Hamilton: drop procedural scripts in favor of declarative functions

Stefan Krawczyk
CEO DAGWorks Inc. (YCW23)
DAGWORKS

@

Motivation:

1. Code lives for longer than you intend it to.
2. “Bad code habits” slow you/your team down.

Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
```

Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
```

Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
```

Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
```

Example: Creating a dataframe (e.g. for ML training)

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

Example: Creating a dataframe (e.g. for ML training)

```python
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

Problem: unit & integration testing; data quality 👎

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
   df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
   df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

Problem: code readability & documentation 🧐

```python
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
    df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
    df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

?

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

Problem: difficulty in tracing lineage 🤯

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
    df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
    df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

Problem: code reuse and duplication

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
    df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
    df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

Problem: onboarding 📈 & debugging 📈

```
df = loader.load_actuals(dates) # e.g. spend, signups
if config['region'] == 'UK':
    df['holidays'] = is_uk_holiday(df['year'], df[' week'])
else:
    df['holidays'] = is_holiday(df['year'], df['week'])
df['avg_3wk_spend'] = df['spend'].rolling(3).mean()
df['acquisition_cost'] = df['spend'] / df['signups']
df['spend_shift_3weeks'] = df['spend'].shift(3)
df['special_feature1'] = compute_bespoke_feature(df)
df['spend_b'] = multiply_columns(df['acquisition_cost'], df['B'])
save_df(df, "some_location")
```

😬 Now picture the passage of time: personnel Δ, sophistication ⬆️, etc

What is Hamilton?

micro-framework for defining dataflows using declarative functions

SWE best practices: ☑️ testing ☑️ documentation ☑️ modularity/reuse

```
pip install sf-hamilton [came from Stitch Fix]
```

www.tryhamilton.dev ← uses pyodide!

Old Way vs Hamilton Way:

Instead of

```python
df['c'] = df['a'] + df['b']
df['d'] = transform(df['c'])
```

Outputs == Function Name

Inputs == Function Arguments

You declare

```python
def c(a: pd.Series, b: pd.Series) -> pd.Series:
    """Sums a with b"""
    return a + b

def d(c: pd.Series) -> pd.Series:
    """Transforms C to ..."""
    new_column = _transform_logic(c)
    return new_column
```

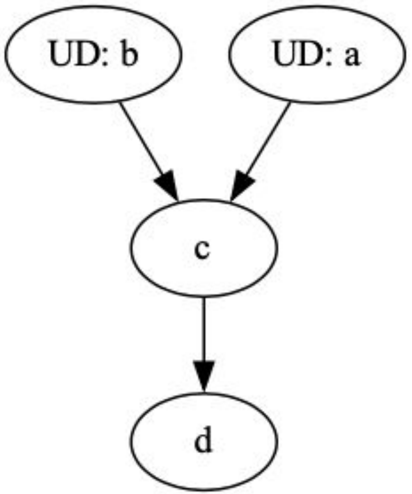
Full Hello World

(Note: works for any python object type)

Functions

```python
# feature_logic.py
def c(a: pd.Series, b: pd.Series) -> pd.Series:
    """Sums a with b"""
    return a + b

def d(c: pd.Series) -> pd.Series:
    """Transforms C to ..."""
    new_column = _transform_logic(c)
    return new_column
```



Driver says what/when to execute

```python
# run.py
from hamilton import driver
import feature_logic
dr = driver.Driver({'a': ..., 'b': ...}, feature_logic)
df_result = dr.execute(['c', 'd'])
print(df_result)
```

Benefits: More reliable & maintainable code

```
def height_zero_mean_unit_variance(height_zero_mean: pd.Series,
                                   height_std_dev: pd.Series) -> pd.Series:
    return height_zero_mean / height_std_dev
```

Testing: easier to unit & integration test.

Benefits: More reliable & maintainable code

```
@check_output(data_type=np.float64, range=(-5.0, 5.0), allow_nans=False)
def height_zero_mean_unit_variance(height_zero_mean: pd.Series,
                                   height_std_dev: pd.Series) -> pd.Series:
    return height_zero_mean / height_std_dev
```

Testing: easier to unit & integration test.

Data quality: runtime checks via annotation.

Benefits: More reliable & maintainable code

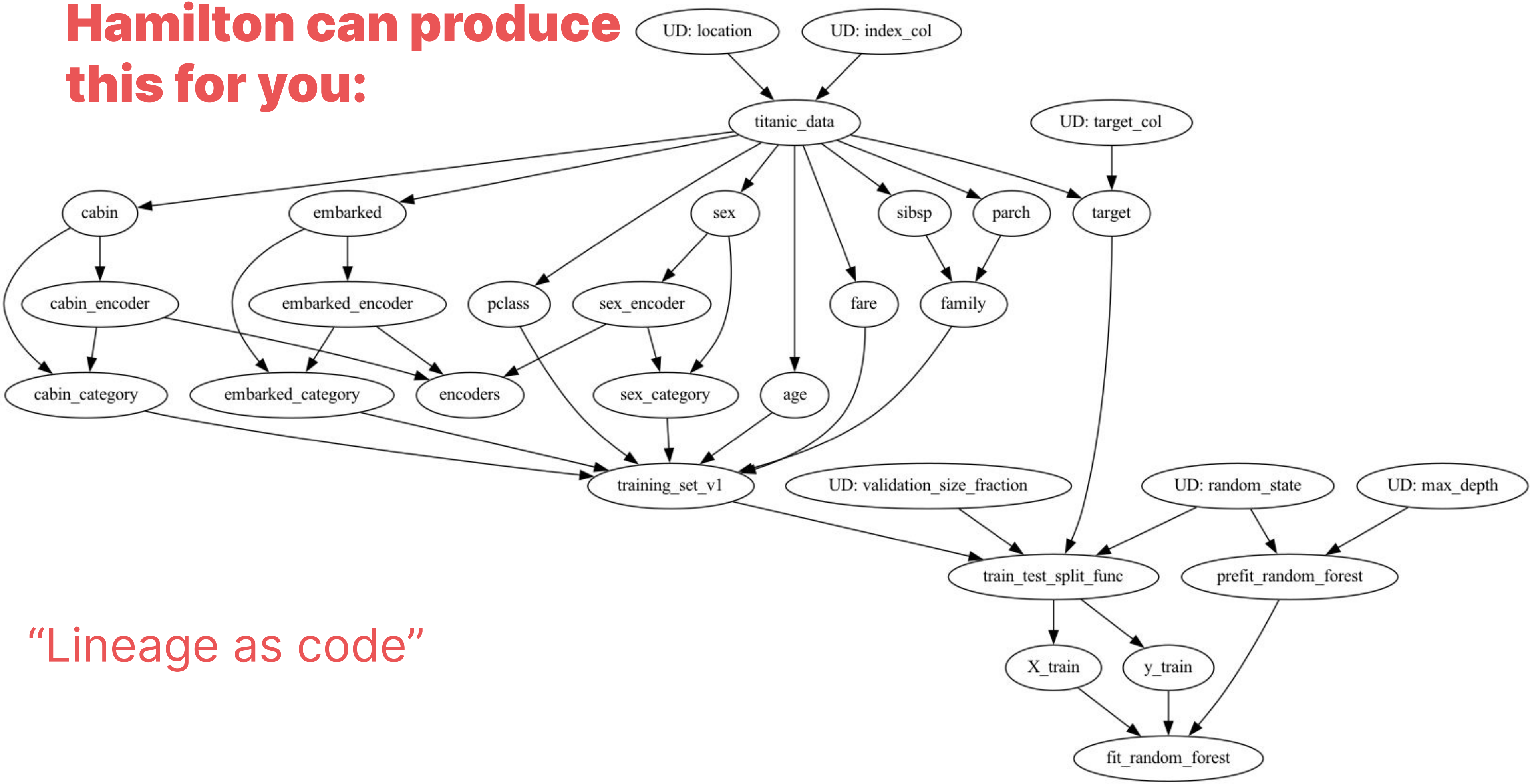
```
# client_features.py
@tag(owner='Data-Science', pii='False')
@check_output(data_type=np.float64, range=(-5.0, 5.0), allow_nans=False)
def height_zero_mean_unit_variance(height_zero_mean: pd.Series,
                                   height_std_dev: pd.Series) -> pd.Series:
    """Zero mean unit variance value of height"""
    return height_zero_mean / height_std_dev
```

Testing: easier to unit & integration test.

Data quality: runtime checks via annotation.

Self-documenting: naming, doc strings, annotations, & visualization

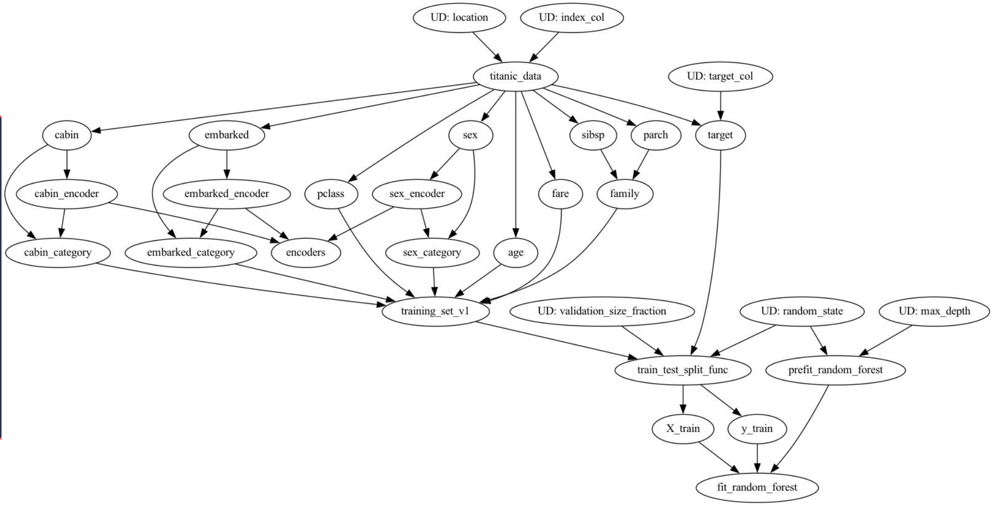
Hamilton can produce this for you:



“Lineage as code”

Benefits: Faster iteration & collaboration

```python
# client_features.py
@tag(owner='Data-Science', pii='False')
@check_output(data_type=np.float64, range=(-5.0, 5.0), allow_nans=False)
def height_zero_mean_unit_variance(height_zero_mean: pd.Series,
                                   height_std_dev: pd.Series) -> pd.Series:
    """Zero mean unit variance value of height"""
    return height_zero_mean / height_std_dev
```



Reusable: Functions are in modules. Reusable from day 0.

Modular: Can define different versions/implementations surgically.

Portable: Runs anywhere python runs. Has some hooks for ray, dask, pyspark.

TL;DR:

Don't miss your shot (⭐):

1. Ditch procedural scripts. They're a pain to manage & maintain.
2. Write declarative functions. Make you & your team happier.

Star Hamilton - ⭐ https://github.com/dagworks-inc/hamilton 👈

Thanks!
Come get a sticker!

Hamilton:

www.tryhamilton.dev

Hamilton (@hamilton_os) / Twitter

⭐ https://github.com/dagworks-inc/hamilton 👈

📚 https://hamilton.dagworks.io

Me: stefan@dagworks.io

https://twitter.com/stefkrawczyk

https://www.linkedin.com/in/skrawczyk/

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 @ SF models business mechanics.

$$H_{operator} = \underbrace{\frac{-\hbar^2}{2m}\frac{\partial^2}{\partial x^2}}_{\text{Operator associated with kinetic energy}} + \underbrace{V(x)}_{\text{Potential energy}}$$

2. We’re doing some basic graph theory.

apropos Hamilton