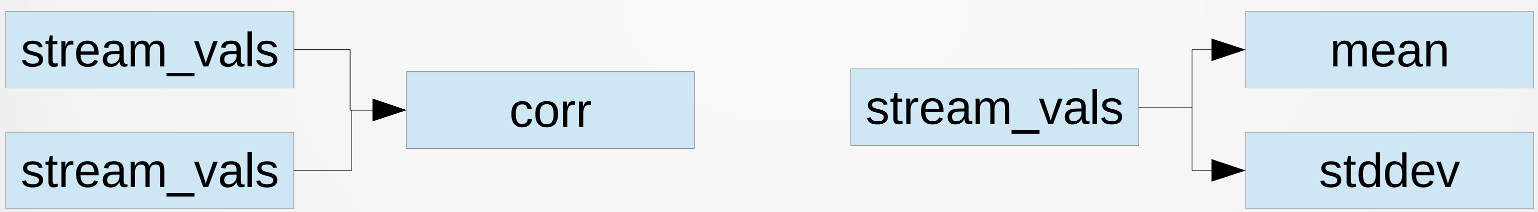


Topolgy - DAGs (Directed Acyclic Graphs)

- Traditional pipelining is linear:
  - Each stage is fed by at most one stage, and feeds at most one stage.



- Data-processing pipelines need more expressiveness.



  - Fanning in, fanning out
- In general, a DAG (directed acyclic graph) topology is needed.

```
source0() + (source1() | filt0()) | \
    filt1() | \
    sink0() + sink1() + (filt2() | sink2())
```

Python Interoperability

- Pipelines are natural for expressing some things, not all
  - Loops, conditionals, functions, classes
- Pipes should work with the rest of the language

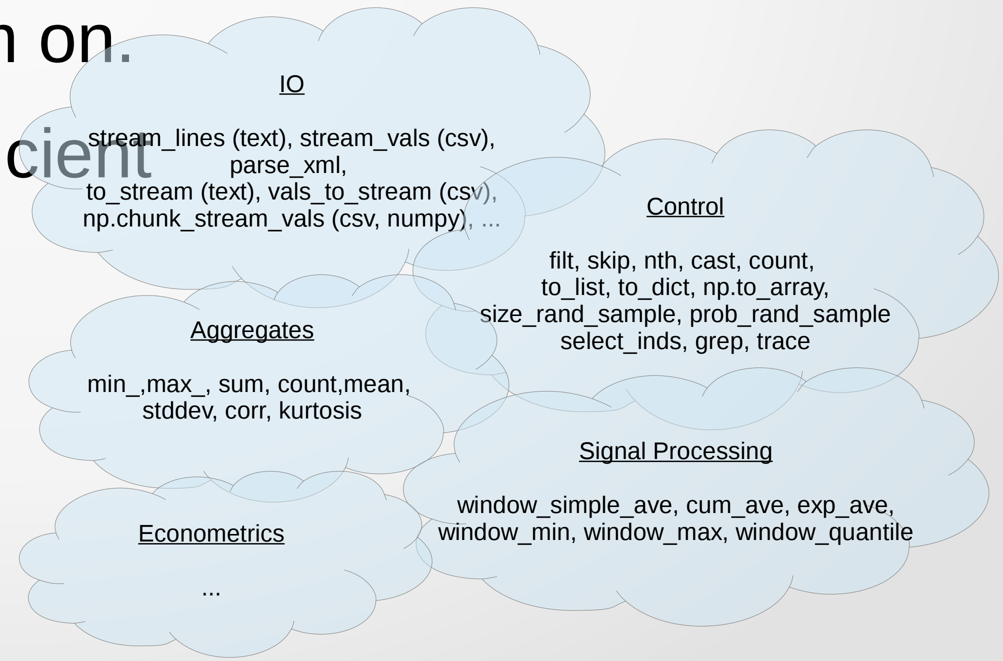
```
means = [ stream_vals('%d_data.dat' % i) | mean() for i in
xrange(10)]
```

Relay / Processing Ratio

- Python pipelines used for many tasks, e.g., web servers

accept_conn → accept_request → authenticate → assemble → reply

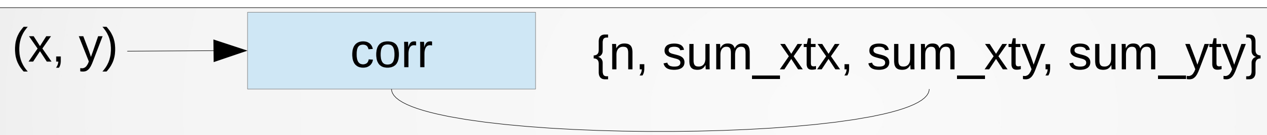
  - Typically have low relay / processing ratio
- Conversely, stages here typically take a few floats, do some operations, send them on.
- Relay should be direct & efficient



Stage State And Synchronization

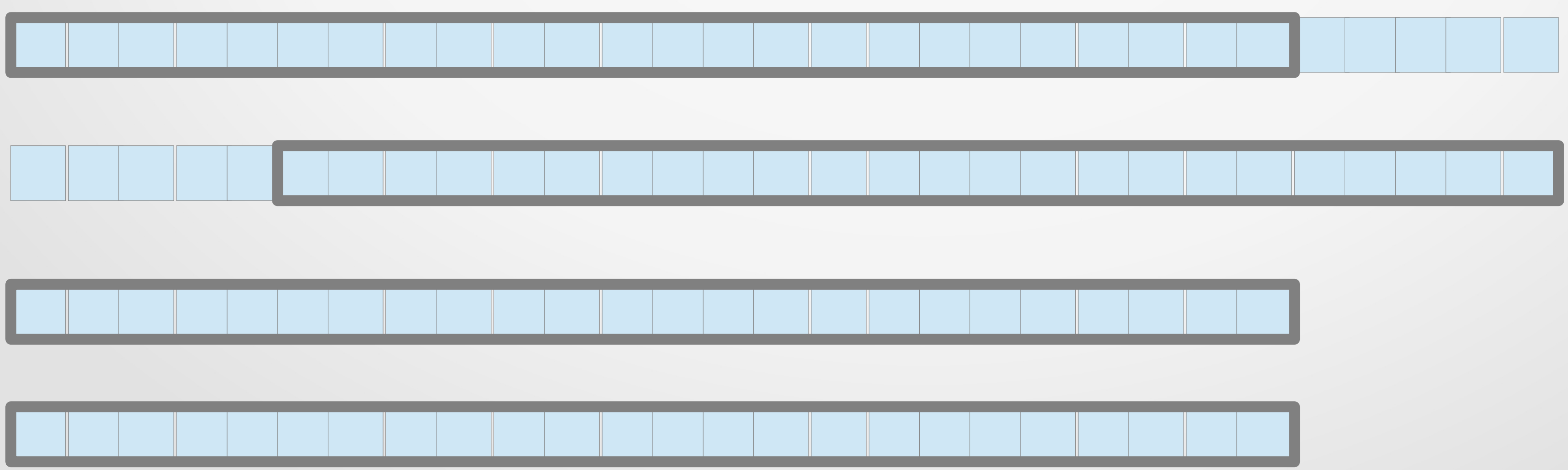
- In general, we must assume that processing stages are stateful...

```
wind_rain_corr = stream_vals(´meteo.csv´, (´wind´, ´rain´)) |
corr()
```



- and unsyncrhonized

```
stream_vals(´meteo.csv´, ´rain´) | relay() + skip(5) | corr()
```

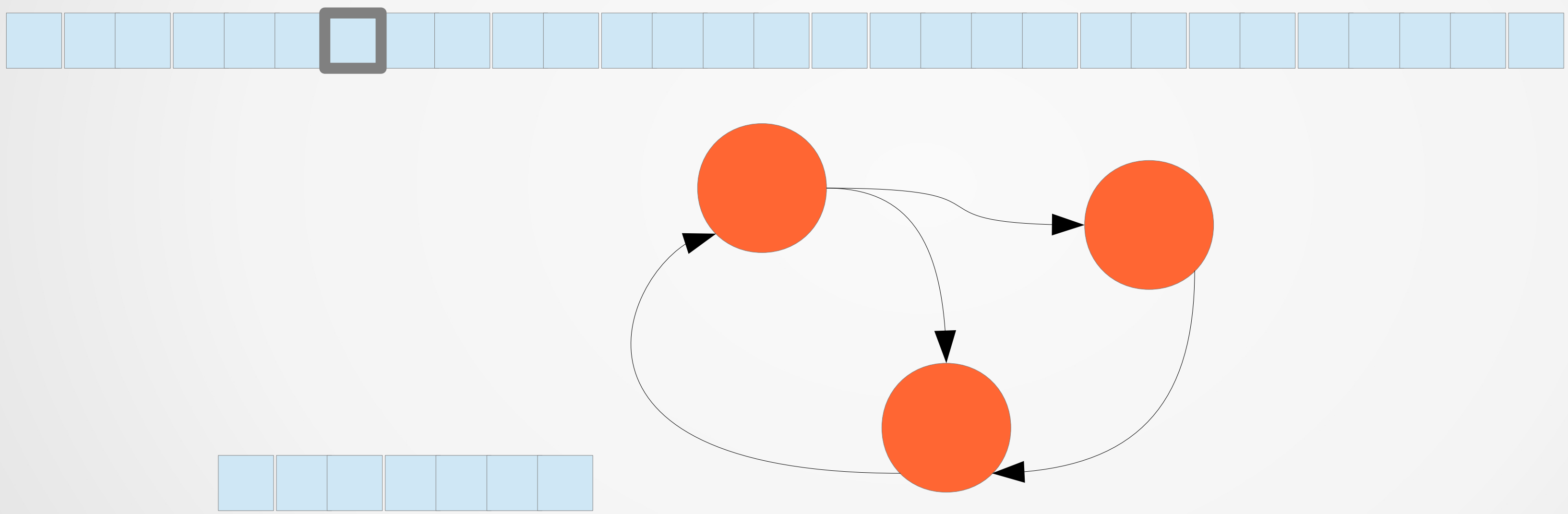


Outline

- Introduction
- Usage Consideration
- Available Language Constructs
- Push, Pull, React, Or Schedule?
- DagPype – A Push-Based Solution
- Conclusions

Starting Point - Statefulness

- As we saw, when stages process elements, they change state



- Which stateful mechanisms does Python offer?

Class Objects - *React*

- Classic OO:
  - State stored in objects
  - Objects *react* through methods

```
class Klass(object):
    def __init__(self, ...):
        ... [Store initial state] ...

    def method(self, ...):
        ... [Update state] ...

    def another_method(self, ...):
        ... [Update state] ...
```

Generators - *Pull*

- Writing style sequential, but execution not

```
def gen(some_sequence, ...):
  ... [set initial state] ...
  for e in some_sequence:
    ... [update state ...]
     yield .. [something depending on s] ...
```

like calling a function

like returning a value … but not quite

- Example: transform … r cumulative sum.
  - [1, 2, 3, 4] → [1, 3, 6, 10]

```
def cum_sum(seq):
   sum_ = 0
   for e in seq:
     sum_ += e
      yield sum
```

Coroutines - *Push*

- Again, writing style sequential, but execution not

```
def co(target, ...):
  ... [set initial state] ...
  try:
    while True:
      e = (yield)
      target.send(... [something] ...)
  except GeneratorExit:
    ... [send something]
    target.close()
```

like being called … but not quite

like calling a function

```
def cum_sum(target):
  sum = 0
  try:
    while True:
      e = (yield)
      sum_ += e
      target.send(sum_)
  except GeneratorExit:
    target.close()
```

Trampoline-Function Coroutines - *Schedule*

- Using coroutines yielding functions, it's possible to build a full-blown scheduler.
- Won't go into it.

Outline

- Introduction
- Usage Consideration
- Available Language Constructs
- Push, Pull, React, Or Schedule?
- DagPype – A Push-Based Solution
- Conclusions

React-Style Stages

- A simple absolute-value stage:
  - [1, -2, -3, 4] → [1, 2, 3, 4]

```
class Abs(object):
    def __init__(self):
        self._has_val = False

    def push(self, e):
        self._has_val = True
        self._val = abs(e)

    def close(self):
        pass

    def has_value(self):
        return self._has_val

    def value(self):
        assert self._has_val
        self._has_val = False
        return self._val
```

needs to be set somewhere

push a value, update one exists

some stages perform after-close ops

stages can be unsynchronized

get a value (assuming one exists)

React-Style Stages

- A simple absolute-value stage:

```
class Abs(object):
    def __init__(self):
        self._has_val = False

    def push(self, e):
        self._has_val = True
        self._val = abs(e)

    def close(self):
        pass

    def has_value(self):
        return self._has_val

    def value(self):
        assert self._has_val
        self._has_val = False
        return self._val
```

  - Code SNR, Runtime

Pull-Style Stages

- A simple absolute-value stage:

```
@pipe
def abs(seq):
    for e in seq:
        yield abs(e)
```

Advantages Of *Pull*-Style Stages

- Simplicity, low overhead

```
@pipe
def abs(seq):
  for e in seq:
    yield abs(e)
```

Advantages Of *Pull*-Style Stages

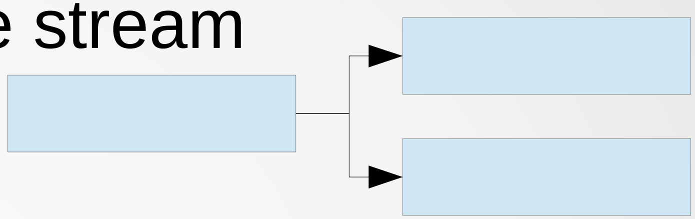
- Simplicity, low overhead
- “Free” left associativity

```
stream_vals('foo.dat') | abs_() | mean()
```

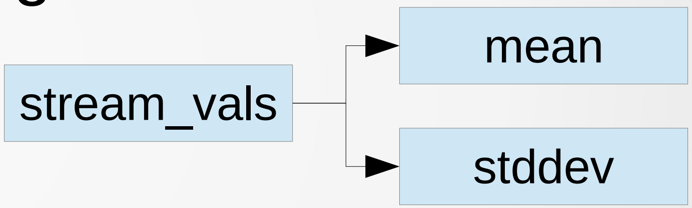
```
@pipe
def abs_(seq):
    for e in seq:
        yield abs(e)
```

Major Disadvantages Of *Pull*-Style Stages

- No efficient fan-out
  - Requires memory loading entire stream

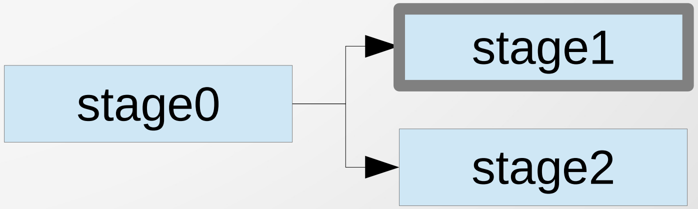


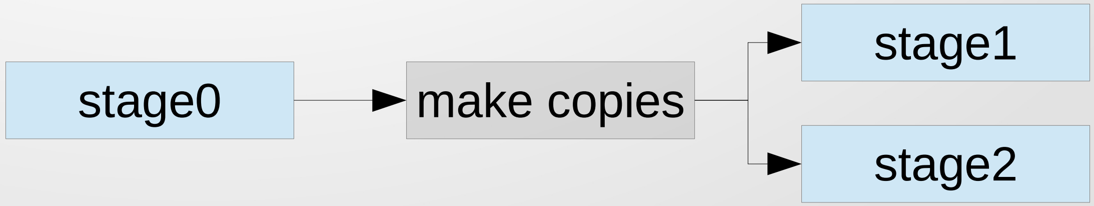
- Implication – no composable aggregates



- Inherent nature of generators

```
for e in seq:
    ... [do something] ...
```





Push-Style Stages

- A simple absolute-value stage:

```
@filter
def abs(target):
    try:
        while True:
            target.send(abs((yield)))
    except GeneratorExit:
        target.close()
```

Assessment Of *Push*-Style Stages

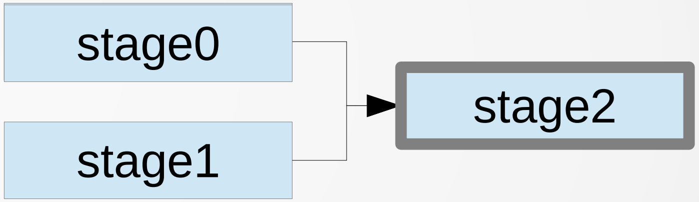
- Clarity and overhead slightly less than *pull,* but much better than *react*
- Natural associativity is right, not left

```
stream_vals('foo.dat') | abs_() | mean()
```

```
@filter
def abs(target):
    try:
        while True:
            target.send(abs((yield)))
    except GeneratorExit:
        target.close()
```

Assessment Of *Push*-Style Stages

- Clarity and overhead slightly less than *pull,* but much better than *react*
- Natural associativity is right, not left
- Slightly unnatural fan-in




```
def stage2(target):
    try:
        while True:
            x, y = (yield)
            ...
    except GeneratorExit:
        ...
        target.close()
```

Outline

- Introduction
- Usage Consideration
- Available Language Constructs
- Push, Pull, React, Or Schedule?
- DAGPype – A Push-Based Solution
  - Some Design Points
  - Examples & Performance
- Conclusions

Lazy Construction

- LTR / RTL discrepancy solution
- During pipeline construction, objects store pipes and fans recursively as lists and tuples.

```
source0() + (source1() | filt0()) | \
   filt1() | \
   sink0() + sink1() + (filt2() | sink2())
```

```
[source0]
```

```
[source1]
```

```
[filt0]
```

Lazy Construction

- LTR / RTL discrepancy solution
- During pipeline construction, objects store pipes and fans recursively as lists and tuples.

```
source0() + (source1() | filt0()) | \
  filt1() | \
  sink0() + sink1() + (filt2() | sink2())
```

```
[source0]
```

```
[source1, filt0]
```

Lazy Construction

- LTR / RTL discrepancy solution
- During pipeline construction, objects store pipes and fans recursively as lists and tuples.

```
source0() + (source1() | filt0()) | \
  filt1() | \
  sink0() + sink1() + (filt2() | sink2())
```

```
[(source0, [source1, filt0])]
```

Lazy Construction

- LTR / RTL discrepancy solution
- During pipeline construction, object store pipes and fans recursively as lists and tuples.

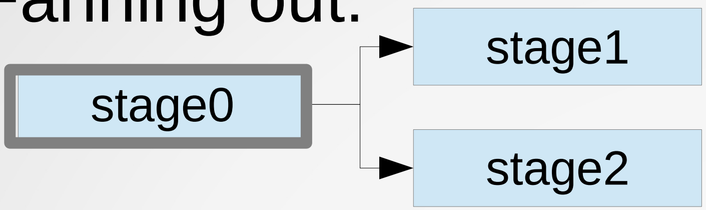
```
source0() + (source1() | filt0()) | \
  Filt1() | \
  sink0() + sink1() + (filt2() | sink2())
```

```
[(source0, [source1, filt0]), filt1, (sink0, sink1, [filt2, sink2
])]
```

- Source->sink connections trigger actual calculations.

Fanning Out And In

- Fanning out:

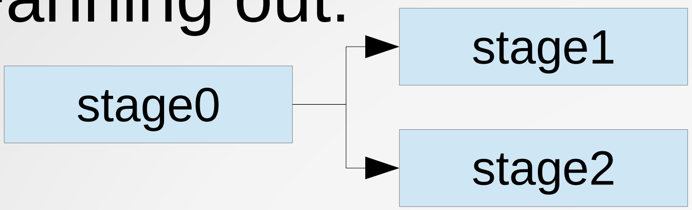


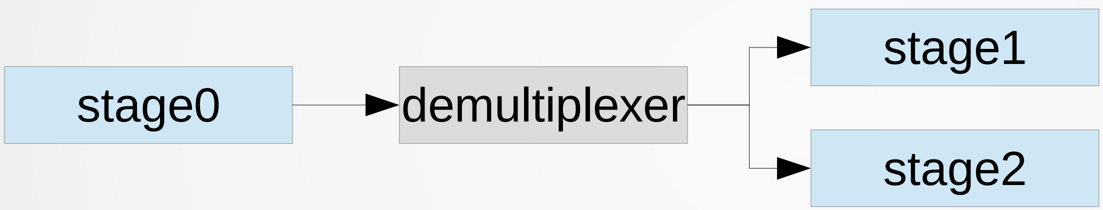

```
stage0 | stage1 + stage2
```

```
def stage0(target):
  while True:
    e = (yield)
    target.send( fn(e) )
```

Fanning Out And In

- Fanning out:

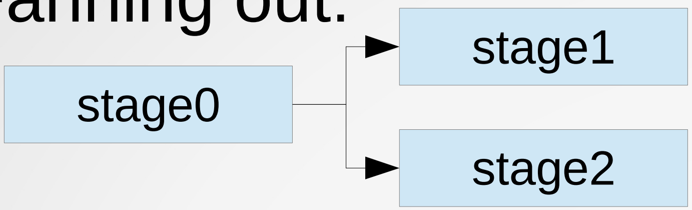




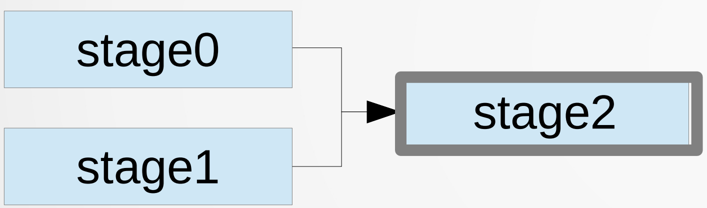

```
def _demux(*targets):
  …
  while True:
    e = (yield)
    for t in *targets:
        t.send(e)
```

Fanning Out And In

- Fanning out:



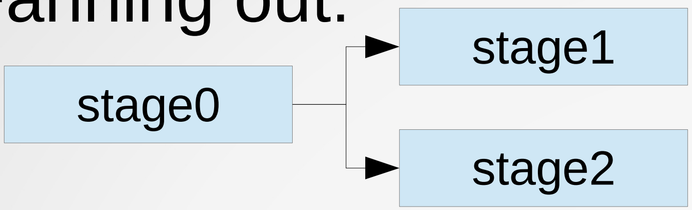
- Fanning in:



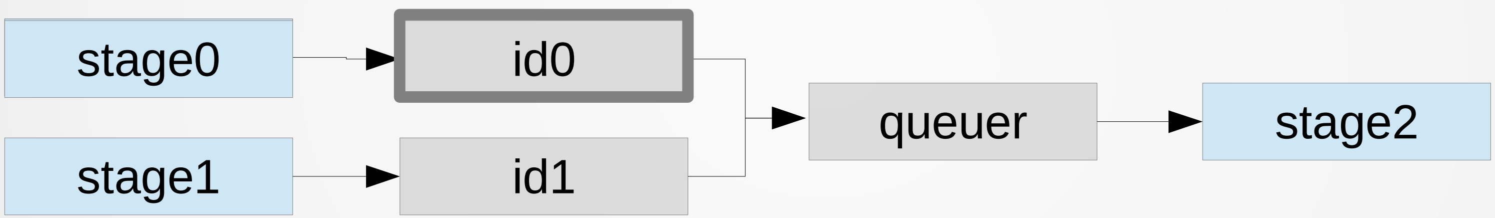

```
def stage2(target):
  ...
  while True:
     x, y = (yield)
```

Fanning Out And In

- Fanning out:



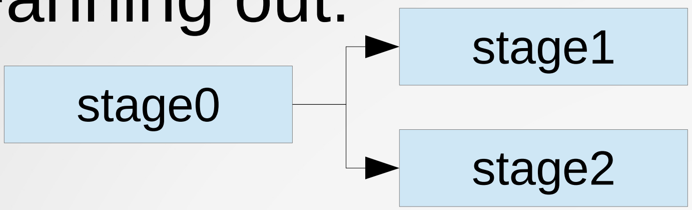
- Fanning in:



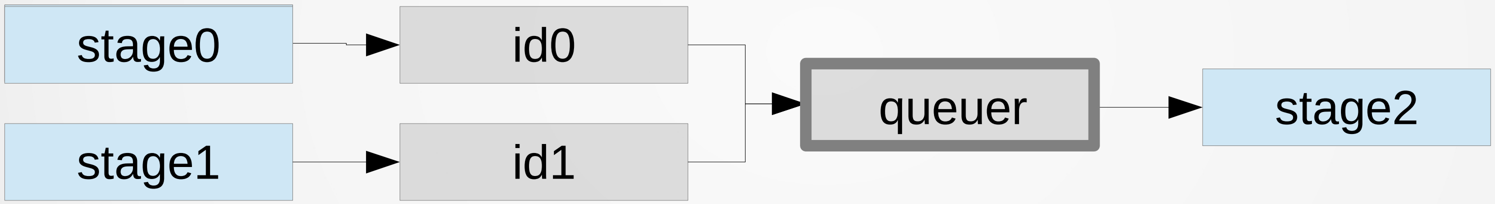

```
def _id(target, id):
 ...
  while True:
    target.send((id, (yield)))
```

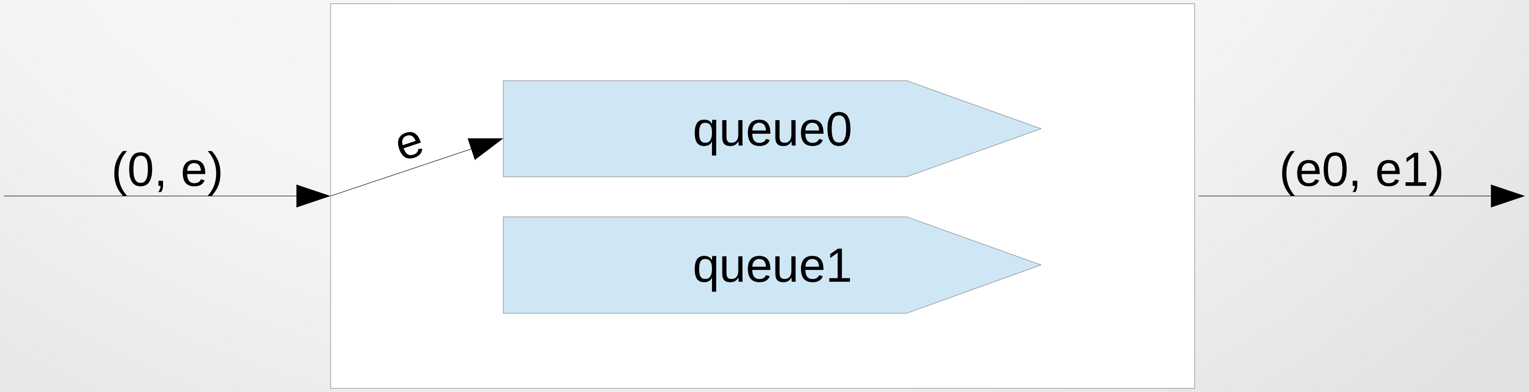
Fanning Out And In

- Fanning out:



- Fanning in:



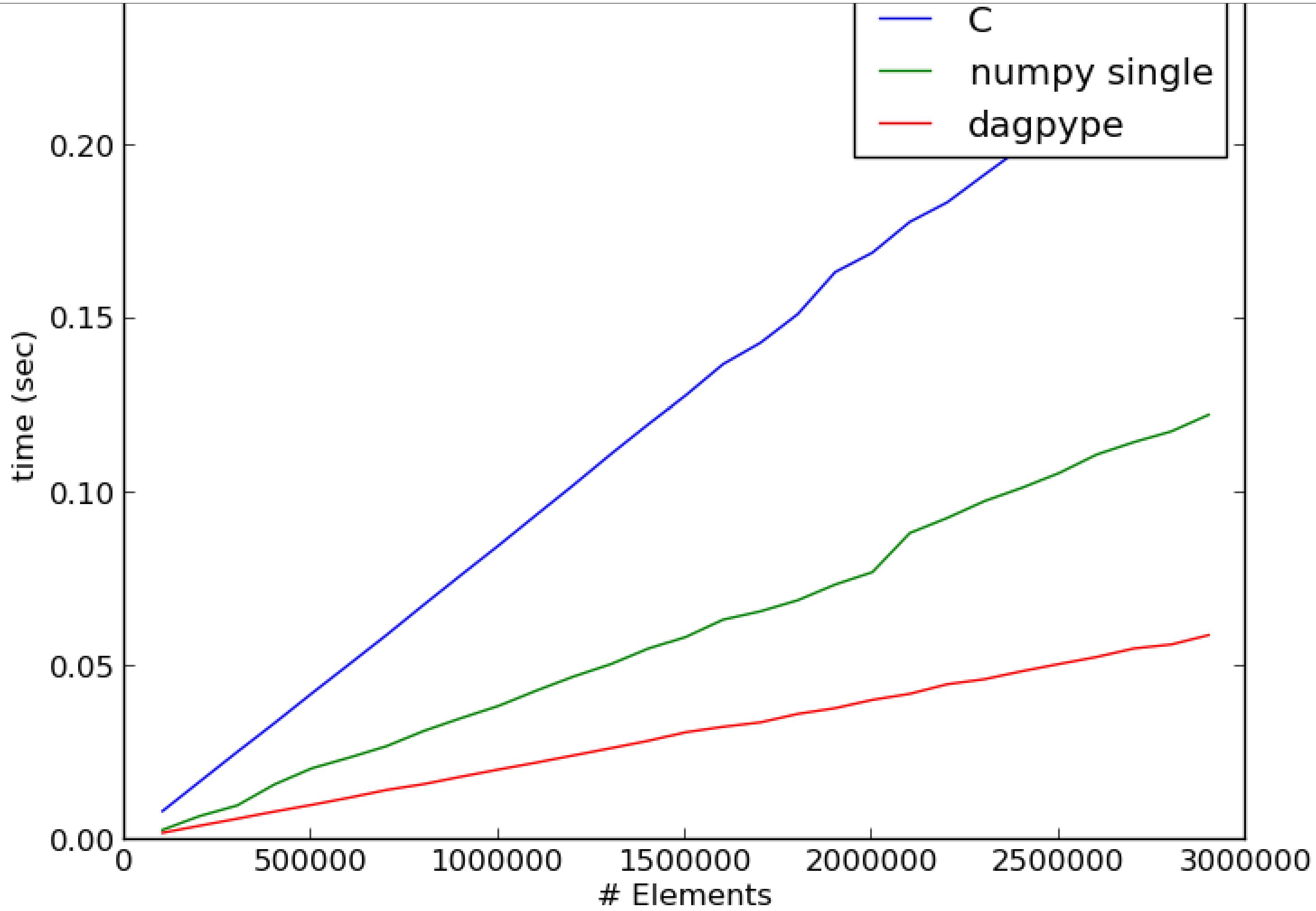


Outline

- Introduction
- Usage Consideration
- Available Language Constructs
- Push, Pull, React, Or Schedule?
- DagPype – A Push-Based Solution
  - Some Design Points
  - Examples & Performance
- Conclusions

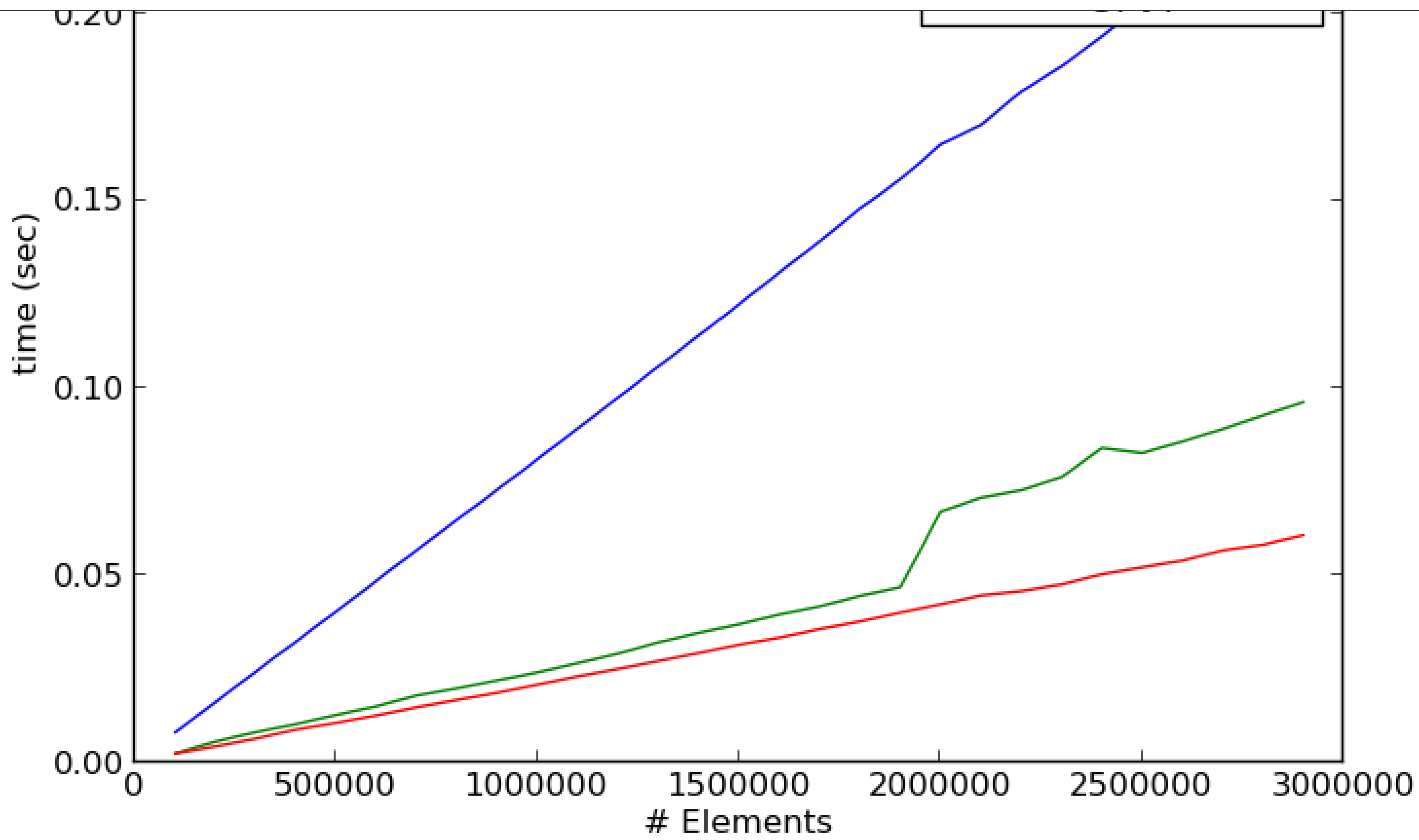
Performance - Binary File Correlation

```
c = np.chunk_stream_bytes('foo.dat', num_cols = 2) | np.corr()
```

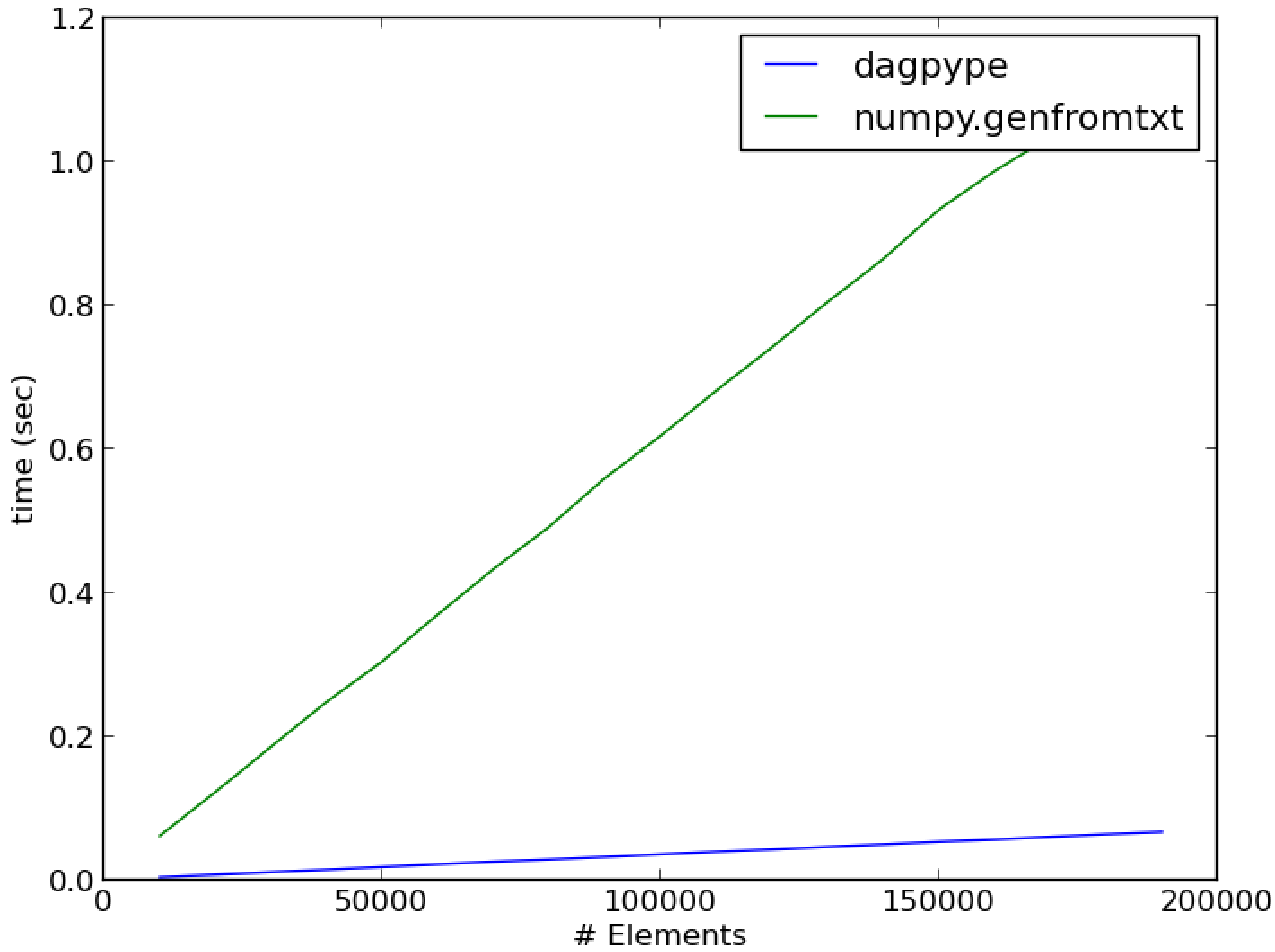


Performance -
Pruned Binary File Correlation

```
c = np.chunk_stream_bytes('foo.csv', num_cols = 2) | \
    filt(lambda a : a[numpy.logical_and(a[:, 0] < 0.25, a[:, 1] <
0.25), :]) | \
    np.corr()
```



Performance -
CSV File Mean



Outline

- Introduction
- Usage Consideration
- Available Language Constructs
- Push, Pull, React, Or Schedule?
- DagPype – A Push-Based Solution
- Conclusions

Main Points Covered

- (Python) pipelines can complement existing great Python libraries and tools
  - A large fraction of (at least my) time is spent
    - trying to figure out what to learn ...
    - out of horrible (noisy) data ...
    - in terrible formats

Main Points Covered

- (Python) pipelines can complement existing great Python libraries and tools
- *Push*-based coroutines are the way to implement them in Python

https://pypi.python.org/pypi/DAGPype/

Points Not Covered

- Chunk granularity
- Parallelism
- Operations beyond DAG-pipelines' expressiveness

Thanks!

- Thank you for your time!