



# Hamilton: a python micro-framework for data / feature engineering at Stitch Fix

May 2022

# What to keep in mind for the next ~40 minutes?

1. Hamilton is a new paradigm to create dataframes\*.
2. Using Hamilton is a productivity boost for teams.
3. It's open source - join us on:  
Github: <https://github.com/stitchfix/hamilton>  
Discord: <https://discord.gg/wCqXqBqn73>



## Talk Outline:

> Backstory: who, what, & why

Hamilton

Hamilton @ Stitch Fix

What can you do with Hamilton?

Future Roadmap

# Backstory: who

## Forecasting, Estimation, & Demand (FED) Team

- Data Scientists that are responsible for forecasts that help the business make operational decisions.
  - e.g. staffing levels
- One of the oldest teams at Stitch Fix.

# Backstory: what

Forecasting, Estimation, & Demand (FED) Team

FED workflow:

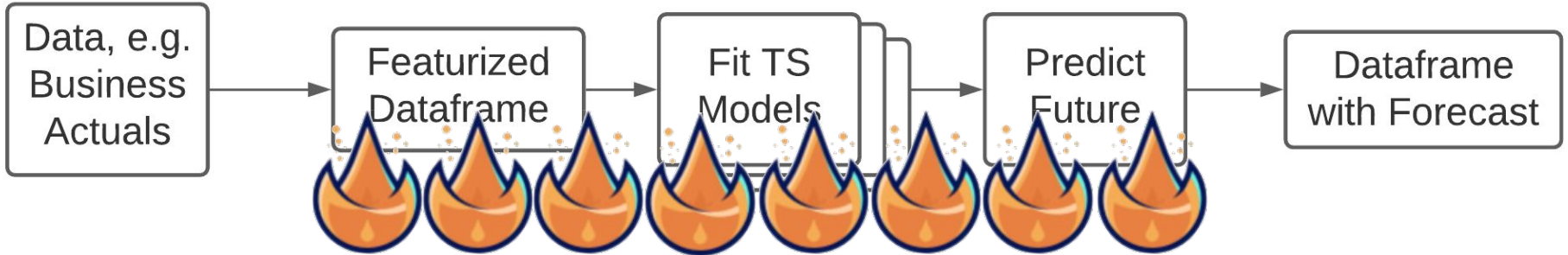


# Backstory: what

Forecasting, Estimation, & Demand Team



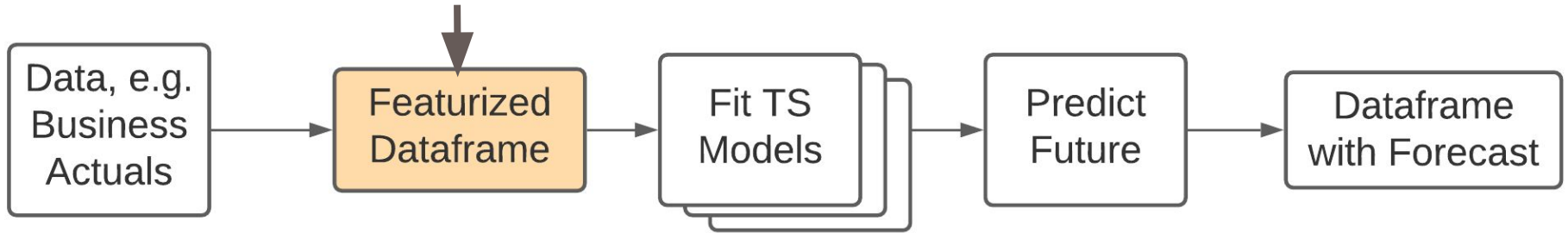
FED workflow:  +  ==



# Backstory: what

Creating a dataframe for time-series modeling.

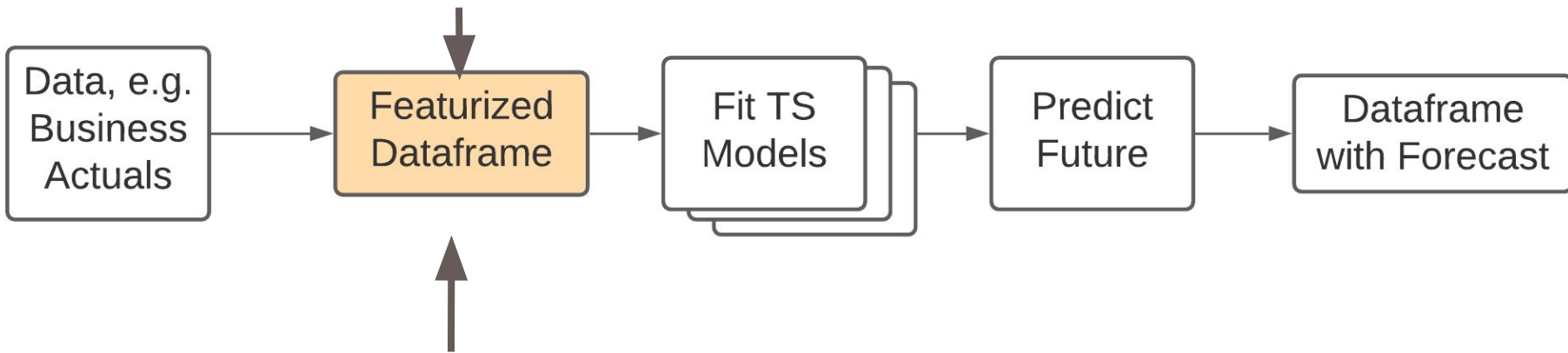
**Biggest problems here**



# Backstory: what

Creating a dataframe for time-series modeling.

**Biggest problems here**



**What  
Hamilton  
helped solve!**



# Backstory: why

What is this dataframe & why is it causing 

?

**O(1000+) of columns** →

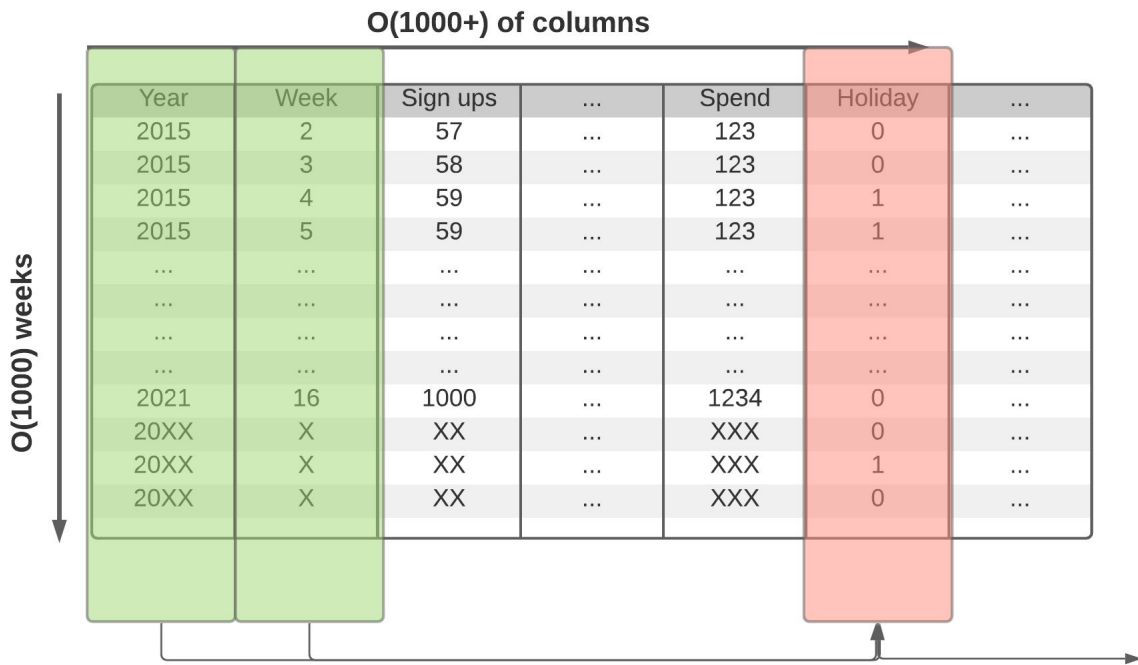
| Year | Week | Sign ups | ... | Spend | Holiday |
|------|------|----------|-----|-------|---------|
| 2015 | 2    | 57       | ... | 123   | 0       |
| 2015 | 3    | 58       | ... | 123   | 0       |
| 2015 | 4    | 59       | ... | 123   | 1       |
| 2015 | 5    | 59       | ... | 123   | 1       |
| ...  | ...  | ...      | ... | ...   | ...     |
| ...  | ...  | ...      | ... | ...   | ...     |
| ...  | ...  | ...      | ... | ...   | ...     |
| ...  | ...  | ...      | ... | ...   | ...     |
| 2021 | 16   | 1000     | ... | 1234  | 0       |
| 20XX | X    | XX       | ... | XXX   | 0       |
| 20XX | X    | XX       | ... | XXX   | 1       |
| 20XX | X    | XX       | ... | XXX   | 0       |

← **O(1000) weeks**

(not big data)

# Backstory: why

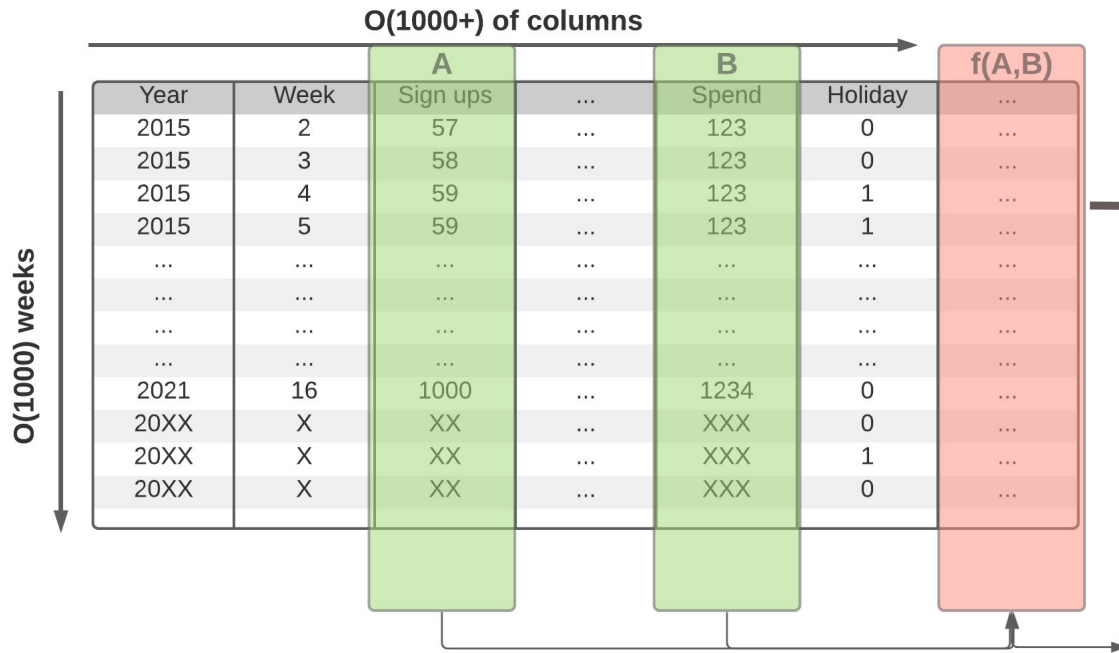
What is this dataframe & why is it causing 🔥 ?



**Columns are functions of other columns**

# Backstory: why

What is this dataframe & why is it causing  ?



Columns are functions of other columns:

$g(f(A,B), \dots)$

$h(g(f(A,B), \dots), \dots)$

etc 

# Backstory: why

Featurization: some example code

```
df = load_dates() # load date ranges
df = load_actuals(df) # load actuals, e.g. spend, signups
df['holidays'] = is_holiday(df['year'], df['week']) # holidays
df['avg_3wk_spend'] = df['spend'].rolling(3).mean() # moving average of spend
df['spend_per_signup'] = df['spend'] / df['signups'] # spend per person signed up
df['spend_shift_3weeks'] = df.spend['spend'].shift(3) # shift spend because ...
df['spend_shift_3weeks_per_signup'] = df['spend_shift_3weeks'] / df['signups']

def my_special_feature(df: pd.DataFrame) -> pd.Series:
    return (df['A'] - df['B'] + df['C']) * weights

df['special_feature'] = my_special_feature(df)
# ...
```

# Backstory: why

Featurization: some example code

```
df = load_dates() # load date ranges
df = load_actuals(df) # load actuals, e.g. spend, signups
df['holidays'] = is_holiday(df['year'], df['week']) # holidays
df['avg_3wk_spend'] = df['spend'].rolling(3).mean() # moving average of spend
df['spend_per_signup'] = df['spend'] / df['signups'] # spend per person signed up
df['spend_shift_3weeks'] = df['spend'].shift(3) # spend 3 weeks ago
df['spend_shift_3weeks_normalized'] = df['spend_shift_3weeks'] / df['signups']

def my_special_feature(df: pd.DataFrame) -> pd.Series:
    return (df['A'] - df['B'] + df['C']) * weights

df['special_feature'] = my_special_feature(df)
# ...
```

*Now scale this code to 1000+ columns & a growing team 🤖*

# Backstory: why



```
df = load_dates() # load date ranges
df = load_actuals()
df['holidays'] = ...
df['avg_3wk_spend'] = ...
df['spend_per_shift'] = ...
df['spend_shift'] = ...

def my_special_feature(df):
    return (df['special_feature'] - df['spend_per_shift'])

df['special_feature'] = ...
# ...
```

Scaling this type of code results in the following:

- lots of heterogeneity in function definitions & behaviors
- inline dataframe manipulations
- code ordering is super important
- monolithic scripts 🤖

- Testing / Unit testing
- Documentation
- Code Reviews
- Onboarding 
- Debugging 



# Backstory - Summary

Code for featurization == 🤖.



## Talk Outline:

Backstory: who, what, & why

> Hamilton

The Outcome

What can you do with Hamilton?

Future Roadmap



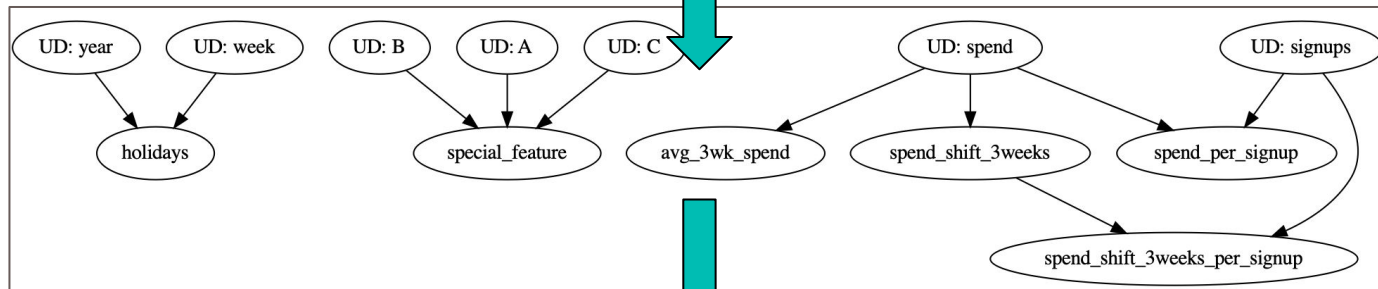
# Hamilton: Code → Directed Acyclic Graph → Object

Code:

```
def holidays(year: pd.Series, week: pd.Series) -> pd.Series:
    """Some docs"""
    return some_library(year, week)
def avg_3wk_spend(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.rolling(3).mean()
def spend_per_signup(spend: pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend / signups
def spend_shift_3weeks(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.shift(3)
def spend_shift_3weeks_per_signup(spend_shift_3weeks: pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend_shift_3weeks / signups
```

User

DAG:



Hamilton

Object  
(e.g. DataFrame):

| Year | Week | Sign ups | ... | Spend | Holiday |
|------|------|----------|-----|-------|---------|
| 2015 | 2    | 57       | ... | 123   | 0       |
| 2015 | 3    | 58       | ... | 123   | 0       |
| 2015 | 4    | 59       | ... | 123   | 1       |
| 2015 | 5    | 59       | ... | 123   | 1       |
| ...  | ...  | ...      | ... | ...   | ...     |
| ...  | ...  | ...      | ... | ...   | ...     |
| ...  | ...  | ...      | ... | ...   | ...     |
| ...  | ...  | ...      | ... | ...   | ...     |
| 2021 | 16   | 1000     | ... | 1234  | 0       |

User

# Hamilton: a new paradigm

1. Write declarative functions!


2. Function name

⇒ output

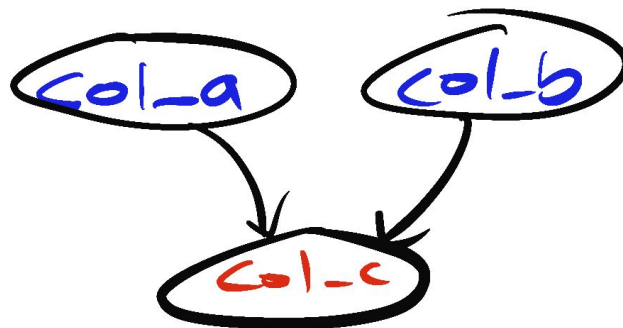
3. Function inputs

⇒ inputs

Code:  $df["col\_c"] = df["col\_a"] + df["col\_b"]$

  
def col\_c(col\_a: pd.Series, col\_b: pd.Series) -> pd.Series:  
 “documentation goes here”  
 return col\_a + col\_b

DAG:




# Hamilton: a new paradigm

4. Use **type hints** for typing checking.
5. Documentation is easy and natural.

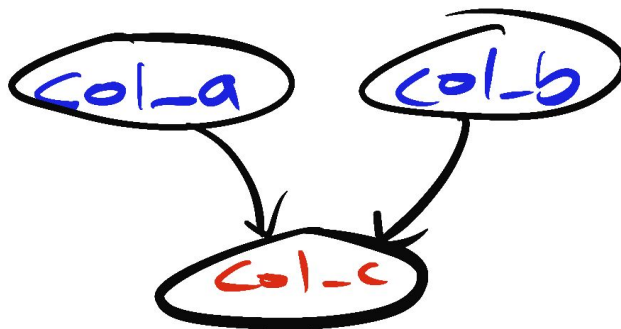
Code:

```
df["col_c"] = df["col_a"] + df["col_b"]
```



```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:  
    "documentation goes here"  
    return col_a + col_b
```

DAG:



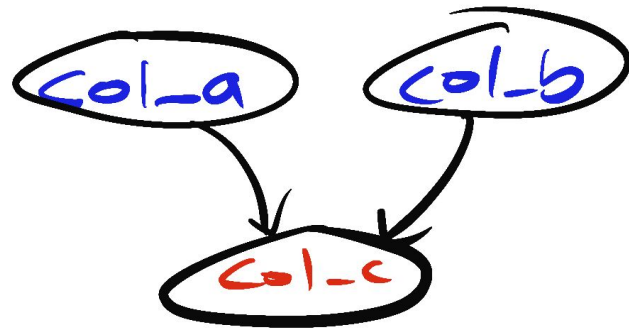
# Hamilton: code to directed acyclic graph - how?

1. Inspect module to extract function names & parameters.
2. Nodes & edges + graph theory 101.

Code:

```
df["col_c"] = df["col_a"] + df["col_b"]  
  
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:  
    "documentation goes here"  
    return col_a + col_b
```

DAG:

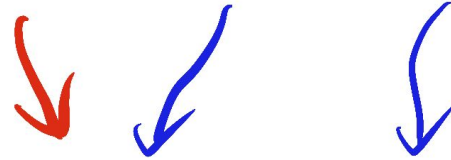


# Hamilton: directed acyclic graph to Object - how?

1. Specify outputs & provide inputs.
2. Determine execution path.
3. Execute functions once.
4. Combine at the end.

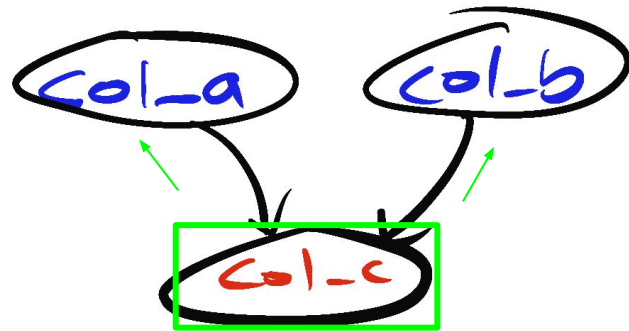
Code:

```
df["col_c"] = df["col_a"] + df["col_b"]
```



```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:  
    “documentation goes here”  
    return col_a + col_b
```

DAG:



# Hamilton: Key Point to remember (1)

Hamilton **requires**:

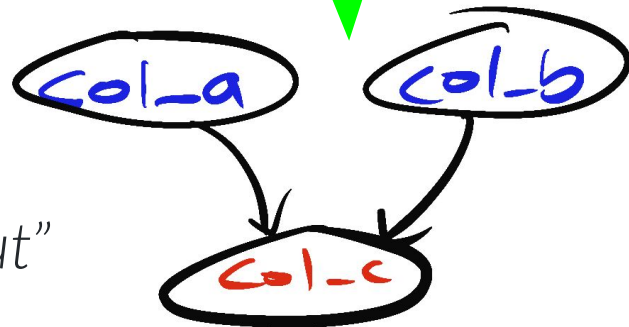
1. Function names
2. Function parameter names

**to match** to stitch together a directed acyclic graph.

*“they declare their inputs & output”*

```
df["col_c"] = df["col_a"] + df["col_b"]
```

```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:  
    "documentation goes here"  
    return col_a + col_b
```



# Hamilton: Key Point to remember (2)

Hamilton users:

```
def holidays(year: pd.Series, week: pd.Series) -> pd.Series:
    """Some docs"""
    return some_library(year, week)
def avg_3wk_spend(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.rolling(3).mean()
def spend_per_signup(spend: pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend / signups
def spend_shift_3weeks(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.shift(3)
def spend_shift_3weeks_per_signup(spend_shift_3w : pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend_shift_3weeks / signups
```

No monolithic script  
to maintain!

Hamilton

| Week | Sign ups | Spend | Holiday |
|------|----------|-------|---------|
| 2    | 57       | 123   | 0       |
| 3    | 58       | 123   | 0       |
| 4    | 59       | 123   | 1       |
| 5    | 59       | 123   | 1       |
| ...  | ...      | ...   | ...     |
| ...  | ...      | ...   | ...     |
| ...  | ...      | ...   | ...     |
| ...  | ...      | ...   | ...     |
| ...  | ...      | ...   | ...     |
| 2021 | 16       | 1000  | 0       |
| 20XX | X        | XX    | 0       |
| 20XX | X        | XX    | 1       |
| 20XX | X        | XX    | 0       |

# Hamilton: in one sentence

A declarative dataflow paradigm.



# Hamilton: why is it called Hamilton?

Naming things is hard...

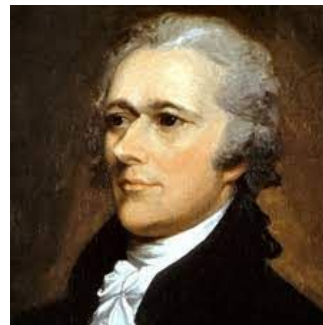
1. Associations with “FED”:

a. Alexander Hamilton is the father of the Fed.

b. FED models business mechanics.

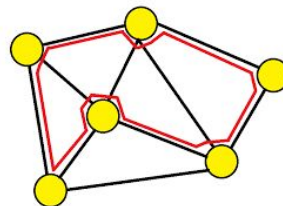
2. We’re doing some basic graph theory.

## apropos Hamilton



$$H_{operator} = \frac{-\hbar^2}{2m} \frac{\partial^2}{\partial x^2} + V(x)$$

Operator associated with kinetic energy      Potential energy





# Example Hamilton Code

So you can get a feel for this paradigm...

# Basic code - defining “Hamilton” functions

my\_functions.py

```
def holidays(year: pd.Series, week: pd.Series) -> pd.Series:
    """Some docs"""
    return some_library(year, week)

def avg_3wk_spend(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.rolling(3).mean()

def spend_per_signup(spend: pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend / signups

def spend_shift_3weeks(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.shift(3)

def spend_shift_3weeks_per_signup(spend_shift_3weeks: pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend_shift_3weeks / signups
```

# Basic code - defining “Hamilton” functions

my\_functions.py

```
def holidays(year: pd.Series, week: pd.Series) -> pd.Series:
    """Some docs"""
    return some_library(year, week)

def avg_3wk_spend(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.rolling(3).mean()

def spend_per_signup(spend: pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend / signups

def spend_shift_3weeks(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.shift(3)

def spend_shift_3weeks_per_signup(spend_shift_3weeks: pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend_shift_3weeks / signups
```

Output Column

Input Column

# Driver code - how do you get a result?

```
from hamilton import driver
config_and_initial_data = { # pass in config, initial data (or load data via funcs)
    'C': 3, # a config variable
    'signups': ..., # can pass in initial data - or pass in at execute time.
    ...
    'year': ...
}
module_name = 'my_functions' # e.g. my_functions.py; can instead `import my_functions`
module = importlib.import_module(module_name) # The python file to crawl
dr = driver.Driver(config_and_initial_data, module) # can pass in multiple modules
output_columns = ['year', 'week', ..., 'spend_shift_3weeks_per_signup', 'special_feature']
df = dr.execute(output_columns) # only walk DAG for what is needed; default obj. is DF.
```

# Driver code - how do you get a result?

```
from hamilton import driver
config_and_initial_data = { # pass in config, initial data (or load data via funcs)
    'C': 3, # a config variable
    'signups': ..., # can pass in initial data - or pass in at execute time.
    ...
    'year': ...
}
module_name = 'my_functions' # e.g. my_functions.py; can instead `import my_functions`
module = importlib.import_module(module_name) # The python file to crawl
dr = driver.Driver(config_and_initial_data, module) # can pass in multiple modules
output_columns = ['year', 'week', ..., 'spend_shift_3weeks_per_signup', 'special_feature']
df = dr.execute(output_columns) # only walk DAG for what is needed; default obj. is DF.
```

# Driver code - how do you get a result?

```
from hamilton import driver
config_and_initial_data = { # pass in config, initial data (or load data via funcs)
    'C': 3, # a config variable
    'signups': ..., # can pass in initial data - or pass in at execute time.
    ...
    'year': ...
}
module_name = 'my_functions' # e.g. my_functions.py; can instead `import my_functions`
module = importlib.import_module(module_name) # The python file to crawl

dr = driver.Driver(config_and_initial_data, module) # can pass in multiple modules

output_columns = ['year', 'week', ..., 'spend_shift_3weeks_per_signup', 'special_feature']

df = dr.execute(output_columns) # only walk DAG for what is needed; default obj. is DF.
```

# Driver code - how do you get a result?

```
from hamilton import driver
config_and_initial_data = { # pass in config, initial data (or load data via funcs)
    'C': 3, # a config variable
    'signups': ..., # can pass in initial data - or pass in at execute time.
    ...
    'year': ...
}
module_name = 'my_functions' # e.g. my_functions.py; can instead `import my_functions`
module = importlib.import_module(module_name) # The python file to crawl

dr = driver.Driver(config_and_initial_data, module) # can pass in multiple modules

output_columns = ['year', 'week', ..., 'spend_shift_3weeks_per_signup', 'special_feature']

df = dr.execute(output_columns) # only walk DAG for what is needed; default obj. is DF.
```



# Driver code - how do you get a result?

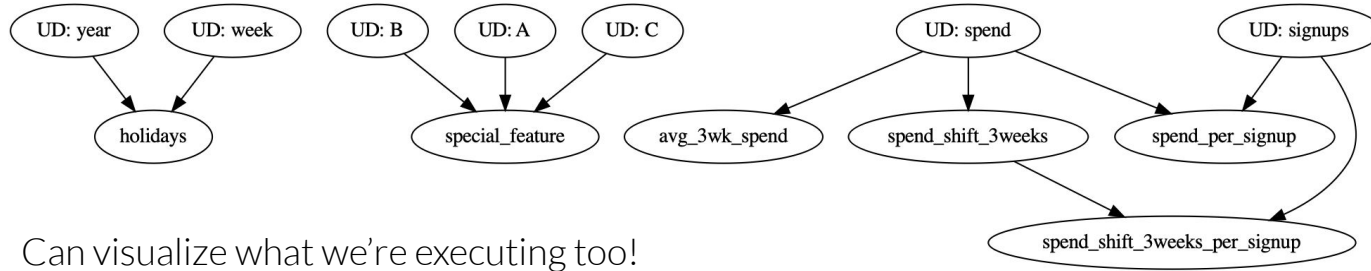
```
from hamilton import driver
config_and_initial_data = { # pass in config, initial data (or load data via funcs)
    'C': 3, # a config variable
    'signups': ..., # can pass in initial data - or pass in at execute time.
    ...
    'year': ...
}
module_name = 'my_functions' # e.g. my_functions.py; can instead `import my_functions`
module = importlib.import_module(module_name) # The python file to crawl
dr = driver.Driver(config_and_initial_data, module) # can pass in multiple modules
output_columns = ['year', 'week', ..., 'spend_shift_3weeks_per_signup', 'special_feature']
df = dr.execute(output_columns) # only walk DAG for what is needed; default obj. is DF.
```

# Driver code - how do you get a result?

```
from hamilton import driver
config_and_initial_data = { # pass in config, initial data (or load data via funcs)
    'C': 3, # a config variable
    'signups': ..., # can pass in initial data - or pass in at execute() time.
    ...
    'year': ...
}
module_name = 'my_functions' # e.g. my_functions.py; can instead `import my_functions`
module = importlib.import_module(module_name) # The python file to crawl
dr = driver.Driver(config_and_initial_data, module) # can pass in multiple modules
output_columns = ['year', 'week', ..., 'spend_shift_3weeks_per_signup', 'special_feature']
df = dr.execute(output_columns) # only walk DAG for what is needed; default obj. is DF.
```

# Driver code - how do you get a result?

```
from hamilton import driver
config_and_initial_data = { # pass in config, initial data (or load data via funcs)
    'C': 3, # a config variable
    's'
```



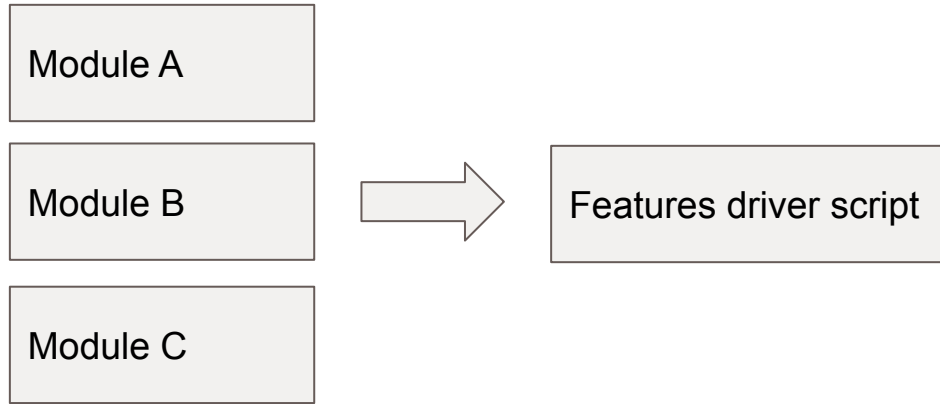
```
output_columns = ['year', 'week', ..., 'spend_shift_3weeks_per_signup', 'special_feature']
```

```
df = dr.execute(output_columns) # only walk DAG for what is needed; default obj. is DF.
```

```
dr.execute_visualization(output_columns, './dag.dot', {...render config...})
```

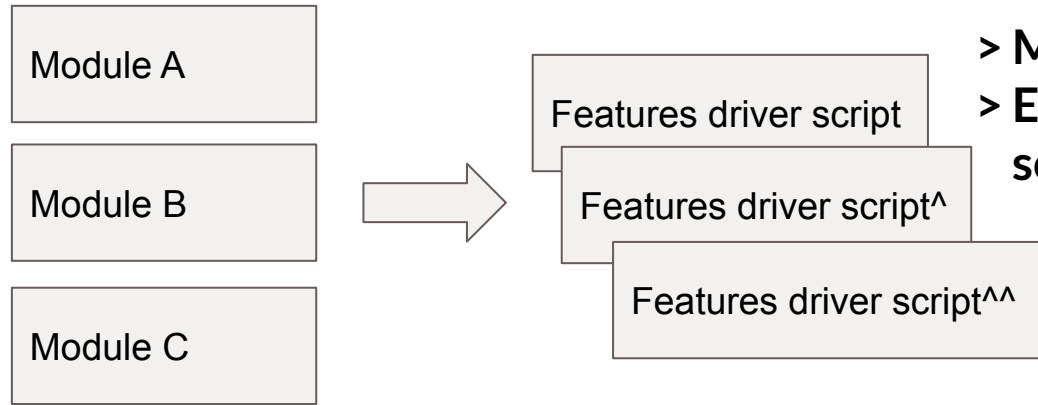
# Implications for your code base

1. Functions are always in modules.
2. Execution script is decoupled from functions.



# Implications for your code base

1. Functions are always in modules.
2. Execution script is decoupled from functions.



- > **Modules enable reuse from day 1.**
- > **Easy to support different “driver” scripts**

# Note: Hamilton is not an orchestration system

1. Hamilton does not replace an orchestration system  
e.g. airflow, kubeflow pipelines, etc.
  
2. Hamilton instead helps you run/model/execute a single step  
in your workflow.  
e.g. you would run Hamilton in a step(s) of your ETL.

**⇒ *Hamilton is a “micro-framework”***

# Open Source: try it for yourself!

> **pip install sf-hamilton**

Get started in < 15 minutes!

Documentation - <https://hamilton-docs.gitbook.io/>

Example

[https://github.com/stitchfix/hamilton/tree/main/examples/hello\\_world](https://github.com/stitchfix/hamilton/tree/main/examples/hello_world)

# Hamilton: Summary

1. A declarative [dataflow](#) paradigm.
2. Users write *declarative* functions that create a DAG *through* function & parameter names.
3. Hamilton handles execution of the DAG; bye bye monolithic scripts.



A large, semi-transparent watermark of the Stitch Fix logo is visible in the background on the left side of the slide. The logo consists of a circular emblem with a grid pattern inside, and the words 'STITCH FIX' are written across it.

## Talk Outline:

Backstory: who, what, & why  
Hamilton

> Hamilton @ Stitch Fix

What can you do with Hamilton?

Future Roadmap

# Hamilton @ SF - after 2+ years in production

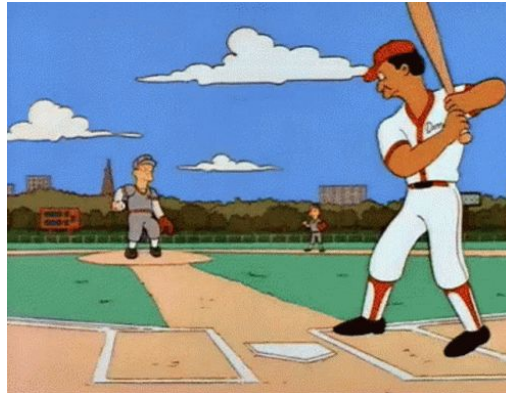


# Stitch Fix FED + Hamilton:

## Original project goals:

- Improve ability to test
- Improve documentation
- Improve development workflow

```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:  
    “documentation goes here”  
    return col_a + col_b
```



Why was it a home run?

# Testing & Documentation

```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:  
    “documentation goes here”  
    return col_a + col_b
```

Output “column” → One function:

1. Single place to find logic.
2. Single function that needs to be tested.
3. Function signature makes providing inputs very easy!
  - a. Function names & input parameters mean something!
4. Functions naturally come with a place for documentation!

⇒ Everything is **naturally** unit testable!

⇒ Everything is **naturally** documentation friendly!

# Workflow improvements

```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:  
    “documentation goes here”  
    return col_a + col_b
```

## What Hamilton also easily enabled:

- Ability to visualize computation
- Faster debug cycles
- Better Onboarding / Collaboration
  - *Bonus:*
    - Central Feature Definition Store

# Visualization

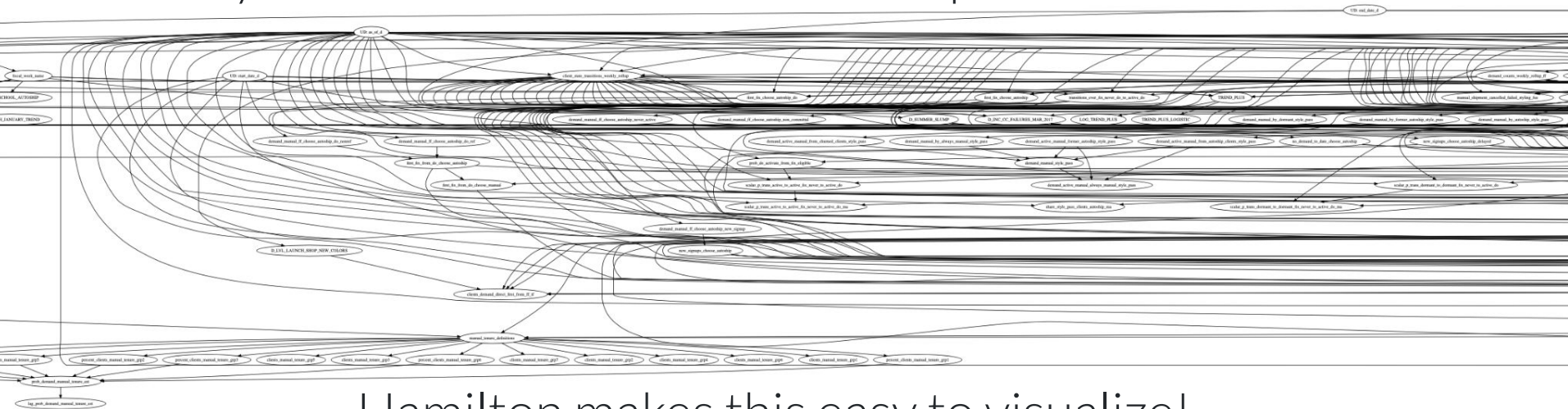
```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:  
    “documentation goes here”  
    return col_a + col_b
```

What if you have 4000+ columns to compute?

# Visualization

```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:  
    “documentation goes here”  
    return col_a + col_b
```

What if you have 4000+ columns to compute?



Hamilton makes this easy to visualize!  
(zoomed out here to obscure names)





# Debugging these functions is easy!

my\_functions.py

```
def holidays(year: pd.Series, week: pd.Series) -> pd.Series:
```

```
    """Some docs"""
```

```
    return some_library(year, week)
```

```
def avg_3wk_spend(spend: pd.Series) -> pd.Series:
```

```
    """Some docs"""
```

```
    return spend.rolling(3).mean()
```

```
def spend_per_signup(spend: pd.Series
```

```
    """Some docs"""
```

```
    return spend / signups
```

Can also import functions into other contexts to help debug.  
e.g. in your REPL:

```
from my_functions import spend_shift_3weeks
```

```
output = spend_shift_3weeks(...)
```

```
def spend_shift_3weeks(spend: pd.Series) -> pd.Series:
```

```
    """Some docs"""
```

```
    return spend.shift(3)
```

```
def spend_shift_3weeks_per_signup(spend_shift_3weeks: pd.Series, signups: pd.Series) -> pd.Series:
```

```
    """Some docs"""
```

```
    return spend_shift_3weeks / signups
```

# Collaborating on these functions is easy!

my\_functions.py

```
def holidays(year: pd.Series, week: pd.Series) -> pd.Series:
    """Some docs"""
    return some_library(year, week)

def avg_3wk_spend(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.rolling(3).mean()

def spend_per_signup(spend: pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend / signups

def spend_shift_3weeks(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.shift(3)

def spend_shift_3weeks_per_signup(spend_shift_3weeks: pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend_shift_3weeks / signups
```

Easy to assess impact & changes when:

- names mean something
- adding a new input
- changing the name of a function
- adding a brand new function
- deleting a function

⇒ Code reviews are much faster!

⇒ Easy to pick up where others left off!

# Stitch Fix FED's Central Feature Definition Store

A nice byproduct of using Hamilton!

How they use it:

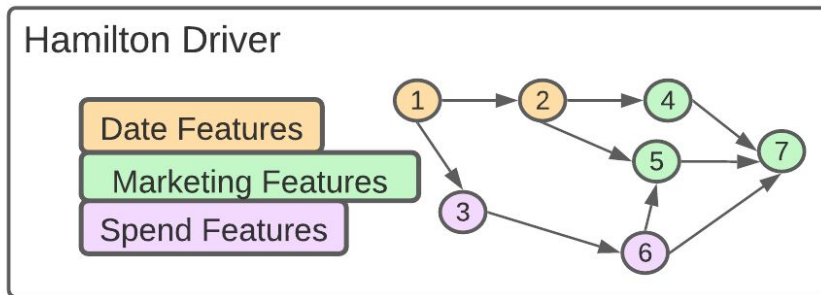
1. Function names follow team convention.
  - a. e.g. **D\_** *prefix* indicates date feature

# Stitch Fix FED's Central Feature Definition Store

A nice byproduct of using Hamilton!

How they use it:

1. Function names follow team convention.
2. It's organized into thematic modules, e.g. `date_features.py`.
  - a. Allows for working on different part of the DAG easily



# Stitch Fix FED's Central Feature Definition Store

**A nice byproduct of using Hamilton!**

## **How they use it:**

1. Function names follow team convention.
2. It's organized into thematic modules, e.g. `date_features.py`.
3. It's in a central repository & versioned by git:
  - a. Can easily find/use/reuse features!
  - b. Can recreate features from different points in time easily.



# FED Testimonials

Just incase you don't believe me

# Testimonial (1)

Danielle Q.



*“the encapsulation of the logic in a single named function makes adding nodes/edges simple to understand, communicate, and transfer knowledge”*

E.g.:

- Pull Requests are easy to review.
- Onboarding is easy.



## Testimonial (2)

Shelly J.



*“I like how easy-breezy it is to add new nodes/edges to the DAG to support evolving business needs.”*

E.g.

- new marketing push & we need to add a new feature:
  - **this takes minutes**, *not hours!*



# Hamilton @ Stitch Fix

FED Impact Summary

# FED Impact Summary

```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:  
    “documentation goes here”  
    return col_a + col_b
```

## With Hamilton, the FED Team gained:

- Naturally testable code. *Always.*
- Naturally documentable code.
- Dataflow visualization for free.
- Faster debug cycles.
- A better onboarding & collaboration experience
  - Central Feature Definition Store as a by product!



-----  
Total



Home Run!

# FED Impact Summary

```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:  
    “documentation goes here”  
    return col_a + col_b
```

## With Hamilton, the FED Team gained:

- Nat [claim]
- Nat By using Hamilton, the FED team can
- Dat ***continue to scale*** their code base,
- Fas without impacting team productivity
- Ab [claim]
- Question: is that true of your feature code base?

Total



Home Run!

A large, semi-transparent watermark of the Stitch Fix logo is visible in the background on the left side of the slide. The logo consists of a circular emblem with a grid pattern inside, and the words "STITCH FIX" are written around the perimeter of the circle.

## Talk Outline:

Backstory: who, what, & why  
Hamilton

Hamilton @ Stitch Fix

> What can you do with Hamilton?

Future Roadmap







# What can you do with Hamilton?

1. Using it within any ETL system
2. Scale to big data
3. Model any dataflow

# 1. Using Hamilton within any ETL system

ETL Framework compatibility:

- all ETL systems that run python 3.6+.

E.g.    Airflow      
          Metaflow     
          Dagster       
          Prefect       
          Kubeflow     
          etc.         

# 1. Using Hamilton within any ETL system

## ETL Recipe:

1. Write Hamilton functions & “driver” code.
2. Publish your Hamilton functions in a package, or import via other means (e.g. checkout a repository).
3. Include *sf-hamilton* as a python dependency
4. Have your ETL system execute your “driver” code.
5. Profit.



## 2. Scale to big data

Hamilton comes with the following integrations:

- Dask
- Ray
- Pandas on Spark (3.2+)

Coming soon:

- Modin

### Cool thing:

- **Only *driver* code needs to be changed.**
- **Makes it easy to switch “backends”.**

# Take this code – and scale it without changing it

my\_functions.py

```
def holidays(year: pd.Series, week: pd.Series) -> pd.Series:
    """Some docs"""
    return some_library(year, week)

def avg_3wk_spend(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.rolling(3).mean()

def spend_per_signup(spend: pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend / signups

def spend_shift_3weeks(spend: pd.Series) -> pd.Series:
    """Some docs"""
    return spend.shift(3)

def spend_shift_3weeks_per_signup(spend_shift_3weeks: pd.Series, signups: pd.Series) -> pd.Series:
    """Some docs"""
    return spend_shift_3weeks / signups
```

# Just how easy it is: Example using Dask – only modify the “driver” script

```
from dask.distributed import Client
from hamilton import driver
from hamilton.experimental import h_dask
dag_config = {...}

bl_module = importlib.import_module('my_functions') # business logic functions
loader_module = importlib.import_module('data_loader') # functions to load data

client = Client(...)
adapter = h_dask.DaskGraphAdapter(client)






dr = driver.Driver(dag_config, bl_module, loader_module, adapter=adapter)

output_columns = ['year', 'week', ..., 'spend_shift_3weeks_per_signup', 'special_feature']

df = dr.execute(output_columns) # only walk DAG for what is needed
```

# 3. Model any dataflow

Hamilton allows you to model any *dataflow*!






- Pandas? 
- Scikit-learn models? 
- Numpy matrices? 
- Ibis Project? 
- Custom python object? 

## What Hamilton provides:

- **lineage insights for free**
- **ability to attach “tags” to functions**
- **ask meta questions**

# 3. Model any dataflow

Hamilton allows you to model any *dataflow*!

- Pandas? 
- Scikit-learn models? 
- Numpy matrices? 
- Ibis Project? 
- Custom python object? 

## What Hamilton provides:

- **lineage insights for free**
- **ability to attach “tags” to functions**
- **ask meta questions**

Would love contributions here!



# A common Hamilton pattern

Here's a common pattern

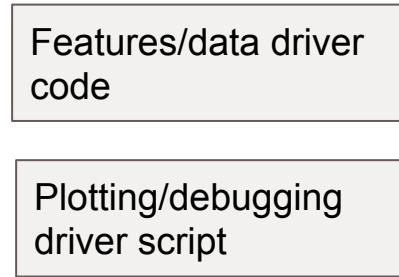
# 3. Model any dataflow - common pattern

Python Modules:



Driver Scripts:

(responsible for data you want to save/use)



Save data/artifact.

# 3. Model any dataflow - common pattern

Python Modules:

Driver Scripts:

(responsible for data you want to save/use)

Data Loading

Transforms 1

Transforms 2

Group By  
Transforms



<-advanced  
uses ->

Features/data driver  
code

Plotting/debugging  
driver script

Chained drivers code



Save data/artifact.



A large, semi-transparent watermark of the Stitch Fix logo is visible in the background on the left side of the slide. The logo consists of a circular emblem with a grid pattern inside, and the words 'STITCH FIX' written around it.

## Talk Outline:

Backstory: who, what, & why  
Hamilton

Hamilton @ Stitch Fix

What can you do with Hamilton?

> Future Roadmap

# Future Roadmap

## Data Quality:

- Runtime inspection of data is a possibility.

Task: incorporate expectations, ala [Pandera](#), on functions.

e.g.

```
@check_output({'type': float, 'range': (0.0, 10000.0)})
def SOME_IMPORTANT_OUTPUT(input1: pd.Series, input2: pd.Series) -> pd.Series:
    """Does some complex logic"""
```

or:

```
schema = ...
@check_output.pandera(schema=schema)
def SOME_IMPORTANT_OUTPUT(input1: pd.Series, input2: pd.Series) -> pd.Series:
    """Does some complex logic"""
```

# Future Roadmap

## Numba:

- [Numba](#) makes your code run much faster.

Task: wrap Hamilton functions with `numba.jit` and compile the graph for speedy execution!

E.g. Scale your numpy & simple python code to:

- GPUs
- C/Fortran like speeds!

# Future Roadmap

## Responding to feedback / feature requests:

- If you have ideas/issues, would love to hear them.

Best way:

- come chat with us on [discord](#)
- file issues on github
- we like to understand your use case too!

# Future Roadmap

## Graduating dask/ray/spark support:

- To do so, we need feedback on the APIs!

Would love to hear:

- if they do or don't work for you?
- what documentation needs to be improved/added?
- etc.

# Future Roadmap

We have few more things :

<https://github.com/stitchfix/hamilton/issues>

Please vote (❤️, 👍, etc) for what we should prioritize!



# To Conclude

Some TL:DRs

# To Conclude

```
def col_c(col_a: pd.Series, col_b: pd.Series) -> pd.Series:  
    “documentation goes here”  
    return col_a + col_b
```

1. Hamilton is a new paradigm to describe data flows.
2. It grew out of a need to tame a feature code base; it'll make yours better too!
3. The Hamilton paradigm can provide teams with multiple productivity improvements & scales with code bases.



# Thanks for listening – would love your feedback!

```
> pip install sf-hamilton
```

 on github



 create & vote on issues on github

 join us on [discord](https://discord.gg/wCqxqBqn73)

(<https://discord.gg/wCqxqBqn73>)



# Thank you! Questions?

 @stefkrawczyk  
 linkedin.com/in/skrawczyk

Try out Stitch Fix → [goo.gl/Q3tCQ3](https://goo.gl/Q3tCQ3)

STITCH FIX