Functions and Graphs

Dr. Scott Gorlin
Harvard University
Fall 2018

Agenda

- Avoiding the `for` loop
- Functional Programming
- The Dag
- Dask
- Luigi
- Pset 2
- Readings

Avoiding the for loop

Primitives

Everyone knows this is bad:

```
def vecsum(vec):
    s = 0
    for val in vec:
        s += val
    return s
```

… when you can use `sum(vec)` or `vec.sum()`!

It's not just about avoiding `for`

It's not about speed

It's about using the right ***functional*** primitive!

Vectorizing

You probably already know to vectorize your code

You should ***rarely*** need to index a numpy array

See: Numpy ufuncs

```
>>> a
array([ 0. ,  0.5,  1. ])

>>> 2**a
array([ 1., 1.41421356, 2.])

>>> np.array([[0, 1]]) \
  + np.array([[0], [1]])


array([[0, 1],
       [1, 2]])
```

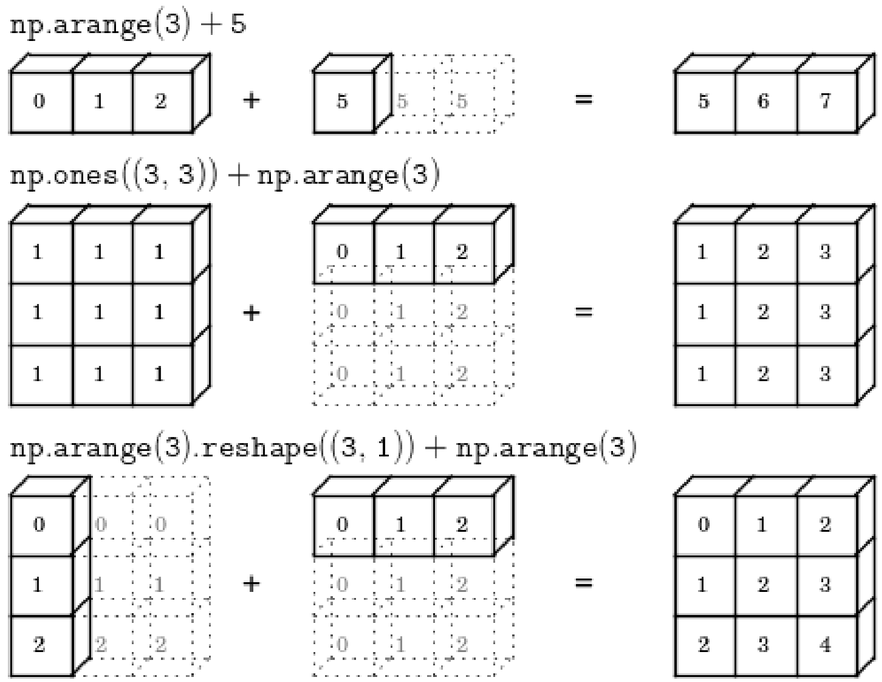
An Advanced Scientist knows how to leverage Broadcasting and ufuncs to solve most problems

This flexibility comes at the cost of extra backend work. We will explore fixing that later in the course, but you ***usually*** shouldn't care.

Vectorizing

Broadcasting…

Intuitively replicates vectors for elementwise operations



astroml

Vectorizing

… and pandas too!

```
>>> df
   x1  x2  y1  y2
0   1   2   4   2
1   2   1   1   3
2   1   3   2   2
3   3   1   1   1

>>> df['dist'] = np.sqrt(
    (df['x1'] - df['x2'])**2
    + (df['y1'] - df['y2'])**2)
```

Vectorizing

You can control the output buffer and avoid extra copying of the data

... but this can sacrifice clarity and is usually not recommended!

```
kwargs = {
    'globals': {'np': numpy},
    'setup': 'a = np.zeros(10000)',
}
# 20% speedup!
>>> timeit("a = .9*(a + 1)", **kwargs)
8.804216878954321
>>> timeit("a += 1; a *= .9", **kwargs)
7.309655925957486
>>> timeit("np.add(a, 1, out=a); np.multiply(a, .9,
out=a)", **kwargs)
7.435892655048519
```

Iteration Primitives

Other methods:

Pure python

```
# Comprehensions
b = [ _ + 1 for _ in a ]      # List
c = { str(_): _ for _ in a }  # Dict
uniques = { _ for _ in a }    # Set

# Careful... this is a generator!
d = (_ + 1 for _ in a)  # not a tuple!

# Functional
map(some_func, iterable)
filter(some_func, iterable)
functools.reduce(some_func, iterable)
```

Pandas/Numpy

```
df = DataFrame(...)

for index, row in df.iterrows():
    ...

df.apply(some_func)  # Apply func elementwise!

# For every ufunc
numpy.add.reduce(vec)

# If desperate...
numpy.vectorize(func)
numpy.frompyfunc(func, nin, nout)
```

Iteration Primitives

Generator

```
def iter_range(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

Zip

```
>>> pairs = zip(vec_a, vec_b, ...)
[(a1, b1, ...), (a2, b2, ...), ...]

>>> zip(*pairs)
[(a1, a2, ...), (b1, b2, ...), ...]
```

Map

```
map(f, iter) === (a(_) for _ in iter)
```

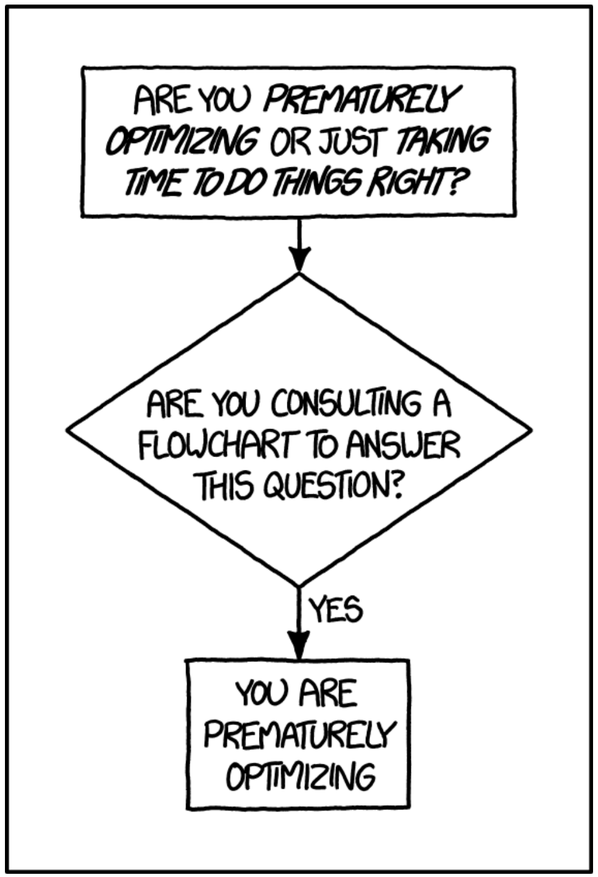
Filter

```
filter(f, iter) === (_ for _ in iter if f(_))
```

Iteration Primitives

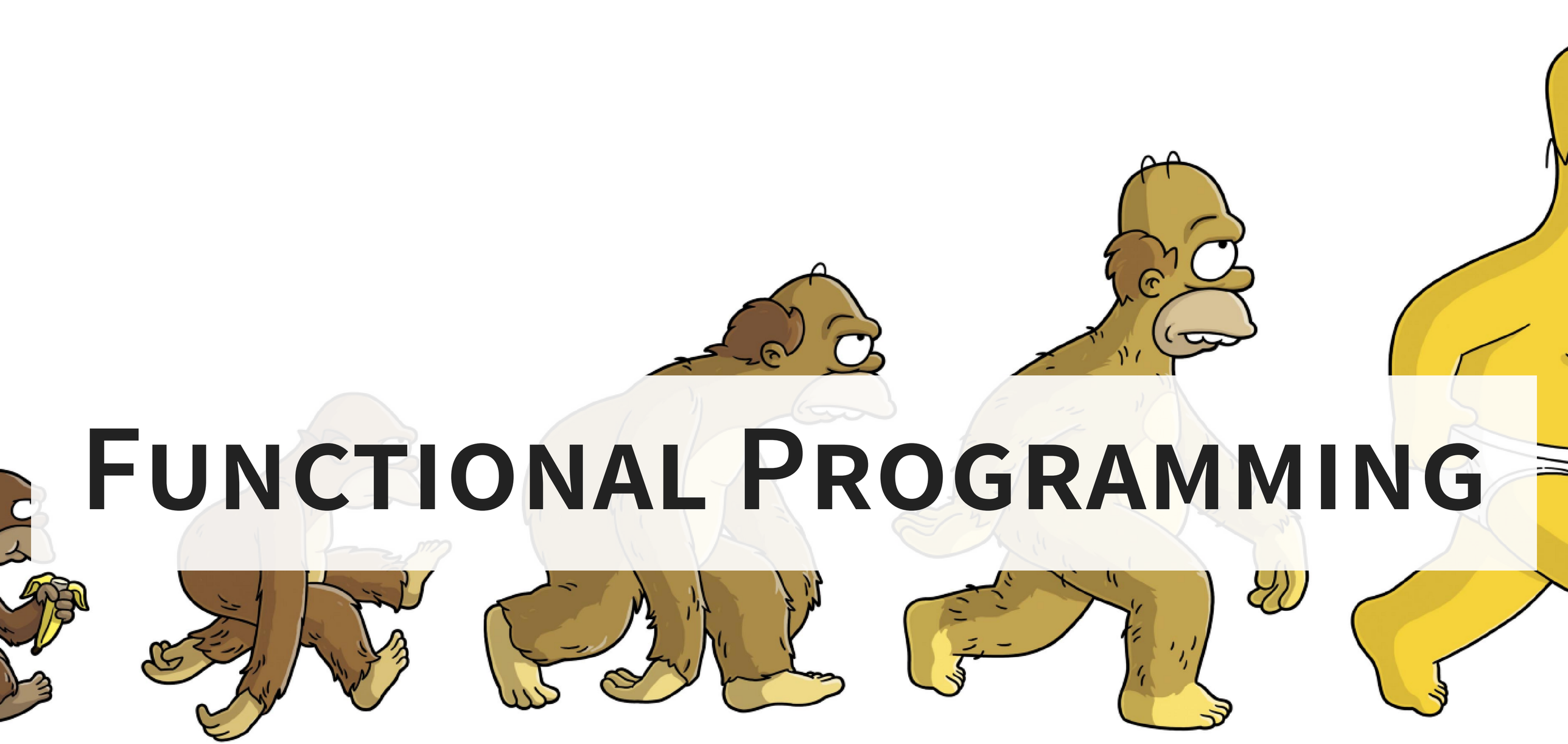
	Speed	Memory	Debugging	Clarity
`for` loops	Bad	$O(1)$	Good	OK
`iterrows()`	Ok	$O(1)$	OK	OK
Comprehensions	Ok	$O(N)$	OK	Good
`apply()`	Fast	$O(N)$	Harder	Good
Series/vector operation	Faster	$O(N)$	Good	Good
Custom ufunc	Fastest	$O(N)$	Hardest	Bad
Custom compiled reduction	Fastest	$O(1)$	Hardest	Bad

Don't be evil!



Premature optimization is the root of all evil XKCD 1691

Functional Programming



Code as Data

Try to write every program as an operation on data, cf:

Good **Procedural**

```
s = 0
s += int('1')
s += int('2')
```

Better **Structured**

```
s = 0
for arg in ['1', '2']:
    s += int(arg)
```

Best **Functional**

```
sum(
    map(int, ['1', '2'])
)
```

Functions are first-class objects!

Functional Programming

Functional

Functions (and Classes) are first-class citizens.

They can be assigned to variables, passed to other functions, or modified.

Declarative

Express what we want *a la* math, primitives

Cf: imperative statements, which express an action to carry out

Stateless

Output depends only on arguments, and invokes no side effects.

This is less universally accepted as The Right Thing.

Functional Programming

*In **declarative** languages, you write a specification that describes the problem to be solved, and the language implementation figures out how to perform the computation efficiently.*

***Functional** programming decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don't have any internal state that affects the output produced for a given input.*

Functional Programming HOWTO

State

Stateful

```
def f(some_list, another):
    some_list.extend(another)
```

```
# Complex data is well handled by OOP
model = GLM()
model.train(data)
model.predict(validation)
```

Stateless

```
def f(some_list, another):
    return some_list + another
```

```
# ... but can be awkward in functional
model = GLM()
coefs, stats = model_logic.train(data)
model.predict(coefs, validation)
```

Example - Reduction

Procedural

```
s = 0
for v in vec:
    s += v
```

Functional

```
sum(vec)
# Or the equivalent long form...
functools.reduce(operator.add, vec)
```

Which may be implemented as:

```
def reduce(func, vec):
    n = len(vec):
    # Parallelize?
    return func(
        reduce(func, vec[:n/2]),
        reduce(func, vec[n/2:])
    )
```

Example - Reduction

But Python is single threaded. So what?

```
numpy.sum(vec)
numpy.add.reduce(vec)
dask.DataFrame().aggregate()
```

Py 3 Constructs

Iterators

```
import itertools
zip
map(f,
    filter(g, iterable)
)
```

… can work with memory $O(1)$ and *infinite or unknown length* iterables

Comprehensions

```
[f(x) for x in vec if g(x)]
```

… are actually a bit faster but memory $O(N)$.

Py 3 Constructs

map's an iterator!!

Py2 `list`

```
>>> m = map(lambda x: x, [1, 2, 3])
>>> m
[1, 2, 3]
>>> list(m)
[1, 2, 3]
>>> list(m)
[1, 2, 3]
```

Py3 ITERATOR

```
>>> m = map(lambda x: x, [1, 2, 3])
>>> m
<map object at 0x10ae90860>
>>> list(m)
[1, 2, 3]
>>> list(m)
[]  # Already consumed!
```

Py 3 Constructs

Don’t abuse the side effects!

```
>>> map(print, range(5))
<map object at 0x10e40a5c0>
>>> list(_)
0
1
2
3
4
[None, None, None, None, None]
```

The next level

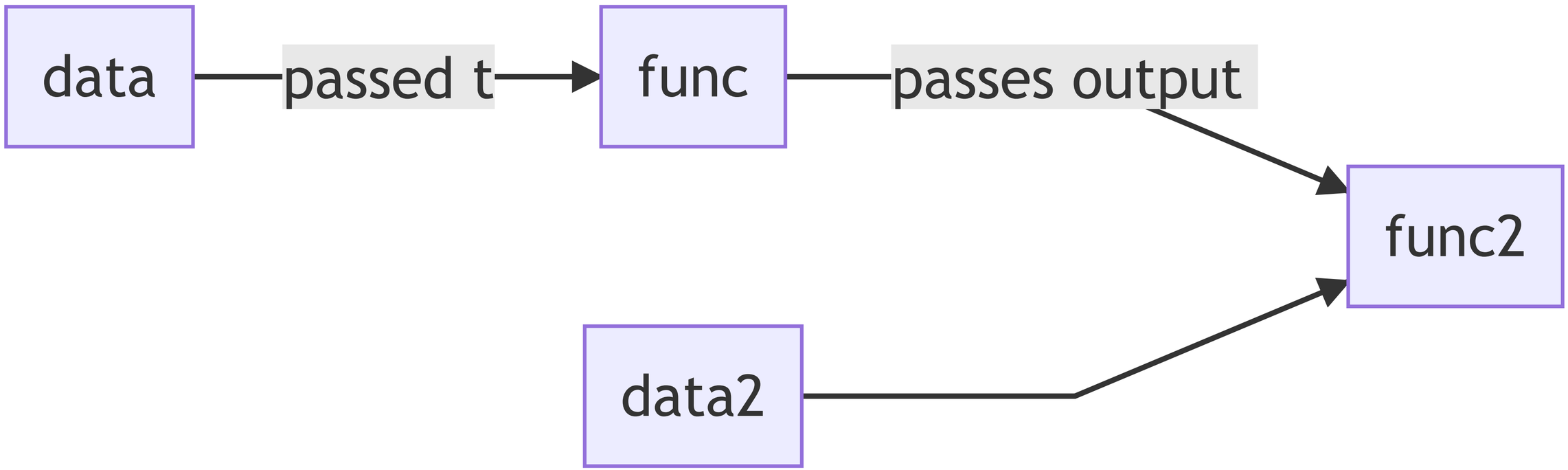
```
def make_adder(val):
    return lambda x: x + val

map(make_adder(5), vec)
```

The Dag

Graphical Programs

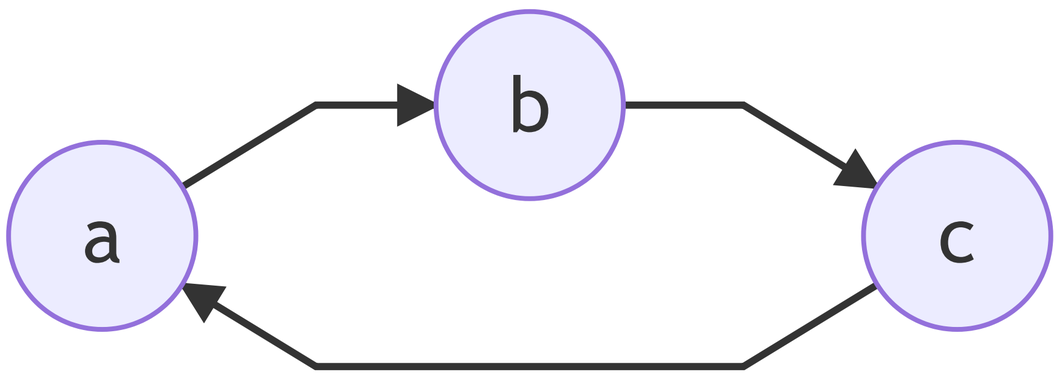
Not visual… programs are graphs!



Graphical Programs

A Directed Graph is a set of nodes and edges which have direction

```
import networkx as nx
g = nx.DiGraph([
    ('a', 'b'),
    ('b', 'c'),
    ('c', 'a')
])])
```

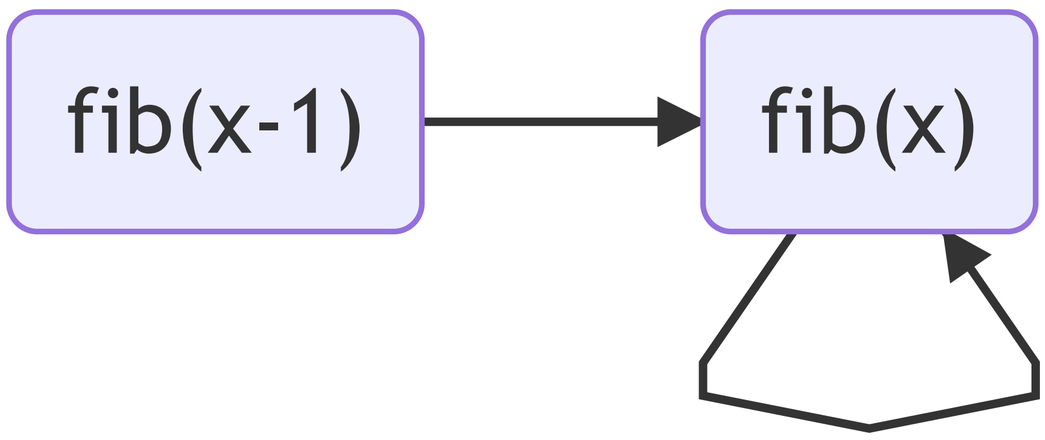


Dags

… a Directed Acyclic Graph has no cycles (which are bad for code!)

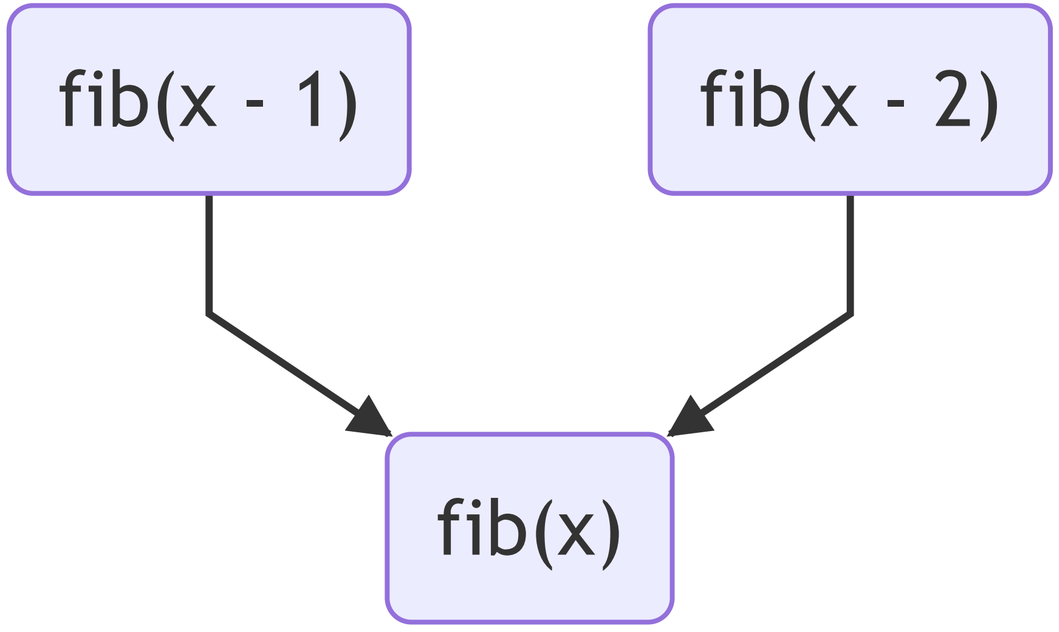
Cyclic

```
def fib(x):
    return fib(x) + fib(x - 1)
```



Acyclic

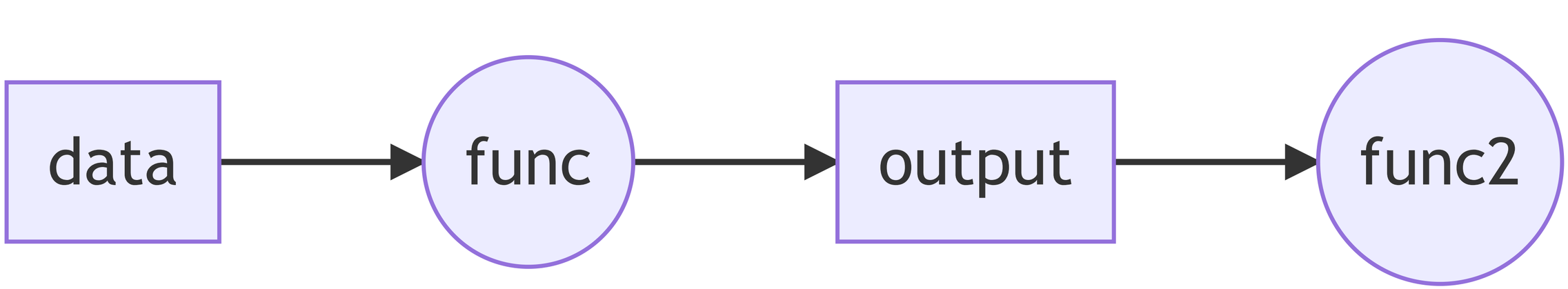
```
def fib(x):
    return fib(x - 1) + fib(x - 2)
```



Graphical Programs

Sometimes we distinguish between functions and outputs

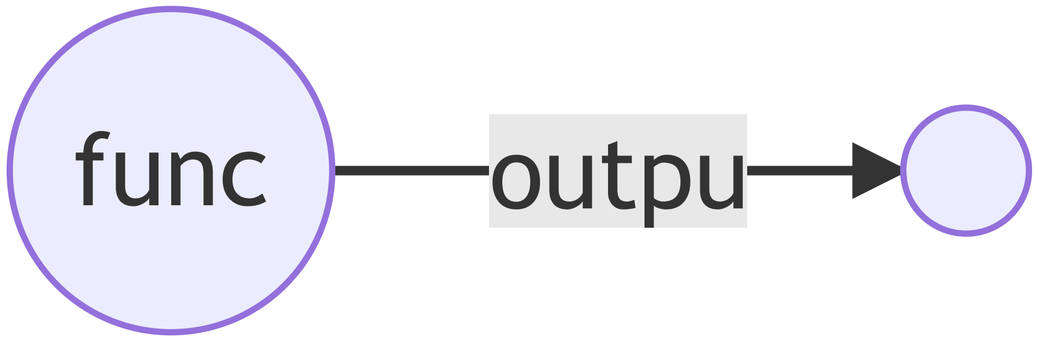
The outputs may be ***delayed*** or ***symbolic***



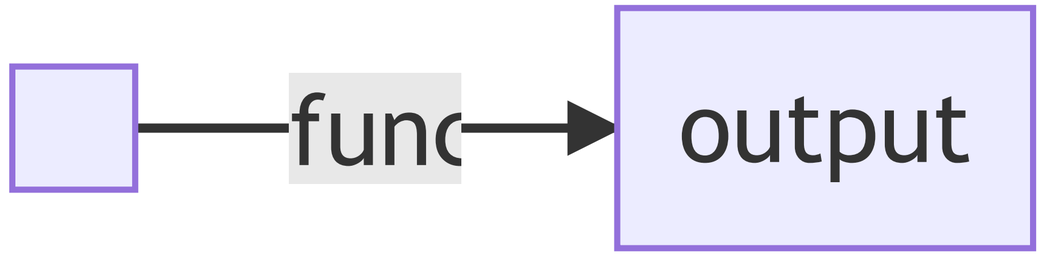
Graphical Programs

… although this is really just playing with edge vs node definitions

Edges are data:

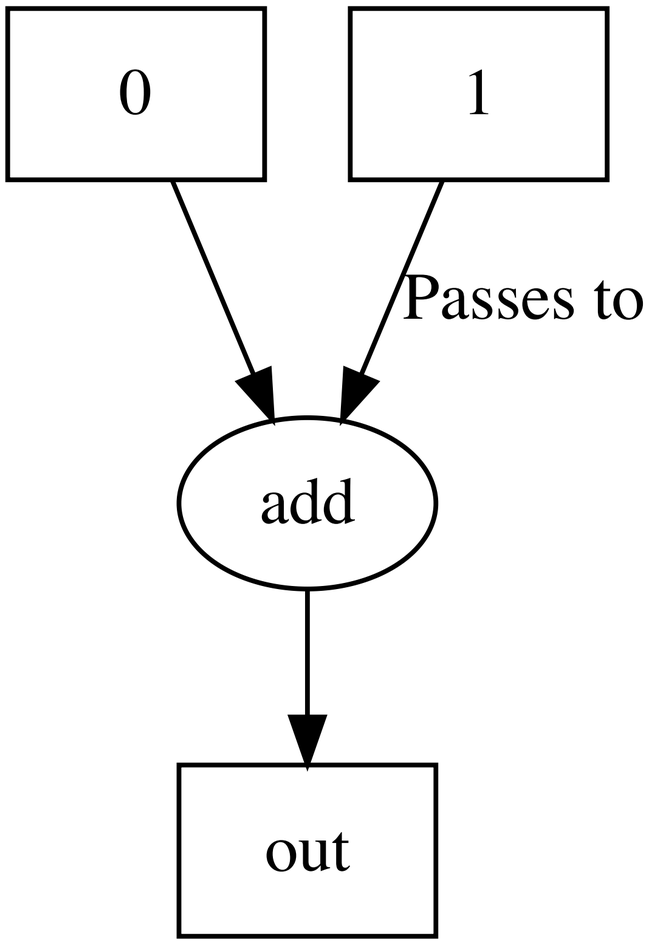


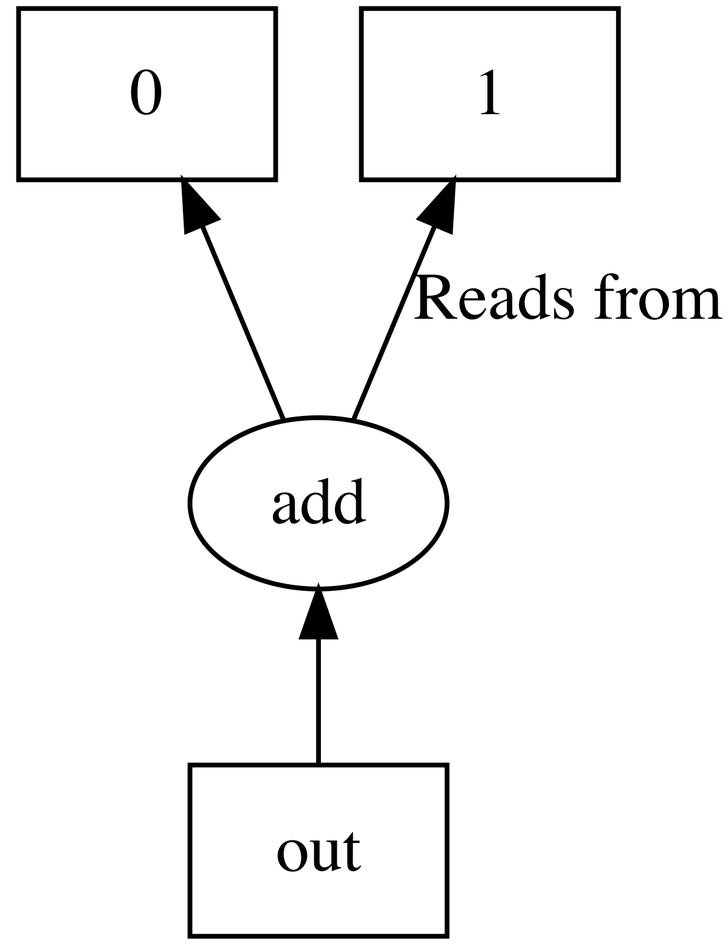
Edges are functions:



Digraph Symmetry

A digraph equals its transpose when you swap the relationship too



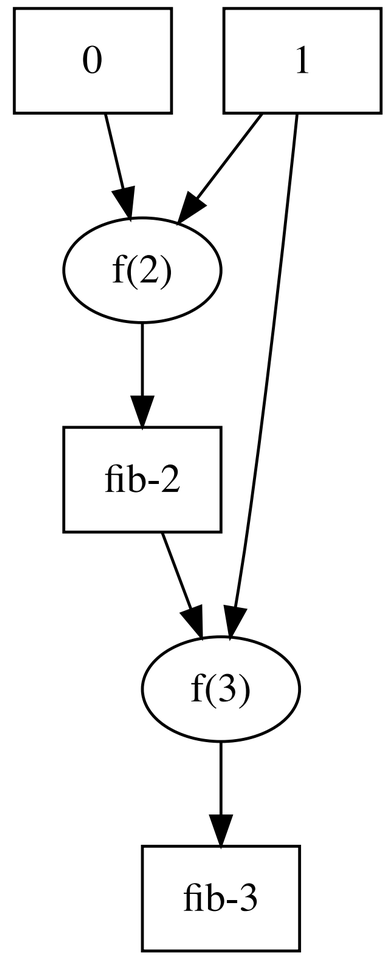


What does Data Science look like?

What we imagine

```
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)

fib(5)
```

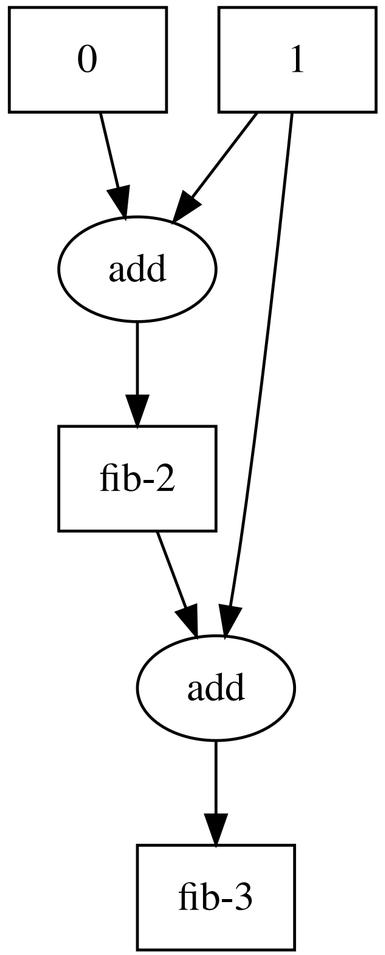


What does Data Science look like?

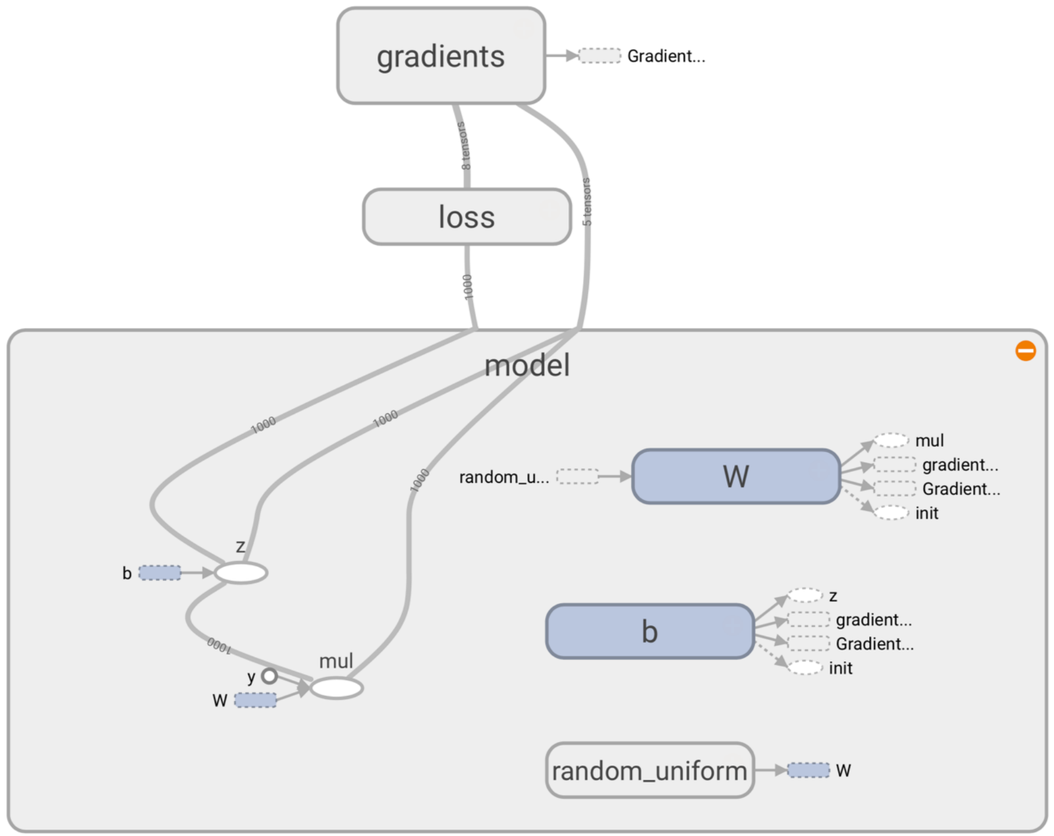
What we imagine

```
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)

fib(5)
```

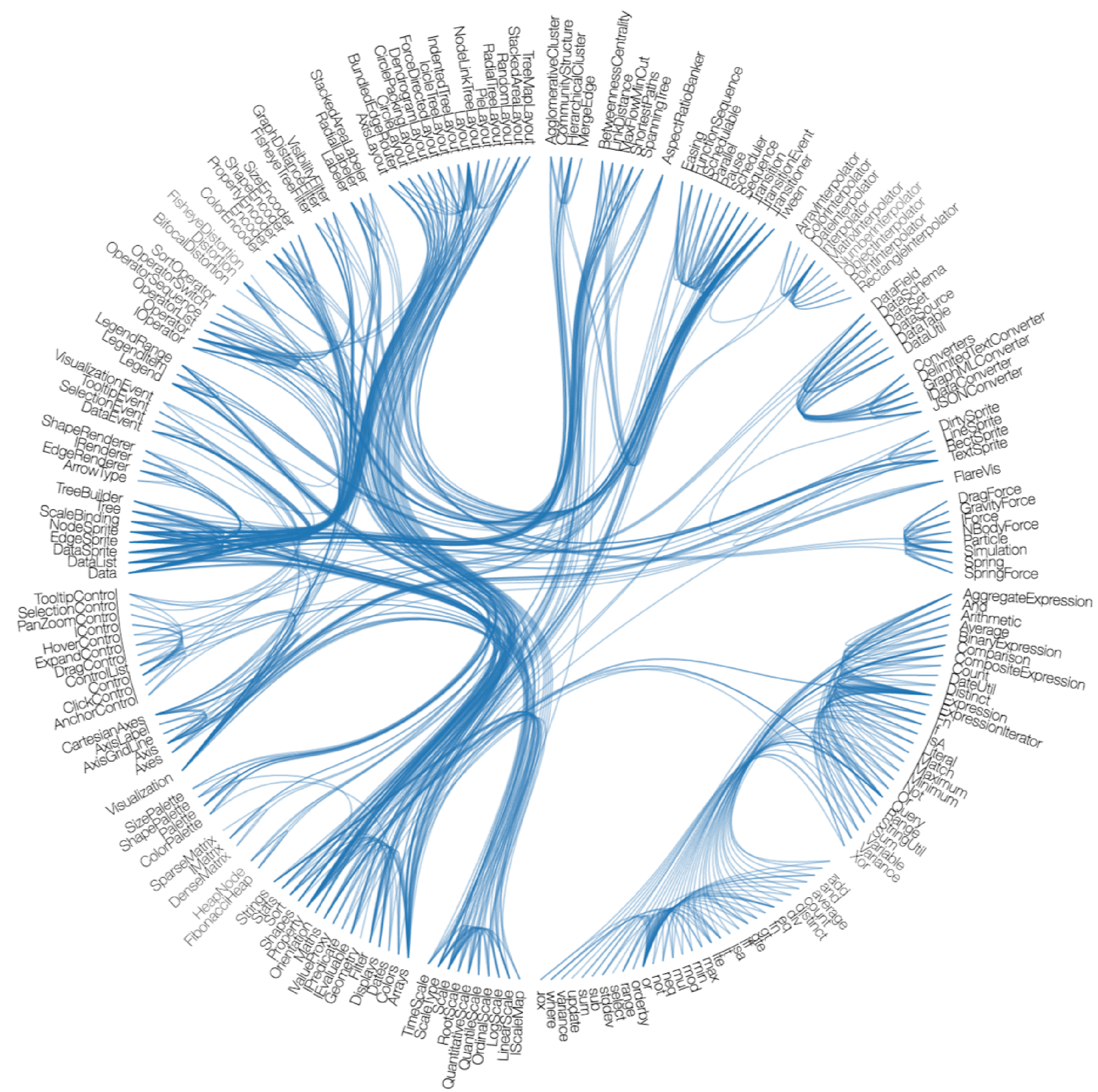


In reality...



Tensorflow

In reality…



Bundle

Dags to the Rescue

Formally representing your code as a graph gives you higher-level tools to manage it.

Dataflow Programming

- Code *is* a graph
- Write your program as (reusable) nodes
- Connect them (argument passing)
- Write higher level graph functions
  - Take graphs as arguments!

Dataflow Programming

Dask

```
from dask import delayed

def fib(n):
    if n >= 2:
        return delayed(add)(fib(n-2), fib(n-1))
    return delayed(n)
```

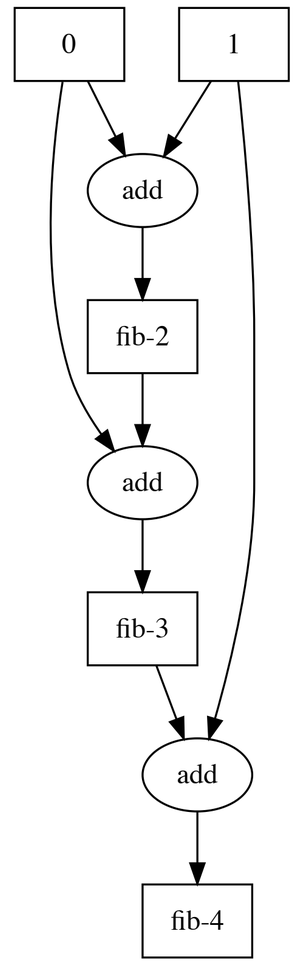
```
{'0': 0,   # 'delayed' values
 '1': 1,
 # (func, *delayed_args)
 # "call add with values in keys 0 and 1"
 'fib-2': (add, '0', '1'),
 'fib-3': (add, '1', 'fib-2')
}
```

Dataflow Programming

- Code *is* a graph
- Visualization and debugging
  - Makes small mistakes obvious!

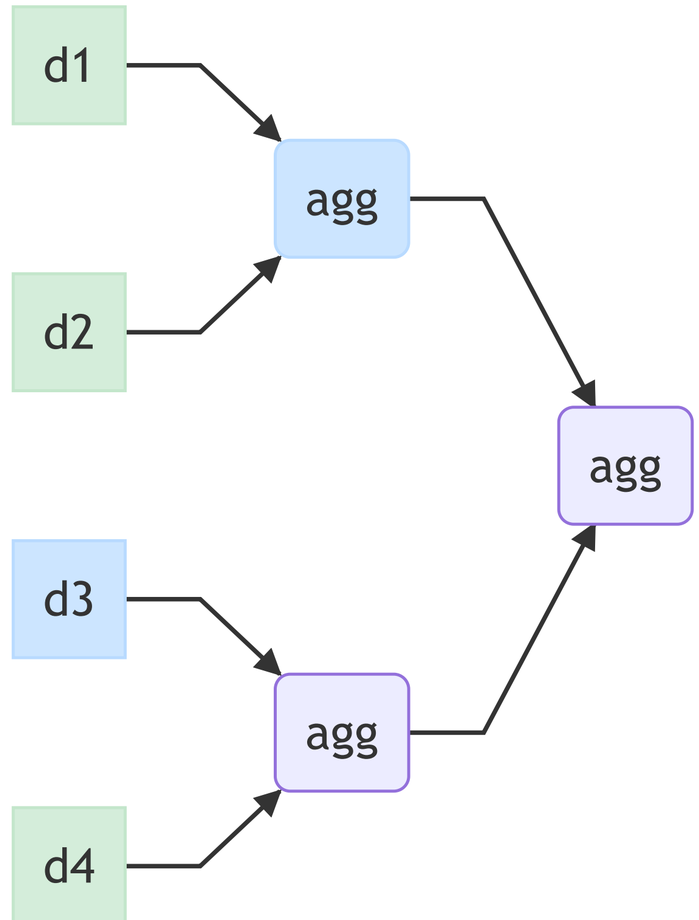
```
from dask import delayed

def fib(n):
    if n >= 2:
        return delayed(add)(fib(n-3), fib(n-1))
    return delayed(n)
```



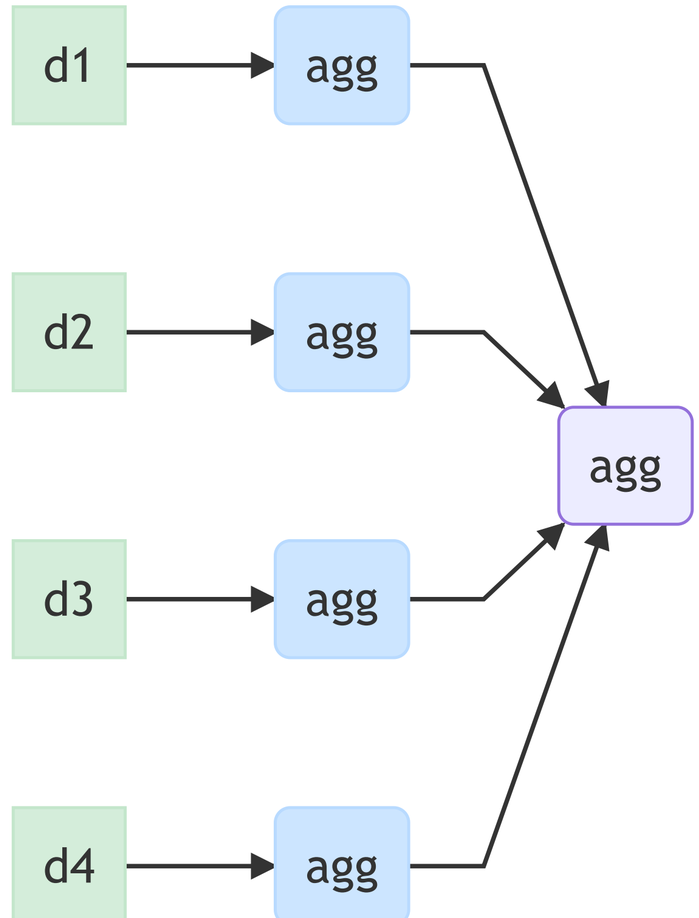
Dataflow Programming

- Code *is* a graph
- Parallel, distributed execution
  - Tasks can run as soon as deps are met
  - Scheduler coordinates work across compute nodes



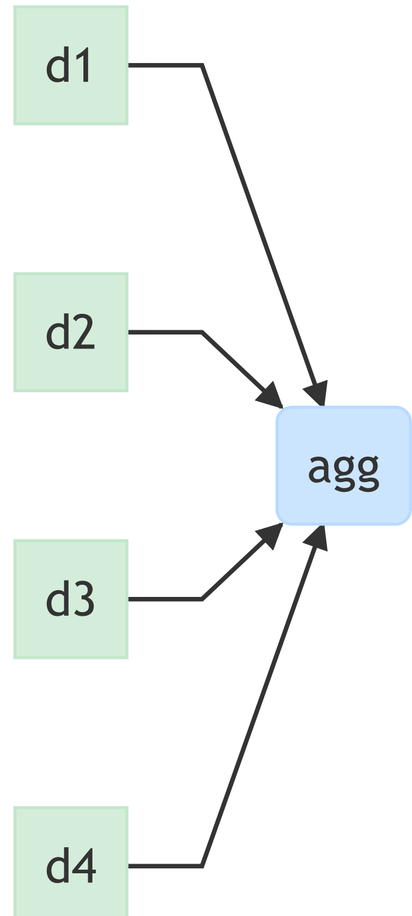
Dataflow Programming

- Code *is* a graph
- Graph optimization
  - Restructure graph at run time
  - Inline functions, combine tasks, cull unneeded…



Dataflow Programming

- Code *is* a graph
- Graph optimization
  - Restructure graph at run time
  - Inline functions, combine tasks, cull unneeded…

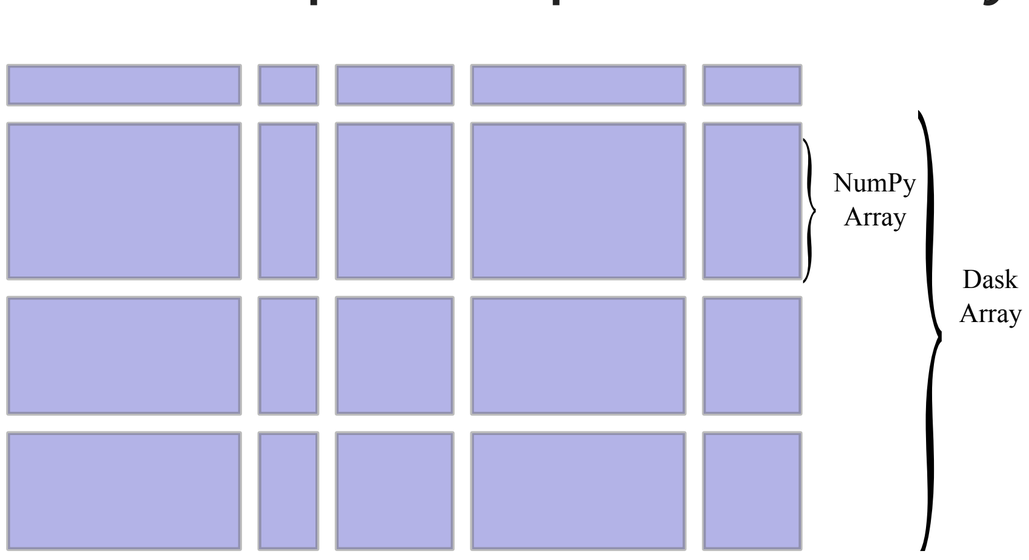


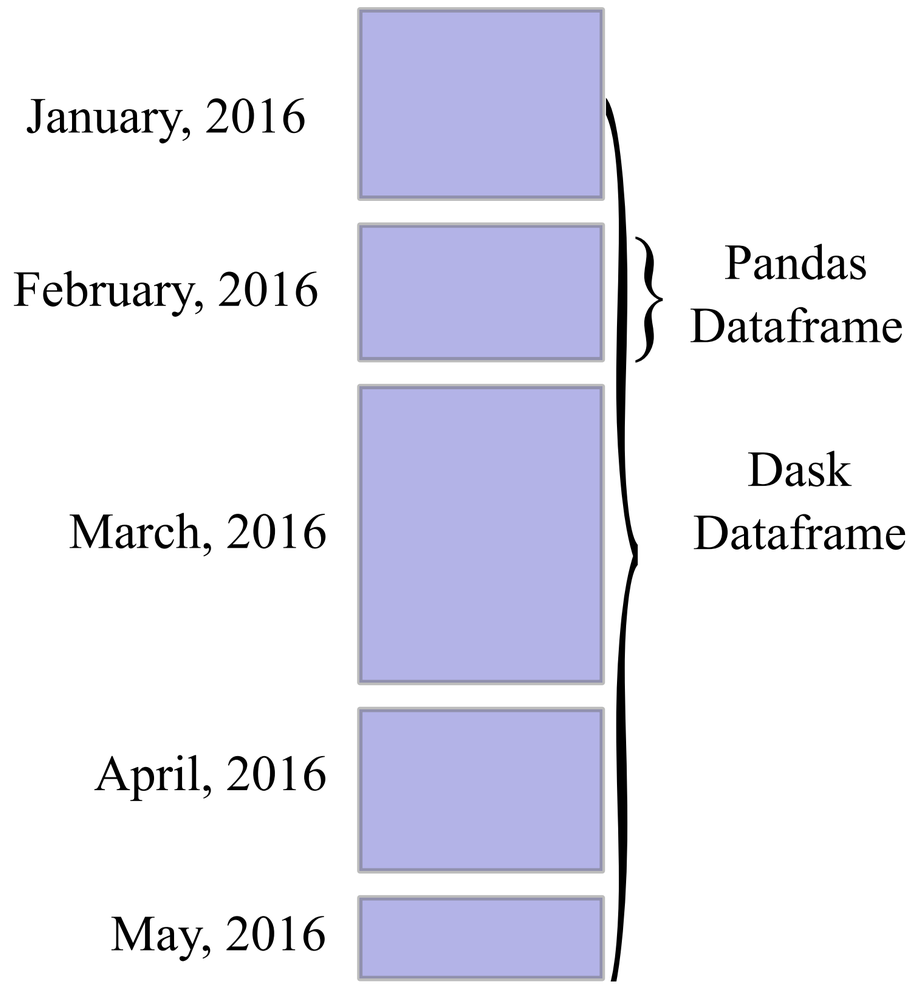
Dask

Dask

Computational Graphs for Python

- Out-Of-Memory numpy, pandas, text
- Chunked, distributed, delayed
- Can replace Spark for many workflows





Dask

```
def delayed_dot(mtx_a, mtx_b):
    """Delayed block-wise dot product"""

    # Outer blocks
    BR = mtx_a.shape[0]
    BC = mtx_b.shape[1]

    INNER_BLOCKS = mtx_a.shape[1]
    # == mtx_b.shape[0]

    # "Outer" matrix:
    # each part sums delayed dots
    # of the inner
    dot = delayed(np.dot)
```

```
    out = np.asarray([
        [
            sum([
                dot(a, b)
                for a, b in zip(
                    mtx_a[block_r, :],
                    mtx_b[:, block_c]
                )
            ]) for block_c in range(BC)
        ] for block_r in range(BR)
    ])
    return out
```

The Revelation

The majority of “Data Science” work you do can be broken down into pieces, operated on independently, and “added” back together

A symbolic workflow captures those larger primitives and helps us stitch together and manage work

It will also let us ***distribute*** and otherwise schedule, manage, and inspect computation later.

Luigi

Project scaffolding

Dask

- Computation graph
- “Low” level
- Computes things

Luigi

- Task graph
- “High” level
- Runs things (which could be dask)

The Task

```python
import luigi

class MyTask(luigi.Task):
    param = luigi.Parameter(default=42)

    def requires(self):
        return SomeOtherTask(self.param)

    def run(self):
        f = self.output().open('w')
        print >>f, "hello, world"
        f.close()

    def output(self):
        return luigi.LocalTarget('/tmp/foo/bar-%s.txt' % self.param)

if __name__ == '__main__':
    luigi.run()
```

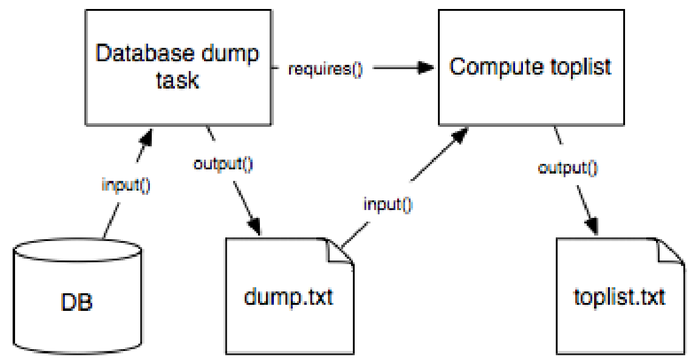
The business logic of the task

Where it writes output

What other tasks it depends on

Parameters for this task

The Task


Database dump task
requires()
Compute toplist
input()
output()
input()
output()
DB
dump.txt
toplist.txt

The Point

Tasks

Represent your data/modelling steps at the high level

Targets

Atomic read/write data stores

`.exists()` for dependency resolution

Scheduler

Runs and coordinates execution on your graph

Skeleton

Provides a way to organize your entire workflow and project

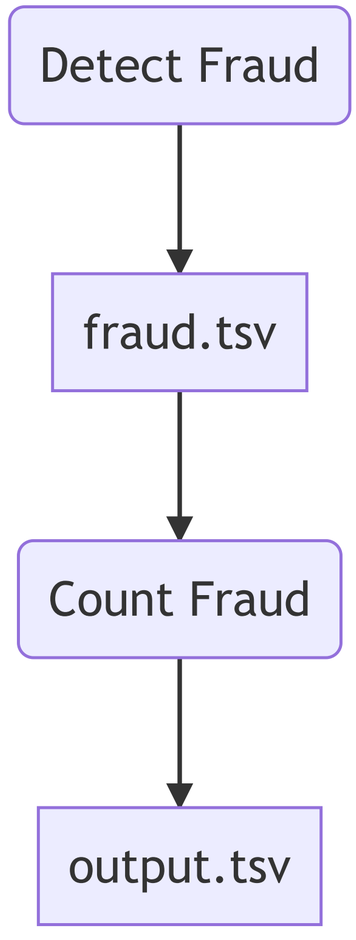
Atomicity

We've seen this:

```
$ cat logs | detect-fraud > fraud.tsv
```

```
select count(*) from fraud where p_fraud > 0.95
```

Failure in `detect-fraud` leaves partial file for reading, resulting in abnormally low fraud counts!



Atomicity

The Atomic Write:

1. Write to tempfile
2. Rename or move on success (atomic)

Atomicity

Ideally

> *an*
> *indivisible and irreducible series of operations such that either all occur, or nothing occurs*
>
> — *Wikipedia*

Eg no partial data files left over

Otherwise

- No time when a task might appear complete before it actually is
- Minimize time between task success and ‘knowing’ it succeeded

Atomicity

But not everything's a file or local…

SQL

```
psql=# BEGIN;
psql=# INSERT INTO table VALUES (1, ‘one’);
psql=# COMMIT;
```

Only works if you know the key or some query that returns non-empty.

Fails for multi-row inserts where you insert 0 rows

Hadoop/S3/Immovables

```
big-slow-calculation > final_output/part_1.tsv
touch final_output/_SUCCESS
```

Atomicity

Key Insights from Luigi:

- Define `.write()`, `.exists()` atomically for a target
  - May never return `True` for an incomplete or failed output
- A task is runnable when all input targets `.exists()`
- A task is done when it's output target `.exists()`

If above are satisfied, our data will always be consistent

No need for central scheduler/DB!

(Really useful for small teams w/ on-demand compute!)

Pset 2

Objectives

1. Continue maintaining our library
2. Use/install/test that library
3. Improve that library in a backwards-compatible way
    - While learning about other packages
4. Practice `class` overrides, working with other code
5. Improve your cookiecutter
6. Vectorized/functional coding
7. *Word embeddings are the gravy, not the steak*

Private Installs

Normally you either

1. pip install from pypi
2. pip install from your VCS

CI systems usually hook into VCS creds more or less automatically

Turns out Travis only authenticates 1 repo at a time

We need to inject your authentication into the Docker image, both on your local machine and in Travis

Up next

Future objectives:

1. Taskify
2. Data storage mechanisms
3. Avoiding data in git - how to download
4. Visualizations and dimensionality reduction

Readings

- Functional Programming HOWTO
- So you want to be a functional programmer
- Optimizing Pandas for Speed