Classes, Composition, and Graphs

Dr. Scott Gorlin

Harvard University

Fall 2018

Agenda

- Pset 2
- The Midterm!
- Docker
- Inheritance
- Composition
- Luigi
- States
- Readings

Pset 2

Piazza Hall of Records

Top (Cumulative) Contributors:

- 41 +0 Aaron White
- 30 +2 Amyrah Arroyo
- 26 +0 Michael Blanchard
- 22 +1 Manish Sinha
- 19 NEW Sarah Reiff

The Point

By now you should be comfortable:

1. With python packaging, and managing your own library
2. Juggling multiple similar, but distinct, development environments via Pipenv
3. Writing unittests, running them locally, and watching the build system replicate that
4. Manipulating the build system for more than just unittests

The Midterm!

Deets

- In class
- Open notes (no Google)
- Online (does everyone have laptop/internet?)
- Short answers involving pseudocode and/or MC
  - We will asses concepts, not perfect runnability
- Probably will not involve running actual code

Deets

- Sections that week are for office hours only (no new material). Please let TA's know if you plan to come Tues but don't normally do so.
- There will either be a take-home component or concurrent pset to be due the Monday after, to be assigned by next week

Practice

Rewrite this function as an iterator, without the `for ... in range`, working for any iterator:

```
def cumsum(iterable):
    s = 0
    out = []
    for i in range(len(some_list)):
        s += some_list[i]
        out.append(s)
    return out
```

```
def cumsum(iterable):
    s = 0
    for v in iterable:
        s += v
        yield s
```

Practice

Which is most likely the best use case for the `mock` module?

1. To monkey patch additional data preprocessing into the `.train()` routine of some machine learning library like `sklearn`
2. To monkey patch a stand-in replacement for an external service, like S3, for testing
3. To ensure a private class method is called with the correct args when you use a public method, like `.train()` or `.save()`, during testing

2; 3 is also a good use case but tends to be abused, and can lead to false positive test failures when you change the public method.

Practice

Write a script to find any integers a, b, and c that satisfy the following:

$$a^n + b^n = c^n$$

```
def solve_fermat(n):
    """Returns (a, b, c) that satisfy Fermat's Last Theorem"""
    ...
```

for any $n > 2$

Just kidding!

Docker

Pristine, Repeatable Builds

Docker images are designed to be 100% frozen and repeatable. This means, once built, they ***never change***.

Expected (local)

```
$ echo "world" > ~/hello.txt
$ cat ~/hello.txt
world
```

Actual (docker)

```
$ ./drun_app bash -c "echo world > ~/hello.txt"
$ ./drun_app bash -c "cat ~/hello.txt"
cat: /root/hello.txt: No such file or directory
```

… you lose all changes when the container exits after the first line!

Pristine, Repeatable Builds

This will not work:

```
./drun_app pipenv install blah
# Need to docker-compose build to 'capture' changes
./drun_app python -c "import blah"
```

Build vs Mounting

When building, docker adds things into the image filesystem

When running, you can ***mount*** the local disk on top of the image filesystem

You can then write back to the local filesystem, eg pipenv install ***updates the Pipfile*** (but other changes are discarded)

```
version: '3'
services:
  app:
    build:.
    volumes:
      # The current local directory, '.',
      # mounted to /app inside the image at run
      - .:/app
```

Update local Pipfile:

```
./drun_app pipenv install blah
```

… and then build a new docker image with the latest version!

The TL;DR;

Docker is normally meant for deploying completely static images.

We're using it a bit differently, as an execution environment on top of our local system.

Our use case only works with mounted directories to persist the data and changes.

Build vs Runtime

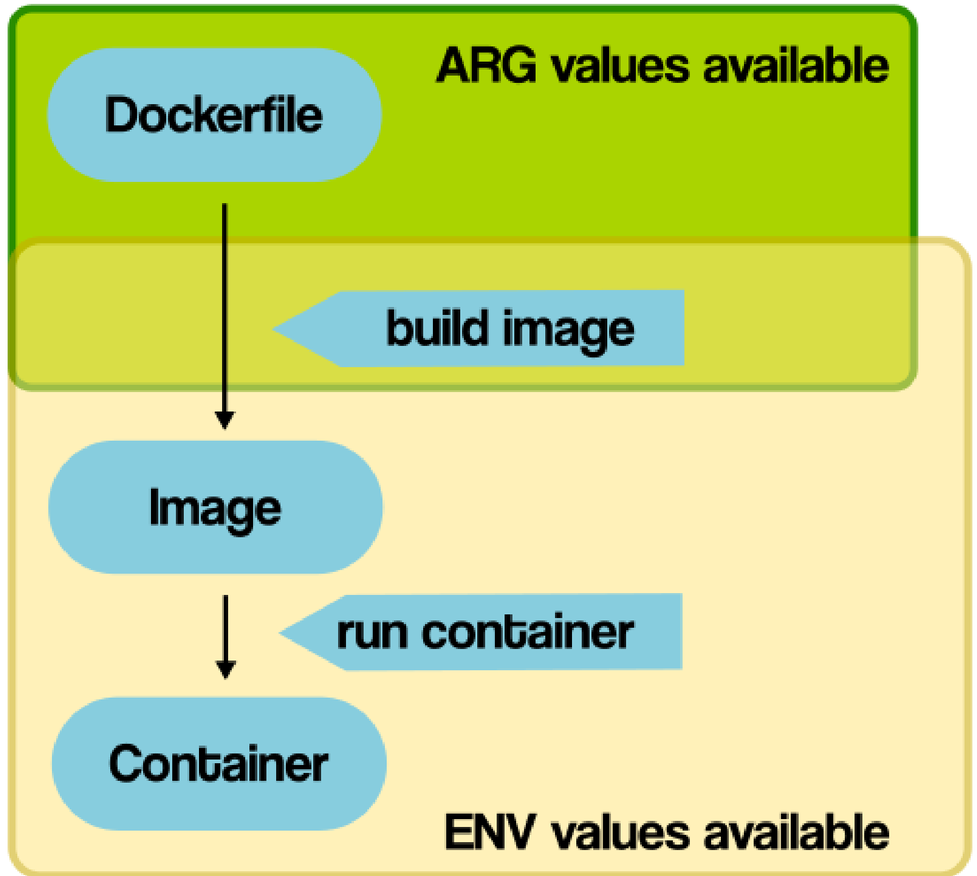
Build

- *Build* arguments
- Static, deterministic

Runtime

- *Env* vars
- Multiple environments, variables, mounts, ports, etc
- Can even run more than one
- Discards all changes (internally) on exit

Build vs Runtime



Docker ARG, ENV and .env - a Complete Guide

Optional...

Docker gives us a LCD and ensures a 100% consistent environment for all students, instructors, and remote build systems. This is the purpose of a container.

You can try using pipenv directy on your machine, without Docker (Windows: YMMV)

You still need to get docker running, and instructions will be tailored for that. You will be assessed by docker runs on Travis.

Small Details

REST

REpresentational State Transfer

architectural style that defines a set of constraints to be used for creating web services.

… eg to communicate between docker client and server

ReST

ReStructured Text

an easy-to-read, what-you-see-is-what-you-get plaintext markup syntax and parser system.

… eg to write your python docs

Inheritance

Class Inheritance (Refresher)

Given a base class…

```
class Regressor:


    def __init__(self, penalty):
        self.penalty = penalty

    def predict(self, x):
        return np.dot(x, self.beta)
```

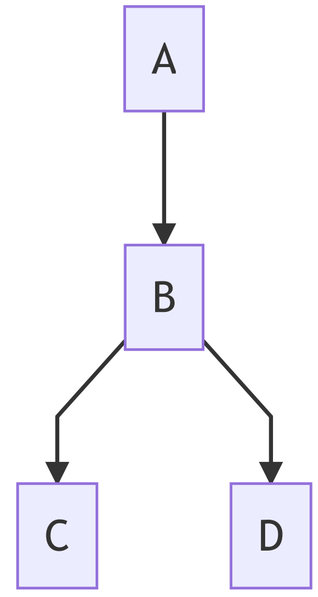
One can inherit and override

```
class Classifier(Regressor):


    def predict(self, x):
        # Go up the chain
        return super(
            Classifier,
            self).predict(x) >= 0
```

Class Inheritance (Refresher)

```
class A(object):
    _X = 5
    def x(self):
        return self._X
```

```
class B(A):
    _X = 7

class C(B):
    def x(self):
        return super(C, self).x() + 1

class D(B):
    def x(self):
        return 2 * super(D, self).x()
```



Note the use of `super(cls, instance)` to call up the inheritance tree

Abstracts and Interfaces

A common pattern is to define an ‘abstract’ base class

This codifies the methods and interface, while allowing you to implement other methods and defaults

```
class BaseRegressor:
    def regress(self):
        raise NotImplementedError('Abstract')
```

C.f. `abc.AbstractBaseClass` which fails on instantiation if a method is not implemented.

I find `abc` to be overly formal. You may not need every method, and it will already safely fail.

Mixins

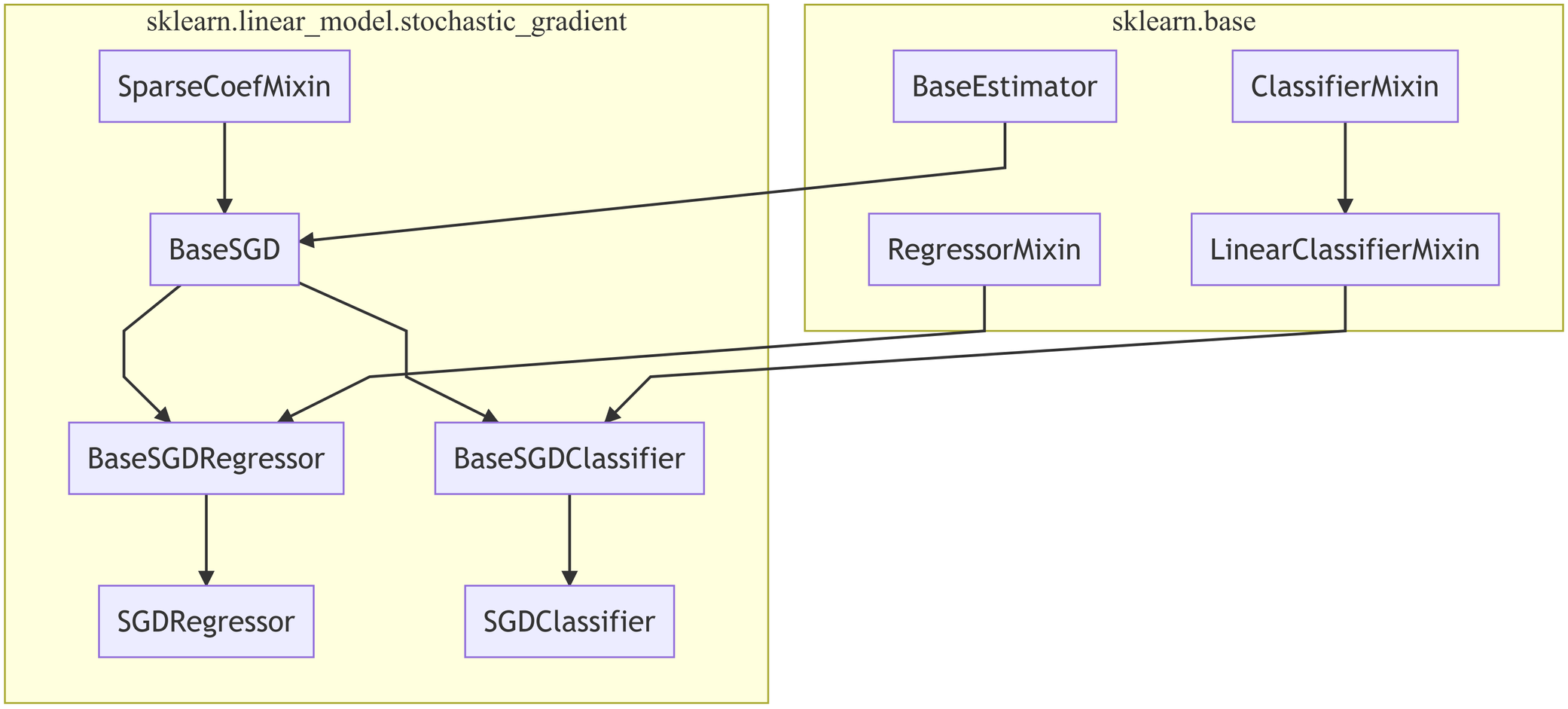
Capture orthogonal properties and methods, mix them in!

```
class SparseCoefMixin(object):
    """Mixin for converting coef_ to/from sparse
    L1-regularizing estimators should inherit this.
    """

    def densify(self):
        """Convert coefs to dense format."""
        if sp.issparse(self.coef_):
            self.coef_ = self.coef_.toarray()

    def sparsify(self):
        """Convert coefs to sparse format."""
        self.coef_ = sp.csr_matrix(self.coef_)
```

```
class BaseSGD(BaseEstimator, SparseCoefMixin):
    ...
```

. . . AND THEN WE JUMP THE SHARK

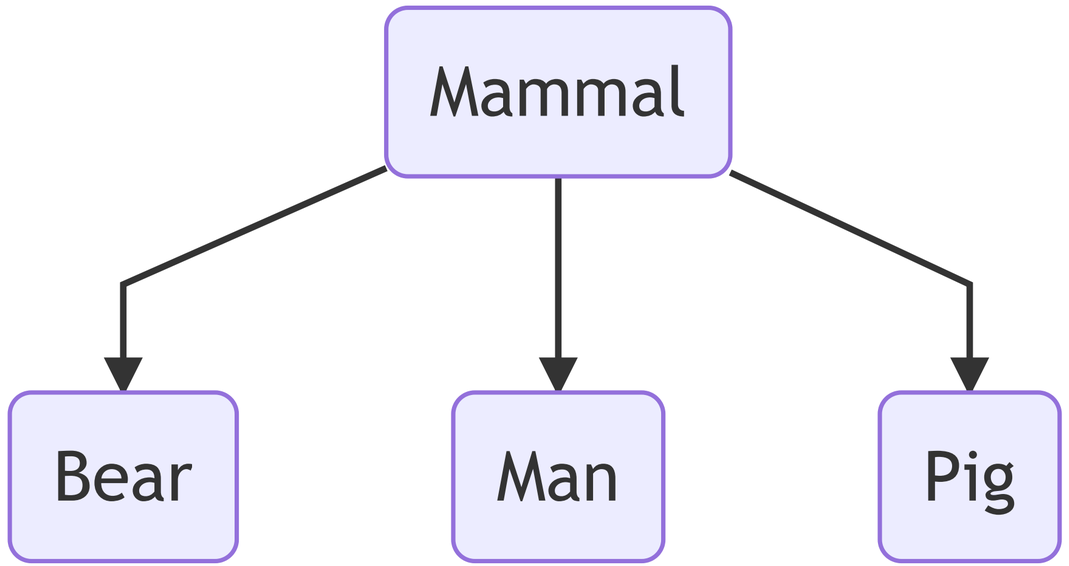
Multi inheritance is bad, m’kay?



... AND THEN WE JUMP THE SHARK

Avoid!

```
class Mammal:
    pass

class Bear(Mammal):
    def speak(self):
        print('growl!')

class Man(Mammal):
    def speak(self):
        print('Hello!')

class Pig(Mammal):
    def speak(self):
        print('Oink!')
```

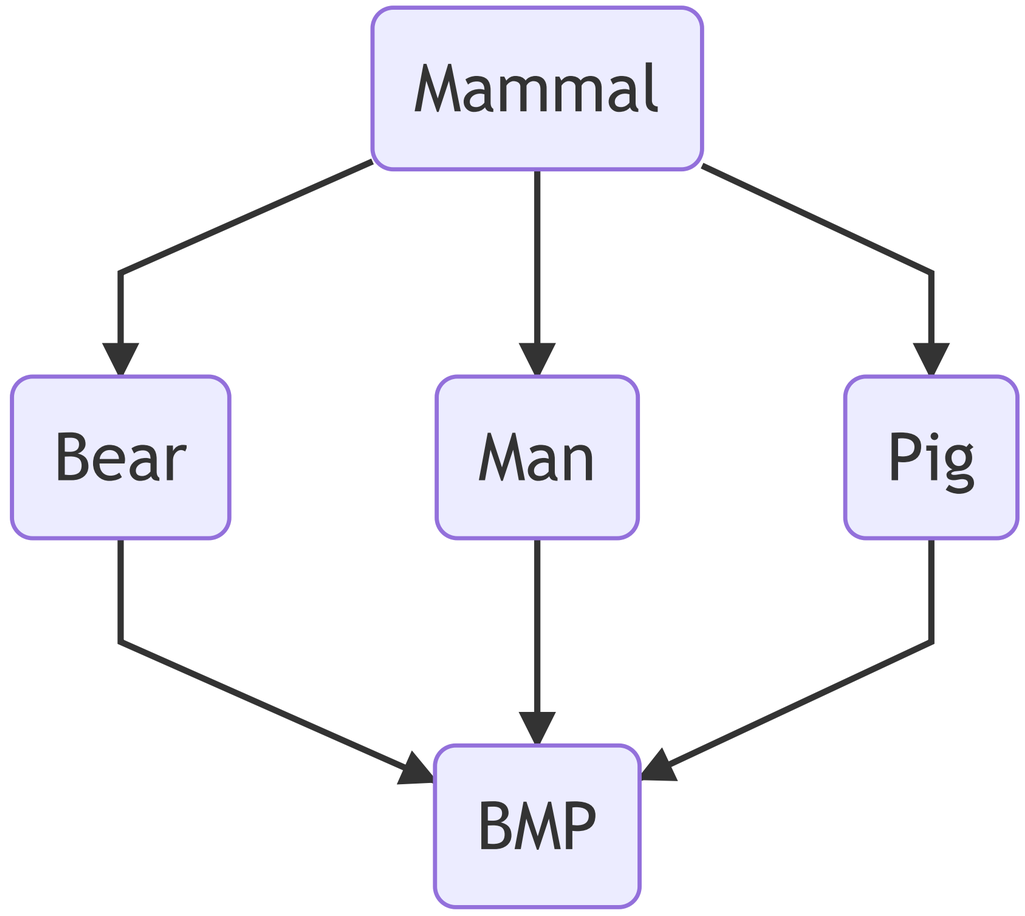


... AND THEN WE JUMP THE SHARK

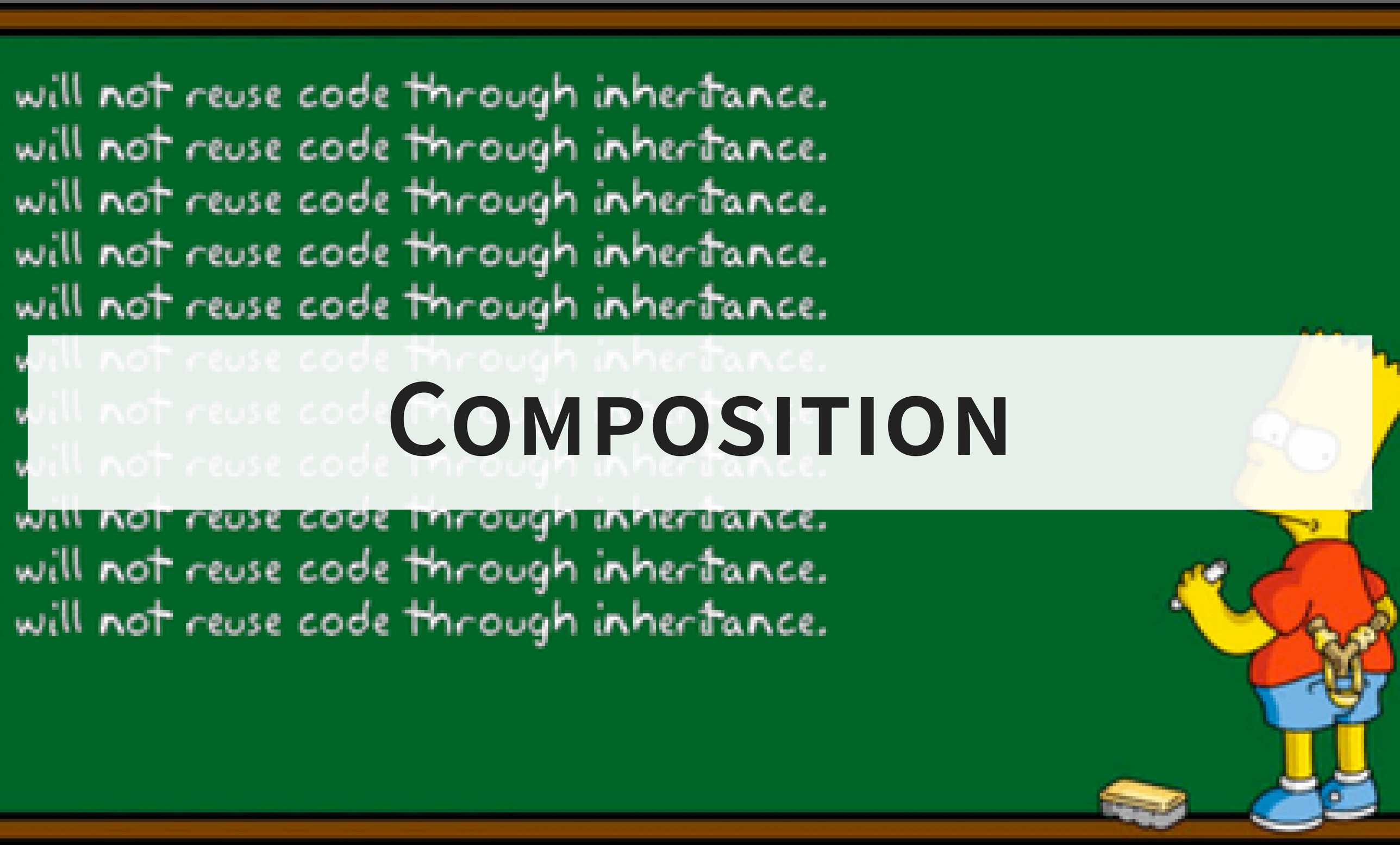
Avoid!

```
class BMP(Bear, Man, Pig):
    def speak(self):
        # super would fail!
        # super(BMP, self).speak()

        # Note unbounded function syntax
        Bear.speak(self)
        Man.speak(self)
        Pig.speak(self)
```

You can manually resolve multi-inheritance if necessary, as above.



Composition



Composition

… an orthogonal organization

Inheritance

Mary ***is a*** Data Scientist

```
mary.compute()
```

Composition

Mary ***has a*** MacBookPro

```
mary.computer.compute()
```

Give classes their sophistication by giving them rich properties, and delegating responsibilities to them

Composition

As instance properties:

```
class Student:
    def __init__(self, computer=None):
        self.computer = computer or Laptop()
    def compute(self, *args):
        return self.computer.compute(*args)
```

As *declarative classes*:

```
class Student:
    COMPUTER_CLASS = Laptop
    def __init__(self):
        self.computer = self.COMPUTER_CLASS()

class MacStudent(Student):
    COMPUTER_CLASS = MacBookPro
```

Composition

Composition allows for more tightly scoped classes and helps prevents nasty inheritance patterns

It is usually easier to test as well

Descriptors

Encapsulation:

```
class A:
    def __init__(self, x):
        self._x = x

    def get_x(self):
        # Post-process x
        return self._x

    def set_x(self, val):
        # Pre-process x
        self._x = val
```

You can hide it:

```
class A:

    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, val):
        self._x = val

>>> A(5).x   # not .x()
```

Provides convenience, backwards compatibility, etc, at small cost of clarity

Descriptors

property...

```
class A:
    @property
    def x(self):
        # access as a.x, not a.x()
        return self._x

>>> A.x
<property at 0x10dd1f9a8>

>>> A(5).x
5
```

... is shorthand for

```
class X:
    def __get__(self, obj, objtype):
        if obj is None:
            return self
        return obj._x

class A:
    x = X()

>>> A.x
<__main__.X at 0x10dcccb38>

>>> A(5).x
5
```

Descriptors

```
class RevealAccess(object):
    """Logs during access"""

    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print('Retrieving', self.name)
        return self.val

    def __set__(self, obj, val):
        print('Updating', self.name)
        self.val = val
```

```
class MyClass(object):
    x = RevealAccess(10, 'var "x"')

>>> m = MyClass()
>>> m.x
Retrieving var "x"
10
>>> m.x = 20
Updating var "x"
>>> m.x
Retrieving var "x"
20
```

Descriptors

```
class _LocIndexer(_LocationIndexer):
    """Access a group of rows and columns"""

    def __init__(self, frame=None):
        self.frame = frame

    def __get__(self, frame, frame_class):
        """Bind a dataframe via frame.loc"""
        return self.__class__(frame)

    def __getitem__(self, key):
        """Return slice of bound frame via .loc[key]"""
        ... lots of logic...
        return self.frame._take(key, axis=axis)
```

```
class DataFrame(NDFrame):
    loc = _LocIndexer()

>>> DataFrame.loc
<property object at 0x112863098>
>>> df.loc
<pandas.core.indexing._LocIndexer object
at 0x1134deb38>
```

… it's not implemented exactly like this

Declarative Classes

Capture logic through composition, declare values and config on class

Minimize unique functions and complex inheritance

```
class SomeClass:
    a = A()
    b = B()

class Variant(SomeClass):
    b = B(val=2)
```

Declarative Classes

Lots of players…

Param

```
class A(Parameterized):
    a = Number(0.5)
    b = Boolean(False,doc="Enable feature...")
```

Django

```
class Choice(Model):
    question = ForeignKey(Question)
    choice_text = CharField(max_length=200)
    votes = IntegerField(default=0)
```

Attr

```
@attr.s
class Coordinates(object):
    x = attr.ib()
    y = attr.ib()
```

Luigi

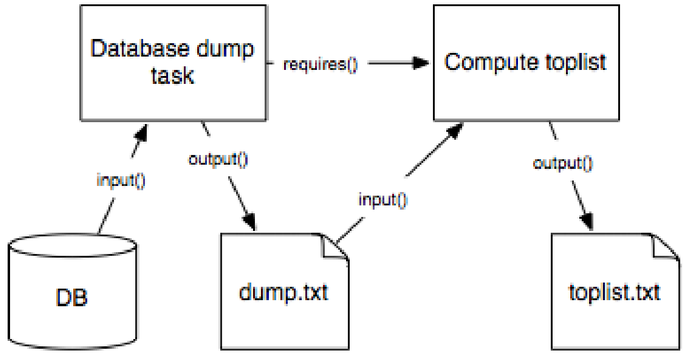
```
class AggregateArtists(Task):
    date_interval = DateIntervalParameter()
```

Luigi

Task Graphs

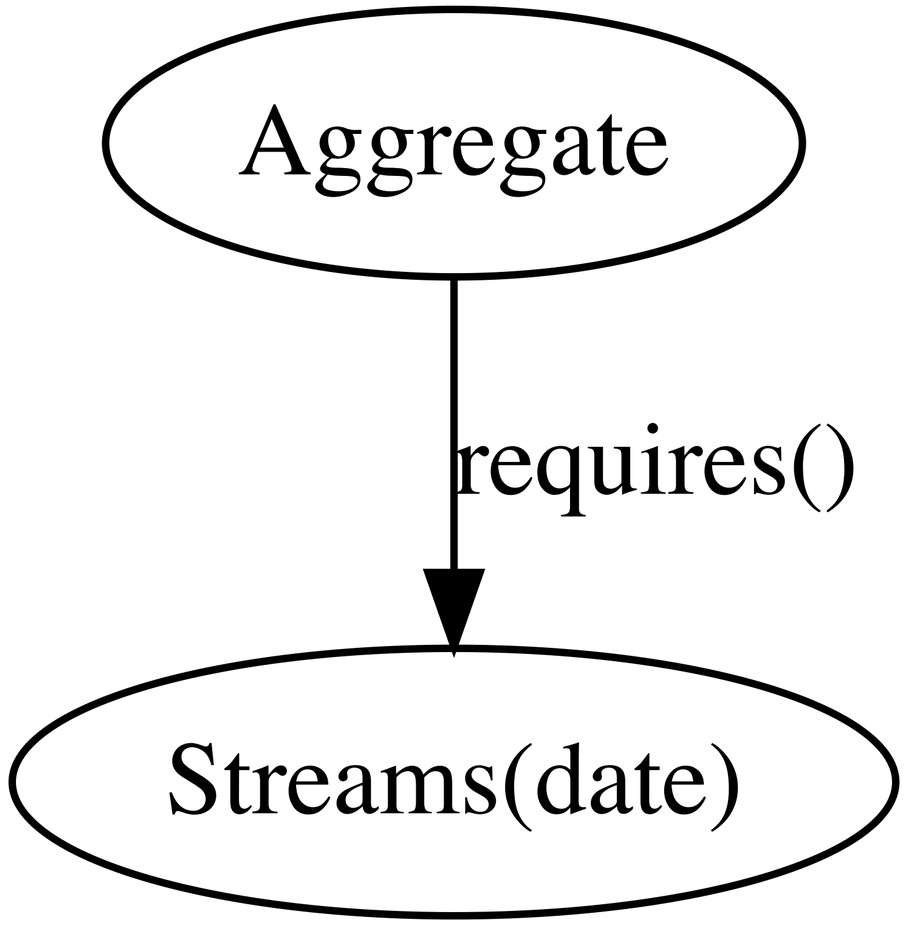
Tasks `.requires()` one another, forming a DAG

(targets are not part of the graph!)



Task Graphs

(arrows indicate direction of function calls)



Parameters

Specification

Pass parameters with `task.clone(...)`

Composition

- CLI parsing
- type support/validation
- batch logic

Declarative

… so they have that going for them.

Atomic Targets

`target.exists()`

Returns `True` iff target was successfully written

If written with an atomic write, can just check for file existance

`target.open()`

An atomic writer, for clean states and to satisfy `.exist()`

Atomic Targets

A task is done iff `task.output().exists()`

A task is ready to run iff

```
all([
    req.exists()
    for req in flatten(task.requires())
])
```

Special Task Types

External

```
class RawData(ExternalTask):
    # Wraps existing data
    # Not managed by Luigi

    # No .run() function!

    def output(self):
        ...
```

These are (usually) your real inputs

Wrappers

```
class AllReports(WrapperTask):
    # Organize many other tasks

    # No .run() or .output()
    def requires(self):
        yield self.clone(Task1)
        yield self.clone(Task2)
```

These are good surrogate outputs

Task Anatomy

```
class AggregateArtists(Task):
    # Declarative Params, composition
    date_interval = DateIntervalParameter()

    def output(self):
        return LocalTarget(
            "data/artist_streams_{}.tsv".format(
                self.date_interval))

    def requires(self):
        return [
            Streams(date)
            for date in self.date_interval
        ]
```

```
def run(self):
    artist_count = defaultdict(int)

    for input in self.input():
        with input.open('r') as in_file:
            for line in in_file:
                ts, artist, tr = line.split()
                artist_count[artist] += 1

    with self.output().open('w') as out_file:
        for artist, count in artist_count.items():
            out_file.write(artist, count)
```

Graphical Programming

A simple example…

```
class A(Task):
    # When multiple dates are ready,
    # only run the latest!
    date = DateParameter(
        batch_method=max,
    )

    @classmethod
    def bulk_complete(cls, parameter_tuples):
        """Find dates that actually need running"""
        ...
```

States

Two Approaches To State Management

Makefile - like

1. `make data.csv` (runs `prog > data.csv`)
2. Tweak `prog`
3. `make data.csv`: timestamp on prog > timestamp on data.csv, reruns and over writes

Newer timestamps don't necessarily mean 'latest code'

May not work for remote systems

Luigi - like

1. `luigi MyTask`
2. Tweak `MyTask`
3. `luigi MyTask`: target already exists, does not run!

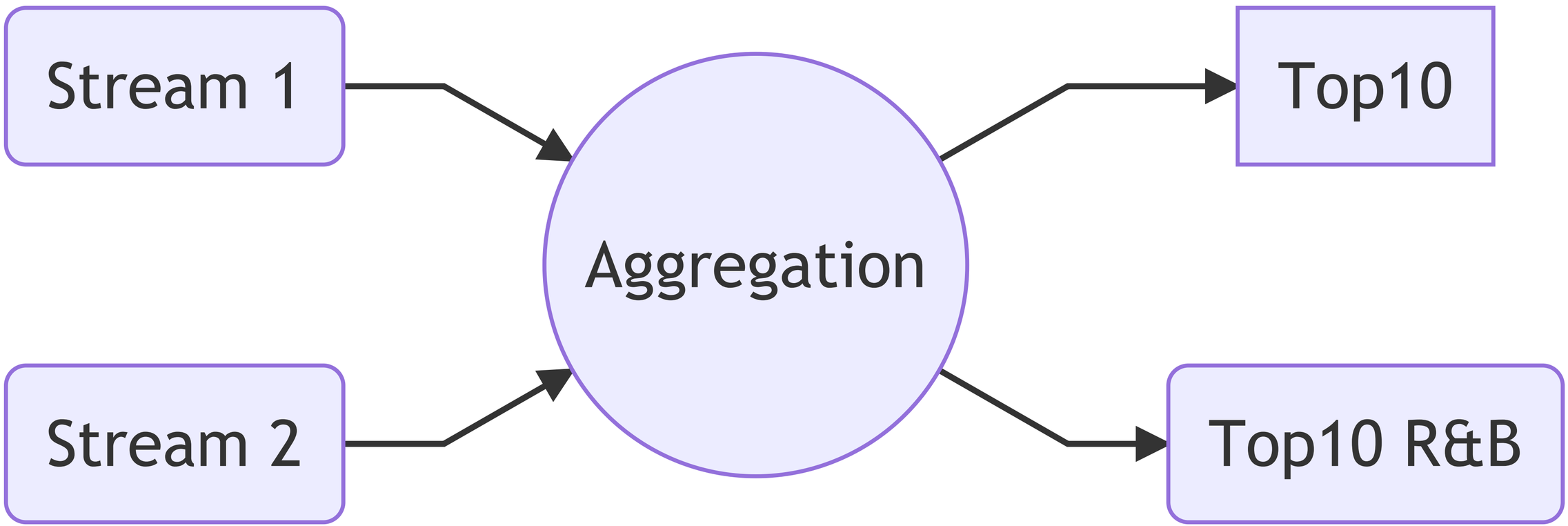
Newer code isn't considered necessary to run unless output is deleted or path is changed

The Sorry State of Stateful Data

Code	Data
Version Controlled	Whatever is there
Functional	Persisted and stateful
Tested	You should be so lucky

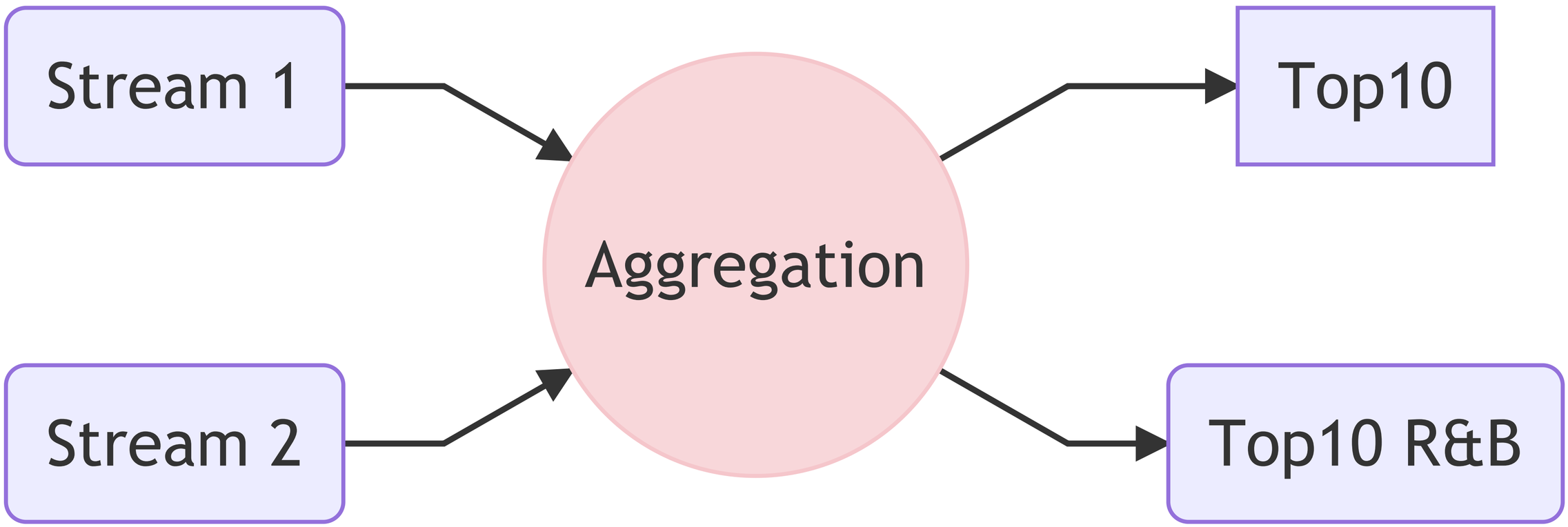
Changes in code are not reflected in stateful data!

Data Dependency Hell



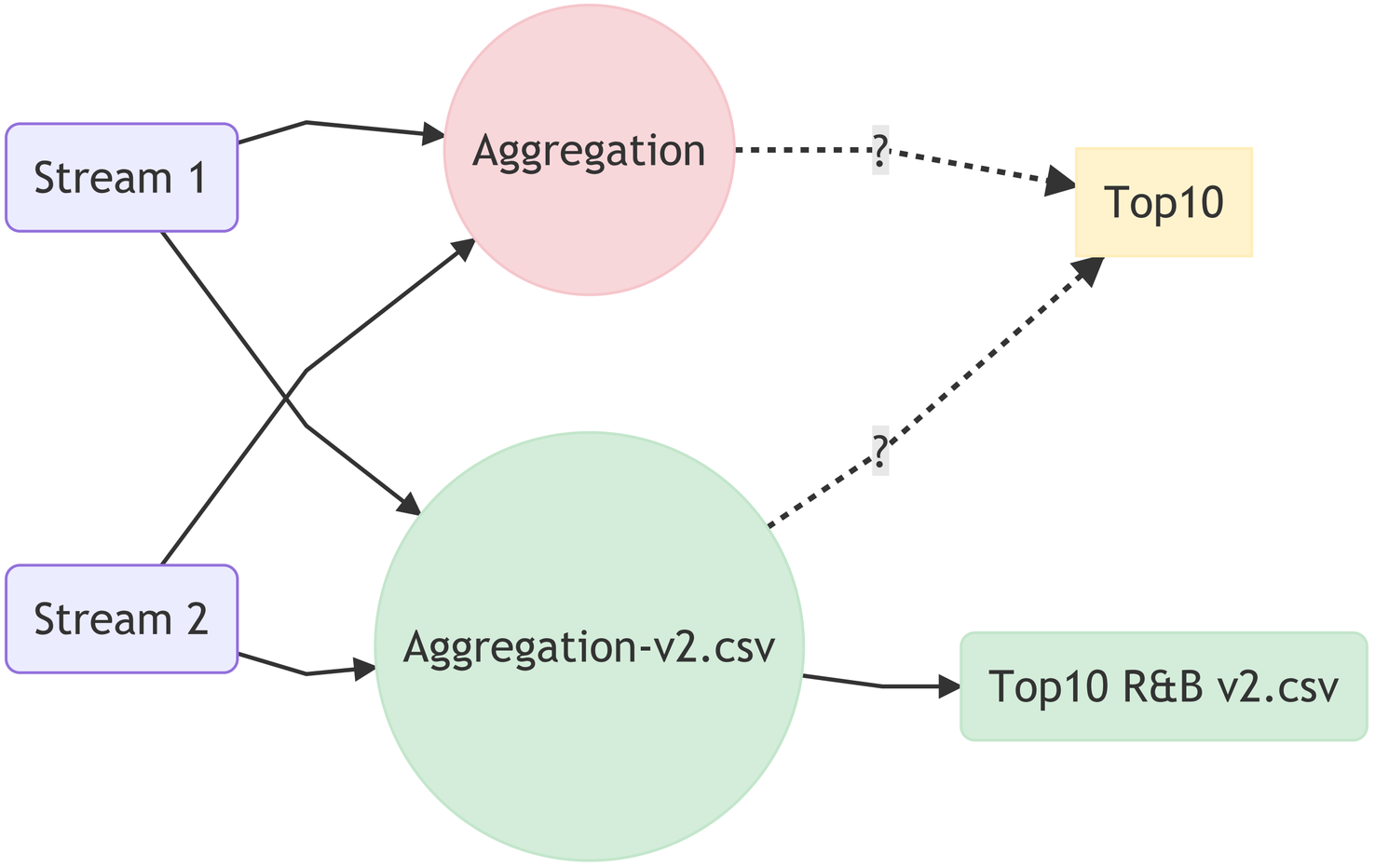
A typical workflow…

Data Dependency Hell



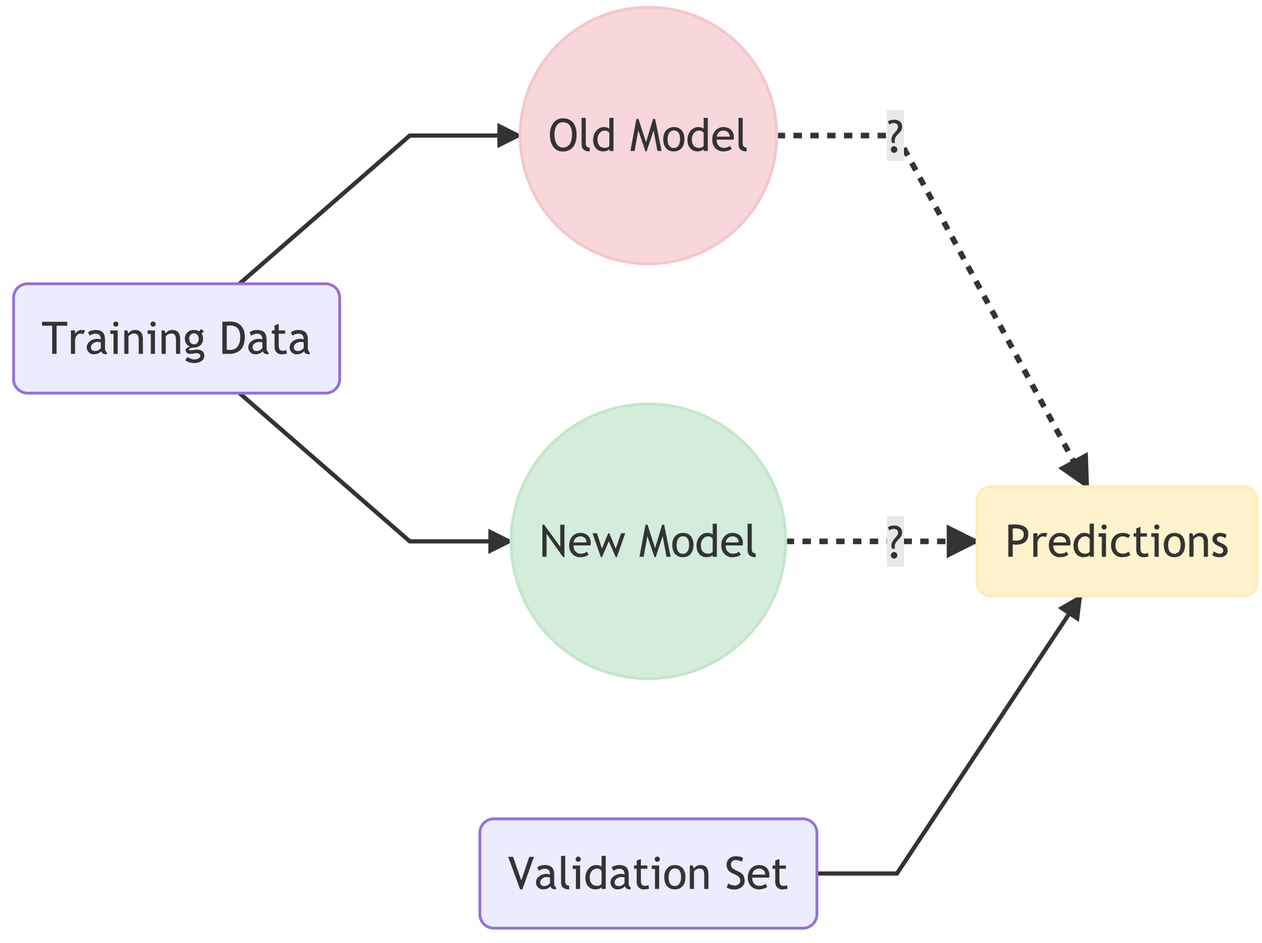
A wild bug appears!

Data Dependency Hell



Data Dependency Hell

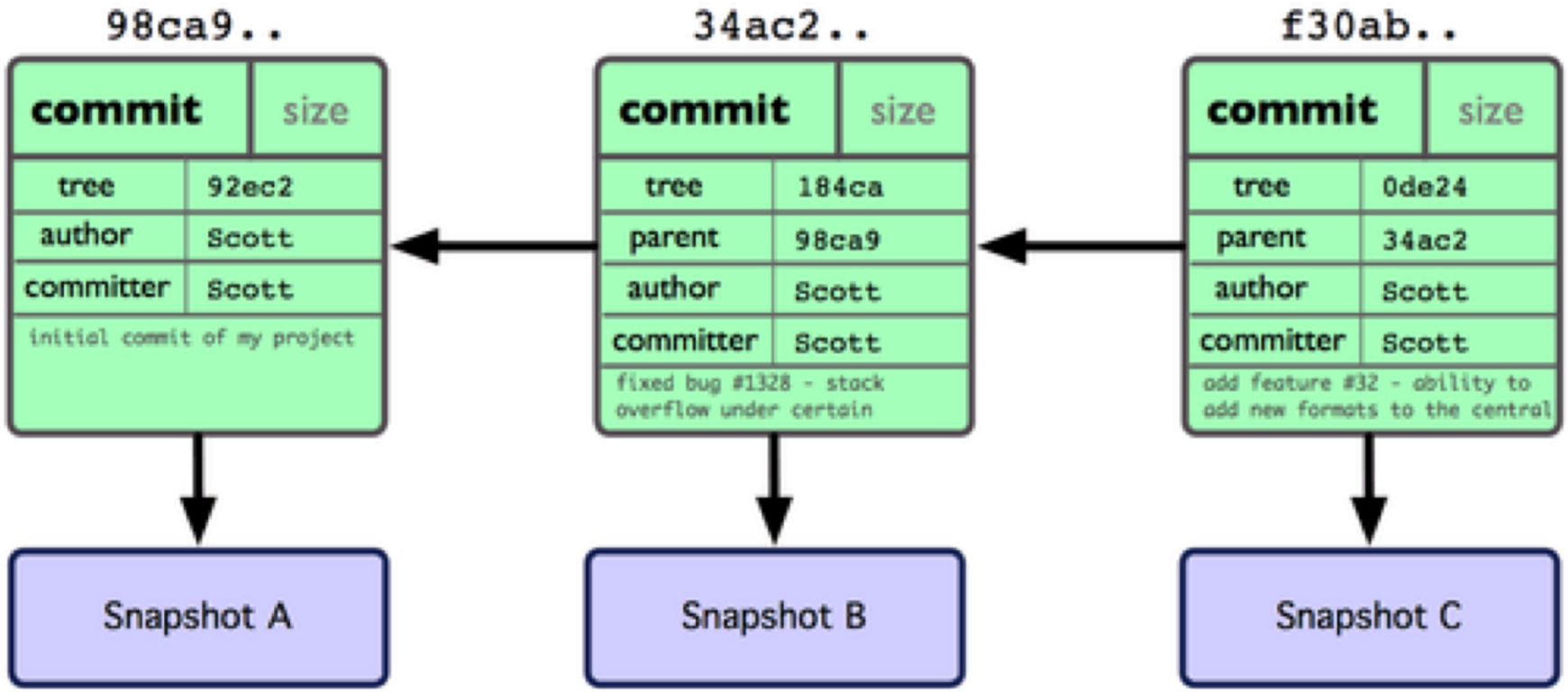
New model updates suffer the same issue



Content Addressable Filesystems

```
def write(file_content):

    # Use the hash as the filename!
    file_name = sha256(file_content.encode()).hexdigest()

    with atomic_write(filename) as f:
        f.write(file_content)
```

Git, Blockchain, Merkle



Pro Git - What a Branch Is

Data Documents

Stream	**Aggregation**	**Top 10**
name = stream version = '1.0'	name = aggregation version = '1.2 bugfix' input = Stream(?)	name = top_10 version = '1.0.1' input = Aggregation(?)

Data Documents

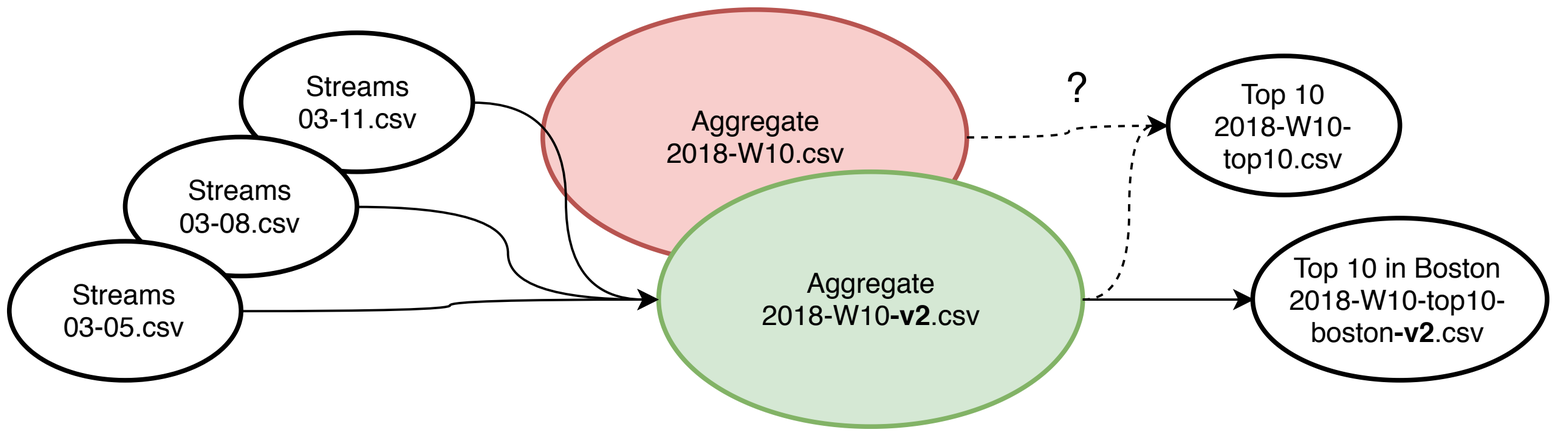
Stream (21b02e)	Aggregation (698939)	Top 10 (3bcad5)
name = stream version = '1.0'	name = aggregation version = '1.2 bugfix' input = 21b02e	name = top_10 version = '1.0.1' input = 698939

The Pseudocode

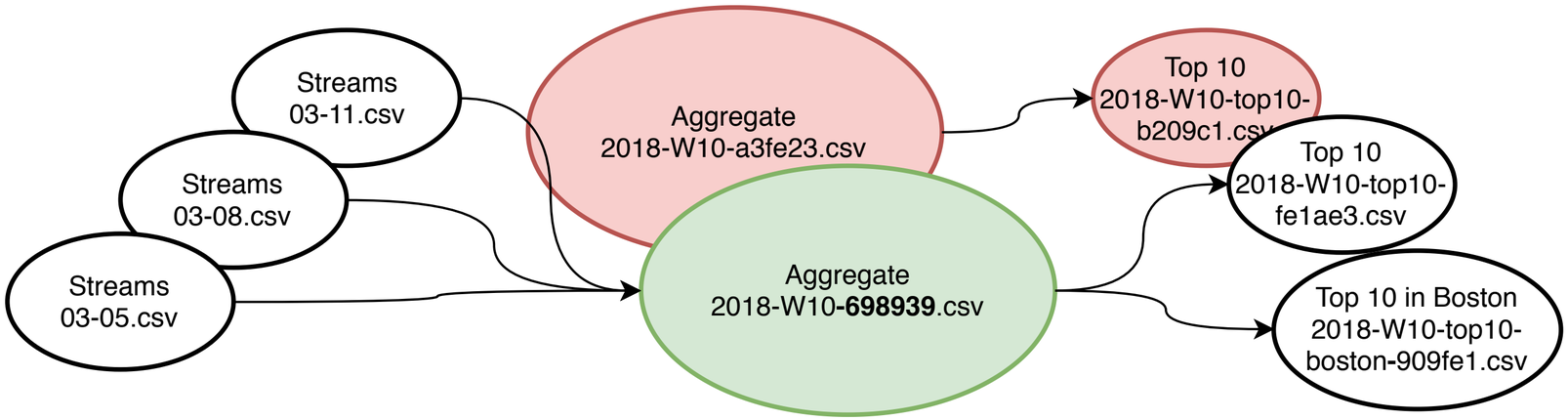
```
def get_salted_version(task):
    """Create a salted id/version for this task and lineage"""

    msg = ""

    # Salt with lineage
    for req in flatten(task.requires()):
        msg += get_salted_version(req)
```

```
    # Uniquely specify this task
    msg += ','.join([
            task.__class__.__name__,
            task.__version__,
        ]
    )
    return sha256(msg.encode()).hexdigest()
```

The resolution



The resolution



Readings

- Why Favor Composition over Inheritance
- Descriptor HowTo Guide
- Salted Graphs
- Pro Git - What a Branch Is
  - Optional Git Pro - Git Internals