Pythonic FP with Coconut

Anthony Khong

16 June 2018

1. Functional Programming
2. Declarative Programming by Example
3. Coconut for Machine Learning Pipeline

Functional Programming

Immutability

Once assigned, a variable cannot change its value.

Speaker notes

The main idea here is that you work with pure functions.

Purity in the mathematical sense: the same inputs give you the same outputs.

In OO, you often work with stateful objects. A method is most definitely not a pure function.

```
>>> def g(xs):
...     ???
...
>>> def f(xs):
...     return xs + [999]
...
>>> xs = [1, 2, 3]
>>> ys = g(xs)
>>> f(xs)
???
```

```python
>>> def g(xs):
...     xs.reverse()
...     return xs
...
>>> def f(xs):
...     return xs + [999]
...
>>> xs = [1, 2, 3]
>>> ys = g(xs)
>>> f(xs)
```

Speaker notes

We say that Python lists are mutable. It carries with it state.

From the point where you define xs, you cannot guarantee that it's the same object.

It is not enough to have xs and the definition of f to find out what f(xs) is.

Here, g is not a pure function, because it modifies a global state. In some sense it is lying to you.

```python
>>> def g(xs):
...     return xs[::-1]
...
>>> def f(xs):
...     return xs + [999]
...
>>> xs = [1, 2, 3]
>>> ys = g(xs)
>>> f(xs)
```

Speaker notes

Superficially, both g and f are doing the same thing as before.

Compare the two slides.

But we changed the implementation of g to make it a pure function.

The diff between reverse and negative step is that the latter makes a new list.

```
λ> g x = ???
λ> f x = x ++ [999]
λ> xs = [1, 2, 3]
λ> ys = g xs
λ> f xs
???
```

Speaker notes

Explain Haskell syntax carefully.

In Haskell you can always guarantee the value of `f xs`.

Anything else, it won't type check. Haskell won't let you do side effects.

```
λ> g x = undefined
λ> f x = x ++ [999]
λ> xs = [1, 2, 3]
λ> ys = g xs
λ> f xs
[1, 2, 3, 999]
```

Speaker notes

There something called the **bottom**, which represents never-ending computation.

```
λ> f x = x ++ [999]
λ> xs = [1, 2, 3]

λ> f xs
[1, 2, 3, 999]
```

Speaker notes

In a real sense, you don't need to know what g is in order to know what `f xs` is. They are irrelevant.

In terms of decoupling. This is a small example that works.

In reality, the side effect could happen somewhere deep in the codebase, where it'd be hard to debug.

Decoupling is good, because you can work with smaller chunks.

Immutability = Predictability

Speaker notes

You'll hear this term a lot from functional programmers. "Your code is easier to reason about."

This is what it boils down to. Without state, your code is predictable.

Declarative Programming

Speaker notes

As a by-product of functional programming, you often write declarative code instead of imperative code.

Declarative vs. Imperative

Say what things are rather than how things are done.

Speaker notes

Take loops as example, how can you implement a loop without changing things? Many times your index always changes!

There are two alternatives for loops: list comprehensions and maps.

If you think about them, they're definitions instead of an instruction to change things.

Let's look at a bigger example.

Sieve of Eratosthenes

Speaker notes

The idea here is that we'd like to find all prime numbers.

So the objective here is to make an infinite stream of prime numbers.

Here is how we do it...

Prime numbers

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90

Speaker notes

Start with a stream of numbers starting from two to infinity.

Take the head of the stream, and filter out all factors of the head.

Repeat this step ad infinitum.

Here is an illustration of the algorithm.

Prime numbers

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Haskell

```haskell
primes :: [Int]
primes = sieve [2..]
where
    sieve (x:xs) = x : sieve (filter (\n -> n `mod` x /= 0) xs)
    sieve []     = []

λ> takeWhile (<60) primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59]
```

Speaker notes

Carefully go through the Haskell syntax.

Python

```
from itertools import count, takewhile

def primes():
    def sieve(numbers):
        head = next(numbers)
        yield head
        yield from sieve(n for n in numbers if n % head)
    return sieve(count(2))

>>> list(takewhile(lambda x: x < 60, primes()))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59]
```

Speaker notes

Note the difference between declarative and imperative programming.

Now I'd like to introduce Coconut...

Coconut

```
from itertools import count, takewhile

def primes():
    def sieve(numbers):
        head = next(numbers)
        yield head
        yield from sieve(n for n in numbers if n % head)
    return sieve(count(2))
```

Speaker notes

The first thing to note here is that Coconut is a superset of Python.

All valid Python is valid Coconut. So you can just write Python if you'd like.

Massage this imperative code to make it 1) more idiomatic and 2) more declarative.

First up is this clunky lambda syntax...

Concise Lambdas

```
from itertools import count, takewhile

def primes():
    def sieve(numbers):
        head = next(numbers)
```

Speaker notes

In Coconut, we can simply write an arrow for a lambda.

It may seem minor, but if you're writing functional code, you often pass functions as arguments.

In fact, this is how you should pass behaviours around instead of objects with methods.

In that case, you'd like your building blocks to be as concise as possible.

It may seem small, but this is very very nice to write functional codes with.

Next up is functional composition...

Forward Piping

```
from itertools import count, takewhile

def primes():
    def sieve(numbers):
        head = next(numbers)
```

Speaker notes

Instead of clumsy brackets, Coconut gives us a forward pipe operator that allows to do F# style code.

There is another type of composition: the dot composition.

It's right to left instead of left to right, which is similar to Haskell.

Whatever you choose Haskell-style or F#-style it's more readable than the original.

I've softened up on this after doing some Clojure, but I'd still prefer the F#-style code.

Next up is currying...

Currying

Speaker notes

Currying is a fancy name for partial application of a function.

In Python, you have the partial module.

The main idea is to start with a function of many arguments, and go down to a function with fewer arguments (usually one).

This makes the syntax more concise, because instead of writing a lambda in takewhile, you write a curried takewhile.

Note that in pure Python, you'd have to apply partial.

It's hard, it's not readable. Nobody would ever do it.

Next is iterator chaining...

Iterator Chaining

```
from itertools import count, takewhile

def primes():
    def sieve(numbers):
        head = next(numbers)
```

Speaker notes

When you write functional code, you often work with lazy lists.

The closest thing you have to this in Python is generators.

In Coconut, there's a specialised syntax to deal with generators.

In particular, you can append to the head.

Here there's still an ugly side effect: the next function.

We instead do pattern matching.

Pattern Matching

```
from itertools import count, takewhile

def primes():
    def sieve([head] :: tail):
        return [head] :: sieve(n for n in tail if n % head)
    return sieve(count(2))

>>> primes() |> takewhile$(x -> x < 60) |> list
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59]
```

Speaker notes

Just like Haskell, we do pattern matching on the head and tail.

Since `sieve` is defined inside the function, we know that it's never empty.

Next is function assigments.

Function Assignments

```
from itertools import count, takewhile

def primes() =
    def sieve([x] :: xs) = [x] :: sieve(n for n in xs if n % x)
```

Speaker notes

In many functional programming languages, you work with expressions instead of statements.

Functions are just any other values. Its return value is simply the last value.

These are called function assignments and we can skip the return statement.

Note that sieve now looks very much like a lambda.

In fact it is, you can write that as an argument of a function.

Finally, we have builtin higher order functions...

Builtin Higher-Order Functions

```
def primes() =
    def sieve([x] :: xs) = [x] :: sieve(n for n in xs if n % x)
    sieve(count(2))

>>> primes() |> takewhile$(x -> x < 60) |> list
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59]
```

Speaker notes

Guido took out reduce saying that it's non-trivial and confusing.

Actually, reduce is just a fold operation in FP, and it's ubiquitous.

This is an important point to make, because if it's not easy to do, you won't do it.

People say Python is a functional language, but it makes your life hard to write FP style.

Coconut

```
def primes() =
    def sieve([x] :: xs) = [x] :: sieve(n for n in xs if n % x)
    sieve(count(2))

>>> primes() |> takewhile$(x -> x < 60) |> list
```

Python

```
from itertools import count, takewhile

def primes():
    def sieve(numbers):
        head = next(numbers)
```

Speaker notes

There are many changes to be made from Python.

There's no right or wrong and better or worse. It's a matter of preference.

As someone who likes FP, I know I prefer the top version.

Coconut

```
def primes() =
    def sieve([x] :: xs) = [x] :: sieve(n for n in xs if n % x)
    sieve(count(2))

>>> primes() |> takewhile$(x -> x < 60) |> list
```

Haskell

```
primes :: [Int]
primes = sieve [2..]
where
    sieve (x:xs) = x : sieve (filter (\n -> n `rem` x /= 0) xs)
    sieve []     = []

λ> takeWhile (<60) primes
```

Speaker notes

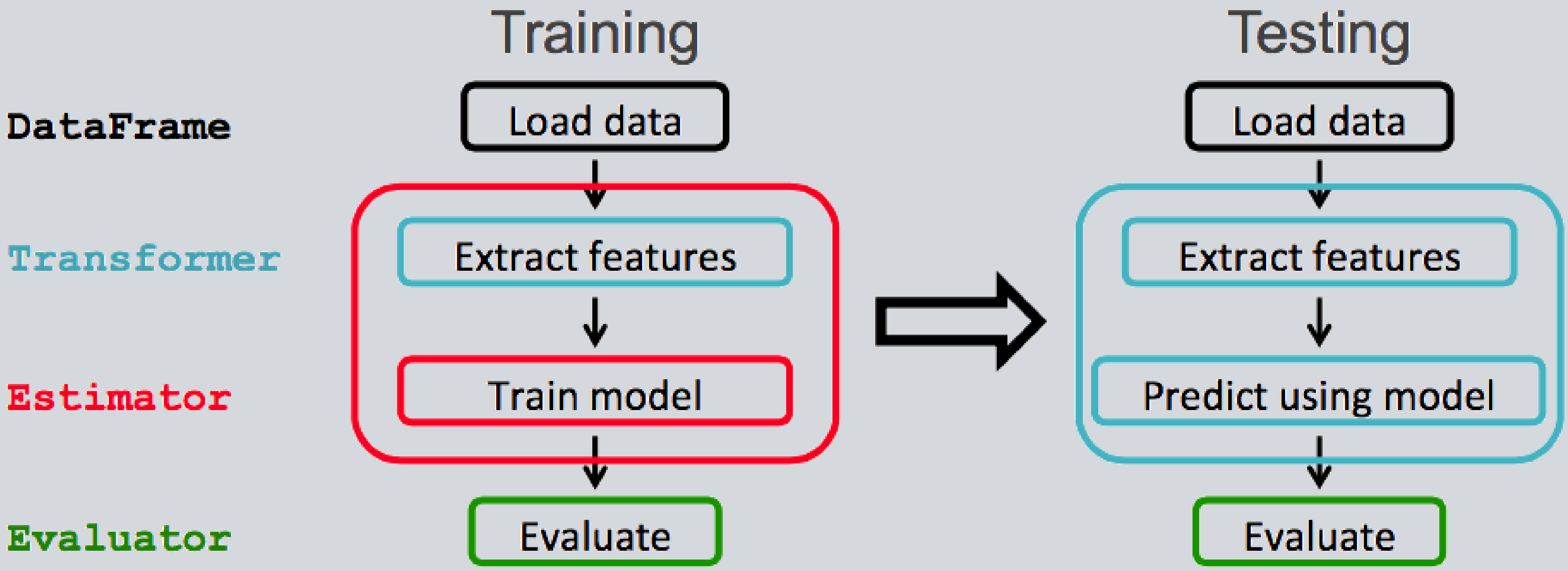
This is just to show how similar these two versions are.

Machine Learning Pipeline

Speaker notes

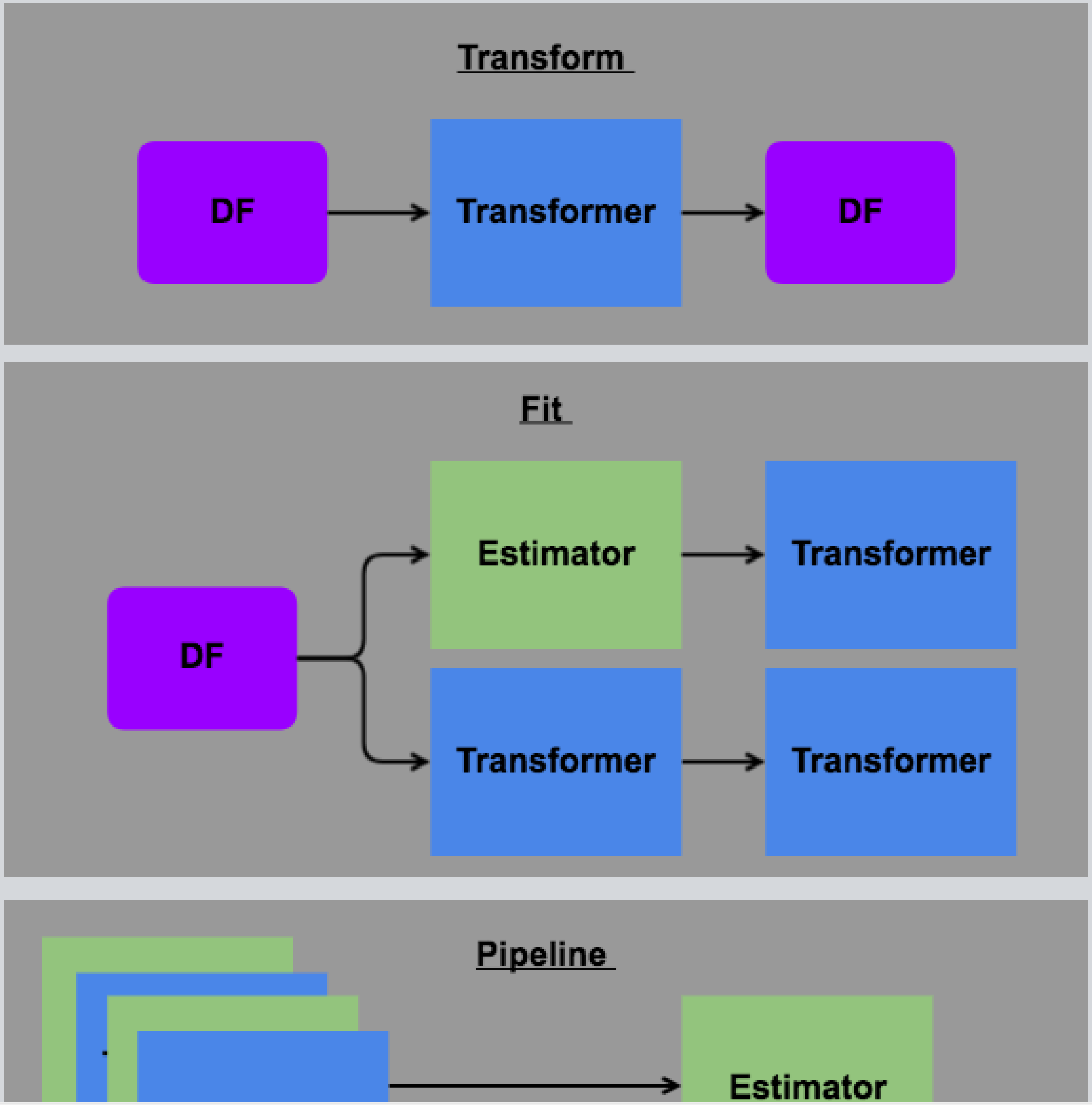
Next, I'd like to talk about a more serious application of Coconut.

Still very much a toy example, but a bit bigger.



Speaker notes

Go through the flowchart very slowly.



Speaker notes

Go through the flowchart very slowly.

Haskell-Style Type Tetris

```
Estimator     = Estimator (DataFrame -> Transformer)
Transformer   = Transformer (DataFrame -> DataFrame)
PipelineStage = Estimator | Transformer

fit            :: PipelineStage   -> DataFrame -> Transformer
transform      :: Transformer     -> DataFrame -> DataFrame
pipeline       :: [PipelineStage] -> PipelineStage
```

Speaker notes

Explain type tetris.

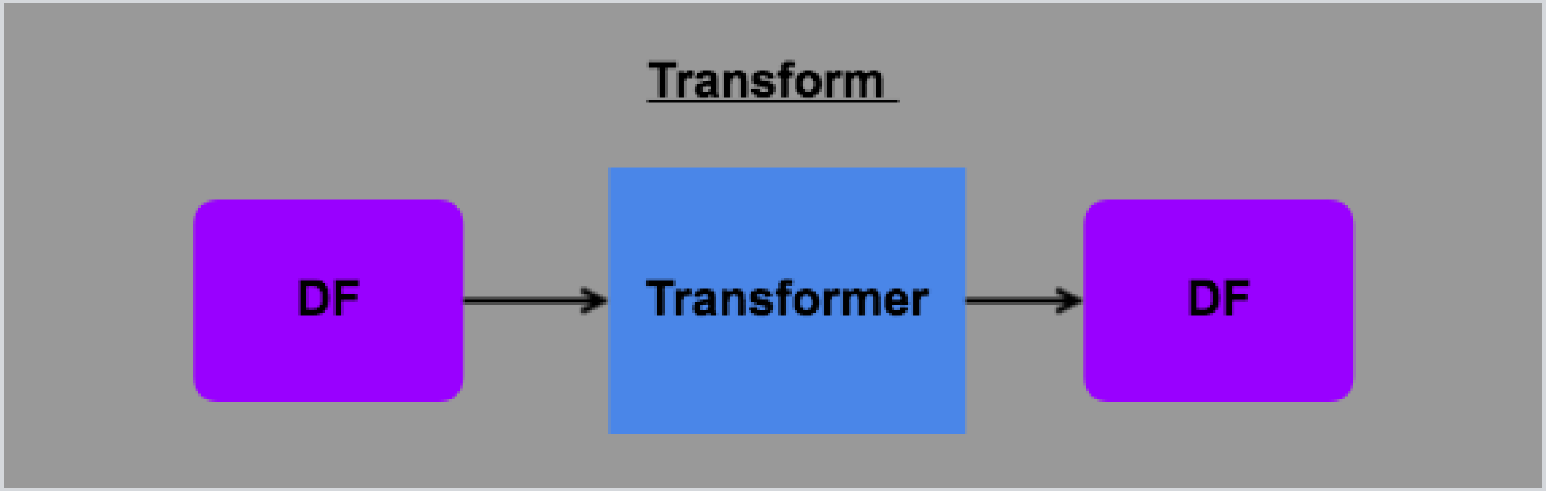
Go through types very slowly.

```
data Estimator(fit_fn)
data Transformer(xform_fn)
```

Transformer

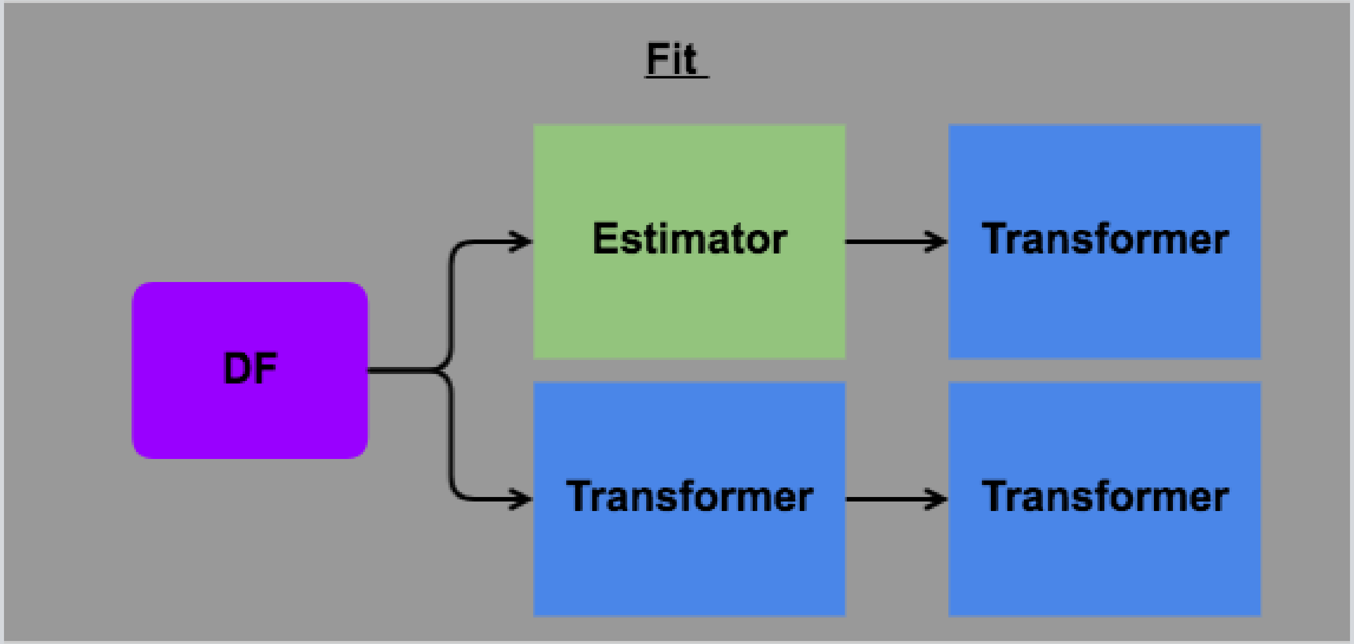
Estimator

```
def transform(Transformer(xform_fn), df) = xform_fn(df)
```

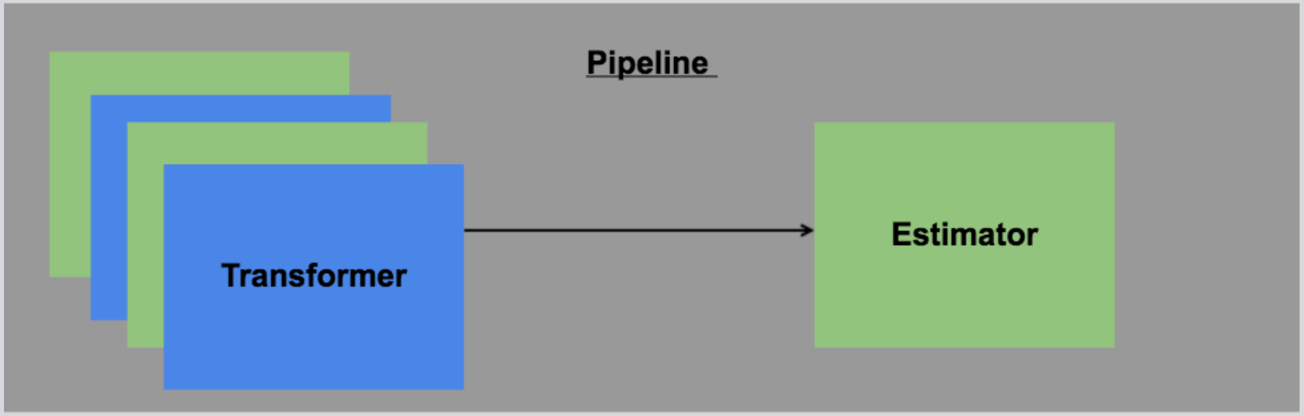


```
def fit(Estimator(fit_fn), df) = fit_fn(df)

@addpattern(fit)
def fit(Transformer(xform_fn), _) = Transformer(xform_fn)
```



What about Pipeline?




```
def pipeline(stages):
    def fit_fn(train_df):
        fitted_stages = []
        for stage in stages:
            fitted_stage = fit(stage, train_df)
            train_df = transform(fitted_stage, train_df)
            fitted_stages.append(fitted_stage)
        def xform_fn(test_df):
            for fitted_stage in fitted_stages:
                test_df = transform(fitted_stage, test_df)
```

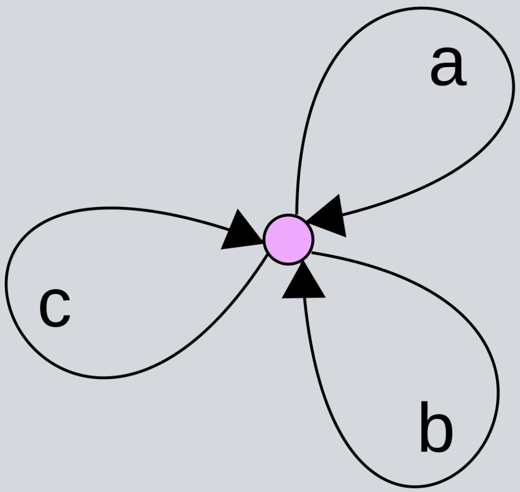
```
        return test_df
    return Transformer(xform_fn)
return Estimator(fit_fn)
```

```
    <aside class="notes">
        <p>Go through the pipeline logic first. Then show the
        <p>Notice triangle of death. This is not how you shou
    </aside>
```


4GIFs
.com

Pipeline Forms a Monoid

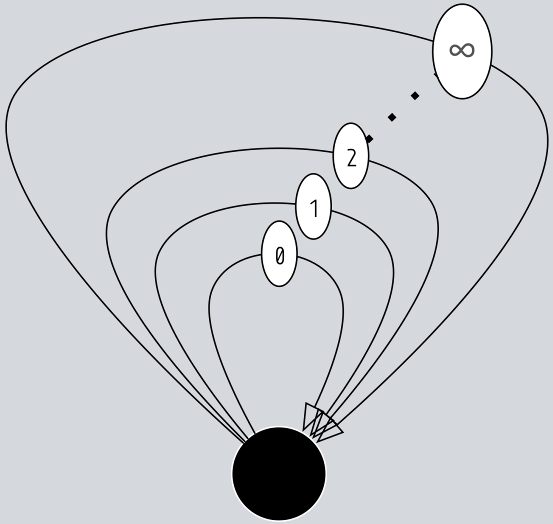
```
pipeline       :: [PipelineStage] -> PipelineStage
```



What is a Monoid?

```
class Monoid m where
    mempty  :: m
    mappend :: m -> m -> m

    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

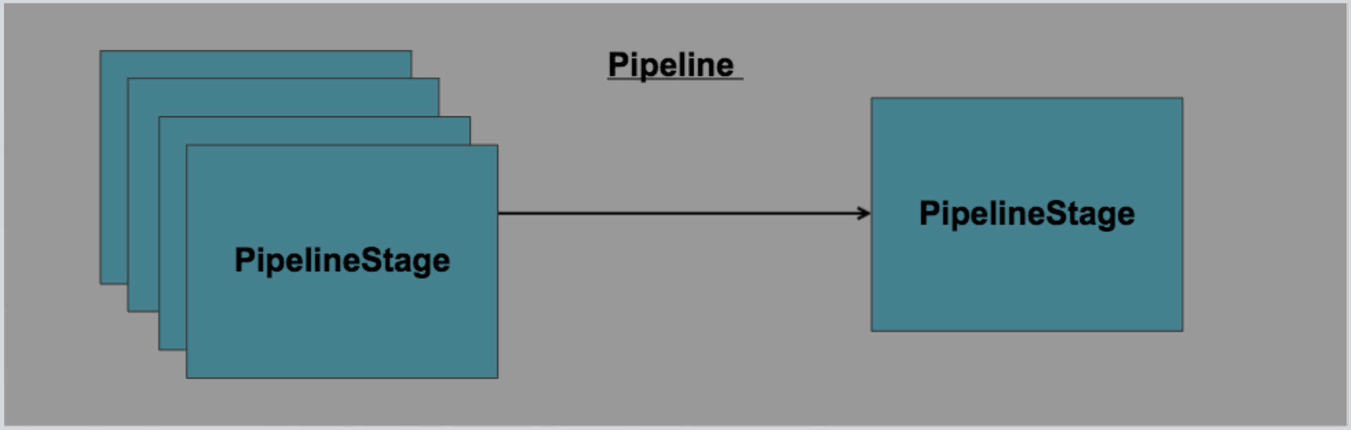


```
def mempty() = Transformer(df -> df)

def mappend(Transformer(fn0), Transformer(fn1)) =
    Transformer(fn1..fn0)

@addpattern(mappend)
def mappend(stage0, stage1) =
    def fit_fn(df) =
        xformer0 = df |> fit$(stage0)
        xformer1 = df |> transform$(xformer0) |> fit$(stage1)
        mappend(xformer0, xformer1)
    Estimator(fit_fn)

def mconcat(stages) = reduce(mappend, stages)
pipeline = mconcat
```



One Hot Encoder

```
import numpy as np

from pipeline import Estimator, Transformer

def one_hot_encoder(column) = fit$(column) |> Estimator

def fit(column, df) =
    factors = np.unique(df[column])
    transform$(column, factors) |> Transformer

def transform(column, factors, df) =
    name = factor -> f'feat:{column}_is_{factor}'
    feats = {name(f): df[column] == f for f in factors}
    df.assign(**feats)
```

```
    <aside class="notes" data-markdown>
```

As you can see, writing an estimator and a transformer is very similar to writing a class in Python.

However, there are no ceremonies such as init and self.

You are only dealing with algebraic data types and pure functions.