Beyond Linearity

Building reactive notebooks for data

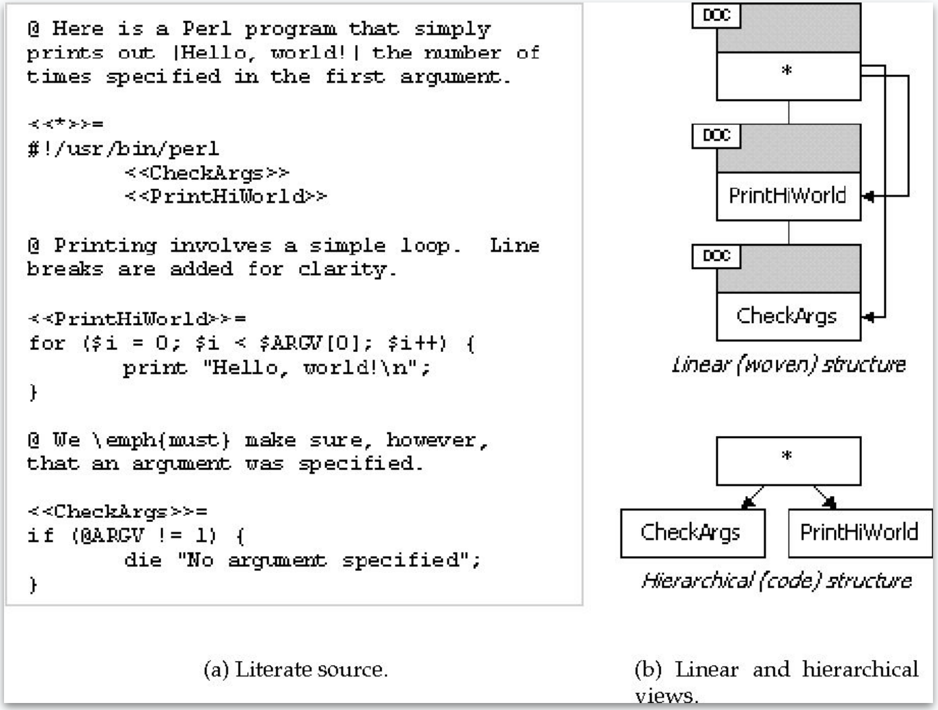
Caitlin Colgrove, CTO @ Hex

Poll: how do code notebooks make you *feel*?

A. I use notebooks for everything! Analysis, text editing, email... all notebooks!
B. They’re useful sometimes but they have their drawbacks.
C. I will literally quit my job if they make me use a notebook.
D. You mean, like... to write in?

Historical background: literate programming

In 1984, Donald Knuth introduced the concept of "literate programming", a way of developing that mixes code, explanation, and outputs together in a way that's meant to be more interpretable by humans.



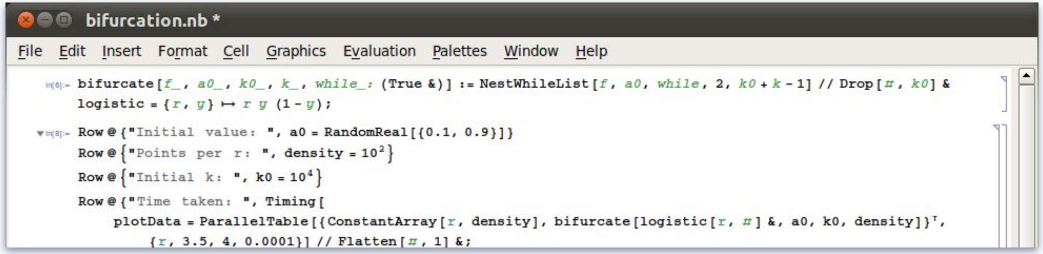
```
@ Here is a Perl program that simply
prints out |Hello, world!| the number of
times specified in the first argument.

<<*>>=
#!/usr/bin/perl
        <<CheckArgs>>
        <<PrintHiWorld>>

@ Printing involves a simple loop.  Line
breaks are added for clarity.

<<PrintHiWorld>>=
for ($i = 0; $i < $ARGV[0]; $i++) {
        print "Hello, world!\n";
}

@ We \emph{must} make sure, however,
that an argument was specified.

<<CheckArgs>>=
if (@ARGV != 1) {
        die "No argument specified";
}
```

(a) Literate source.

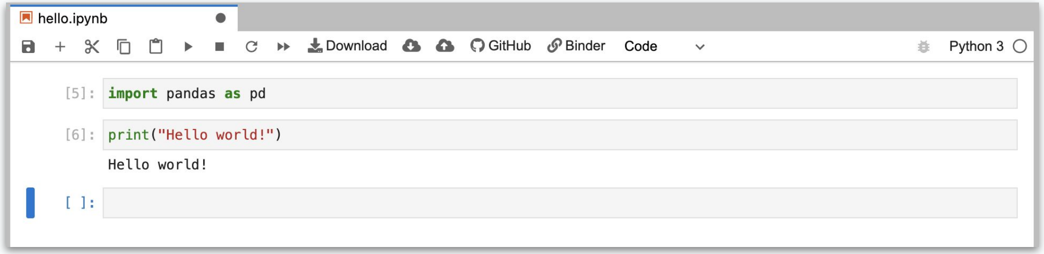
(b) Linear and hierarchical views.

Fast forward to 2022

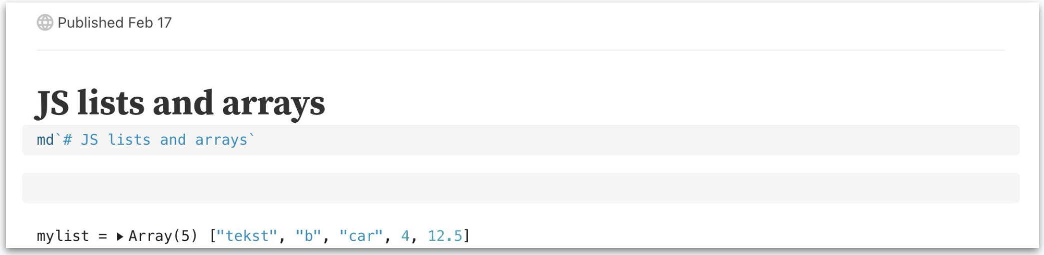
Notebooks are the most widely-used example of literate programming in practice.


bifurcation.nb *
File Edit Insert Format Cell Graphics Evaluation Palettes Window Help

```
In[6]:= bifurcate[f_, a0_, k0_, k_, while_: (True &)] := NestWhileList[f, a0, while, 2, k0 + k - 1] // Drop[#, k0] &
        logistic = {r, y} ↦ r y (1 - y);

In[8]:= Row @ {"Initial value: ", a0 = RandomReal[{0.1, 0.9}]}
        Row @ {"Points per r: ", density = 10^2}
        Row @ {"Initial k: ", k0 = 10^4}
        Row @ {"Time taken: ", Timing[
            plotData = ParallelTable[{ConstantArray[r, density], bifurcate[logistic[r, #] &, a0, k0, density]}^T,
                {r, 3.5, 4, 0.0001}] // Flatten[#, 1] &;
```



```
[5]: import pandas as pd

[6]: print("Hello world!")

     Hello world!

[ ]:
```



JS lists and arrays

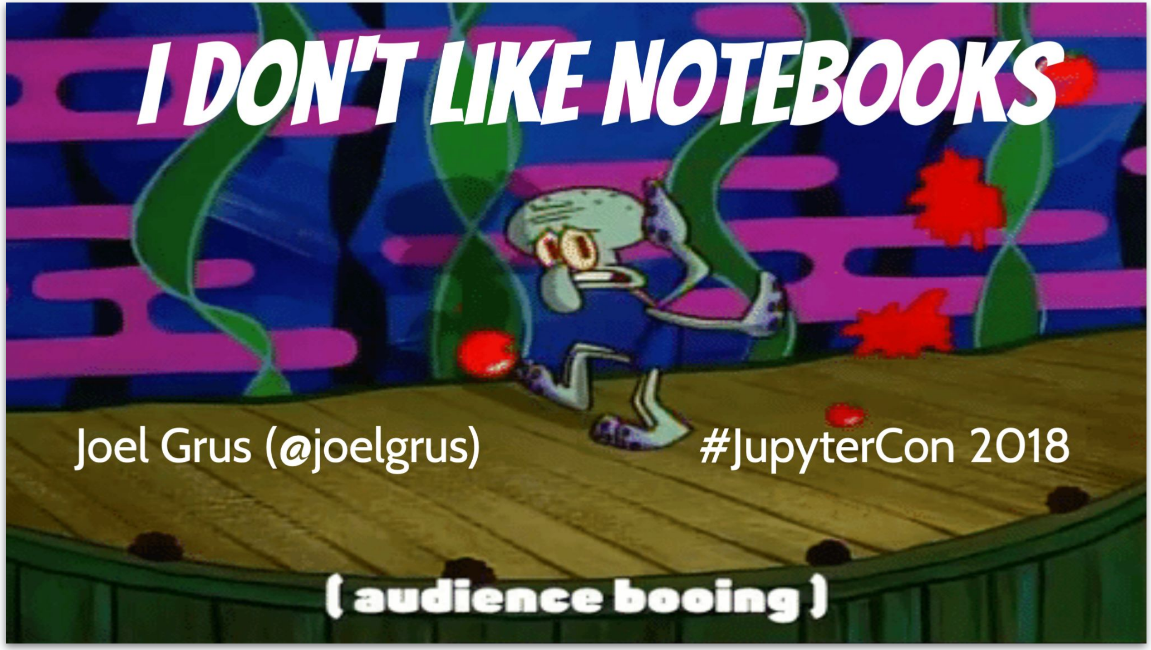
```
md`# JS lists and arrays`
```

```
mylist = ▸Array(5) ["tekst", "b", "car", 4, 12.5]
```

Why notebooks?

- Mix code and outputs together
- Great for iterating on smaller chunks of code; well-suited to exploration
- Linear, narrative layout that is great for storytelling

But notebooks have... issues



The State Problem

```
a = 1
```

```
a = 2
```

```
print(a)
```

What does this print?

imperative programming

a programming paradigm that uses statements that change a program's state.

Notebook state causes 3 major problems

1. **Interpretability**

   It’s hard to reason about what’s happening in a notebook, especially someone else’s.

2. **Reproducibility**

   Out of order cells make it hard to reproduce work without frequent restart-and-run-alls.

3. **Performance**

   Re-runs are wasteful and time-consuming… especially in Hex :(

Another barrier to entry

This is exactly the kind of thing that scares people off from analytics and data science, and gives code a bad name.

The state of state

NOTEBOOKS ARE GREAT FOR ITERATIVE DEVELOPMENT
NOTEBOOKS ARE DANGEROUS UNLESS YOU RUN EACH CELL EXACTLY ONCE, IN ORDER
imgflip.com
JAKE-CLARK.TUMBLR

DATA DEPARTMENT
HAVE YOU TRIED RESTARTING AND RUNNING FROM SCRATCH
imgflip.com

Re-thinking state

reactive programming

a programming paradigm oriented around data flows and the propagation of change.

In practice, this means that reactive objects maintain references to their dependencies and update automatically when their dependencies change.

Why reactive programming?

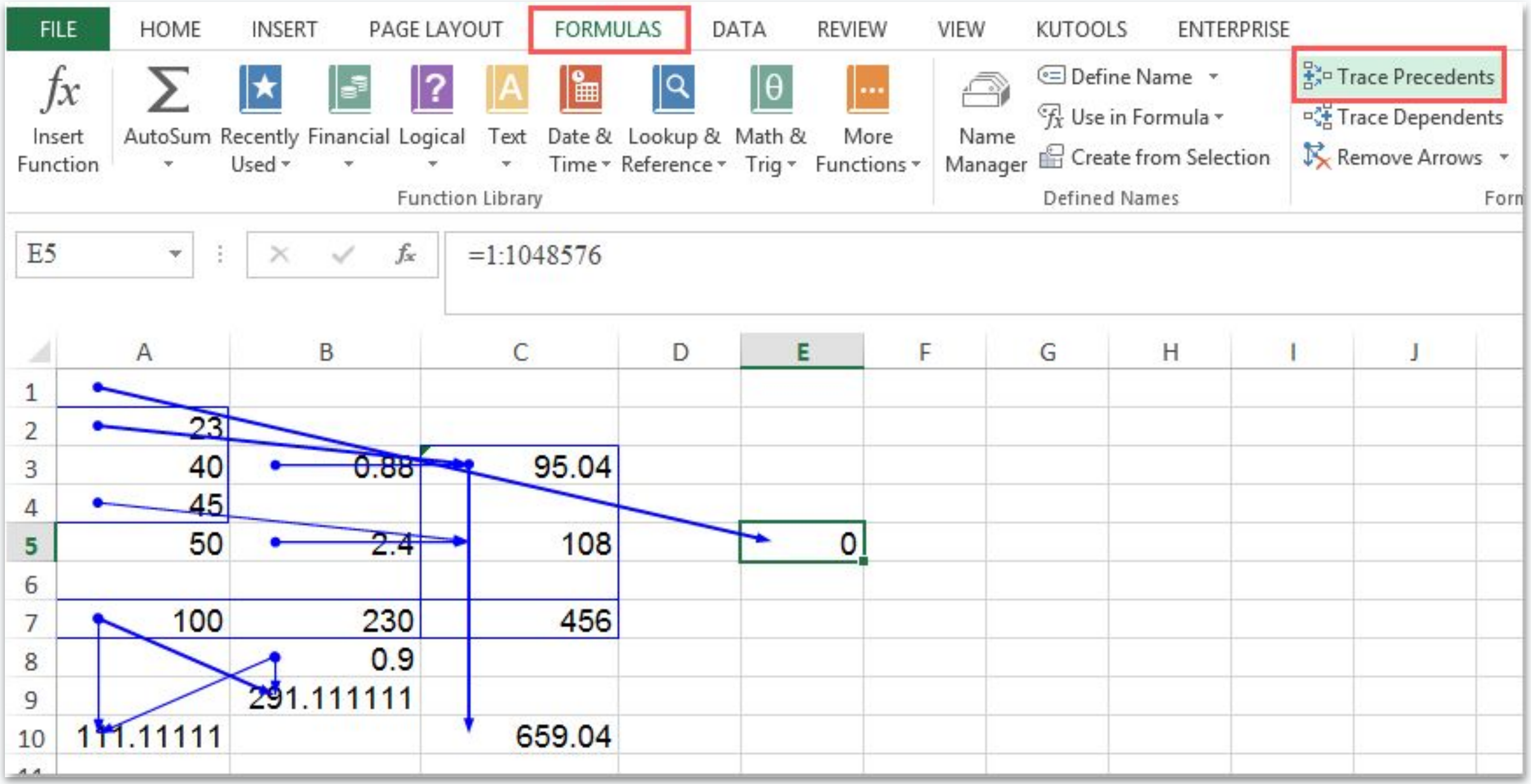
- State consistency
- Performance
- Nice abstractions for async and concurrent data flows

Imperative

```
>> a = 4
>> b = 10
>> c = a + b
>> c
14
>> a = 25
>> c
14
```

Reactive

```
>> a = 4
>> b = 10
>> c = a + b
>> c
14
>> a = 25
>> c
35
```

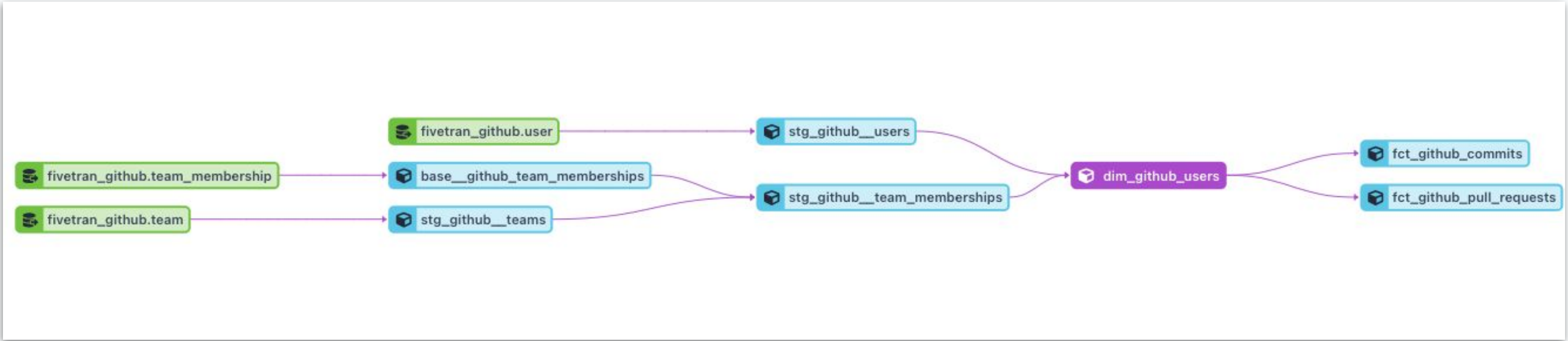


E5 =1:1048576

	A	B	C	D	E	F	G	H	I	J
1										
2	23									
3	40	0.88	95.04							
4	45									
5	50	2.4	108		0					
6										
7	100	230	456							
8		0.9								
9		291.111111								
10	111.11111		659.04							

Everyone's favorite reactive programming tool

DAGs!

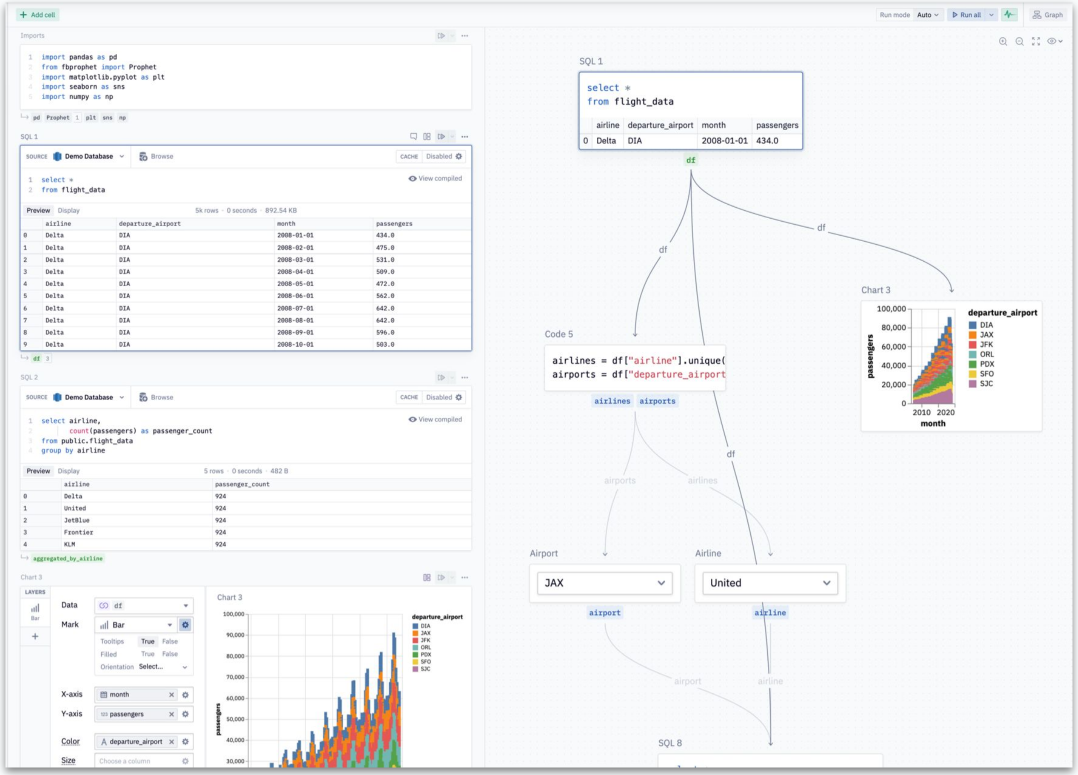


a DAG in dbt

Bringing reactivity and DAGs to notebooks

We introduced a **fully-reactive, DAG-based execution model** in Hex 2.0, which solves for all 3 problems we discussed earlier:

- Interpretability
- Reproducibility
- Performance



```
import pandas as pd
from fbprophet import Prophet
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

```
select *
from flight_data
```

5k rows · 0 seconds · 892.54 KB

	airline	departure_airport	month	passengers
0	Delta	DIA	2008-01-01	434.0
1	Delta	DIA	2008-02-01	475.0
2	Delta	DIA	2008-03-01	531.0
3	Delta	DIA	2008-04-01	509.0
4	Delta	DIA	2008-05-01	472.0
5	Delta	DIA	2008-06-01	562.0
6	Delta	DIA	2008-07-01	642.0
7	Delta	DIA	2008-08-01	642.0
8	Delta	DIA	2008-09-01	596.0
9	Delta	DIA	2008-10-01	503.0

```
select airline,
    count(passengers) as passenger_count
from public.flight_data
group by airline
```

5 rows · 0 seconds · 482 B

	airline	passenger_count
0	Delta	924
1	United	924
2	JetBlue	924
3	Frontier	924
4	KLM	924

Data: df
Mark: Bar
Tooltips: [✓ True] False
Filled: True False
Orientation: Select...
X-axis: month
Y-axis: passengers
Color: departure_airport
Size: Choose a column

Chart 3

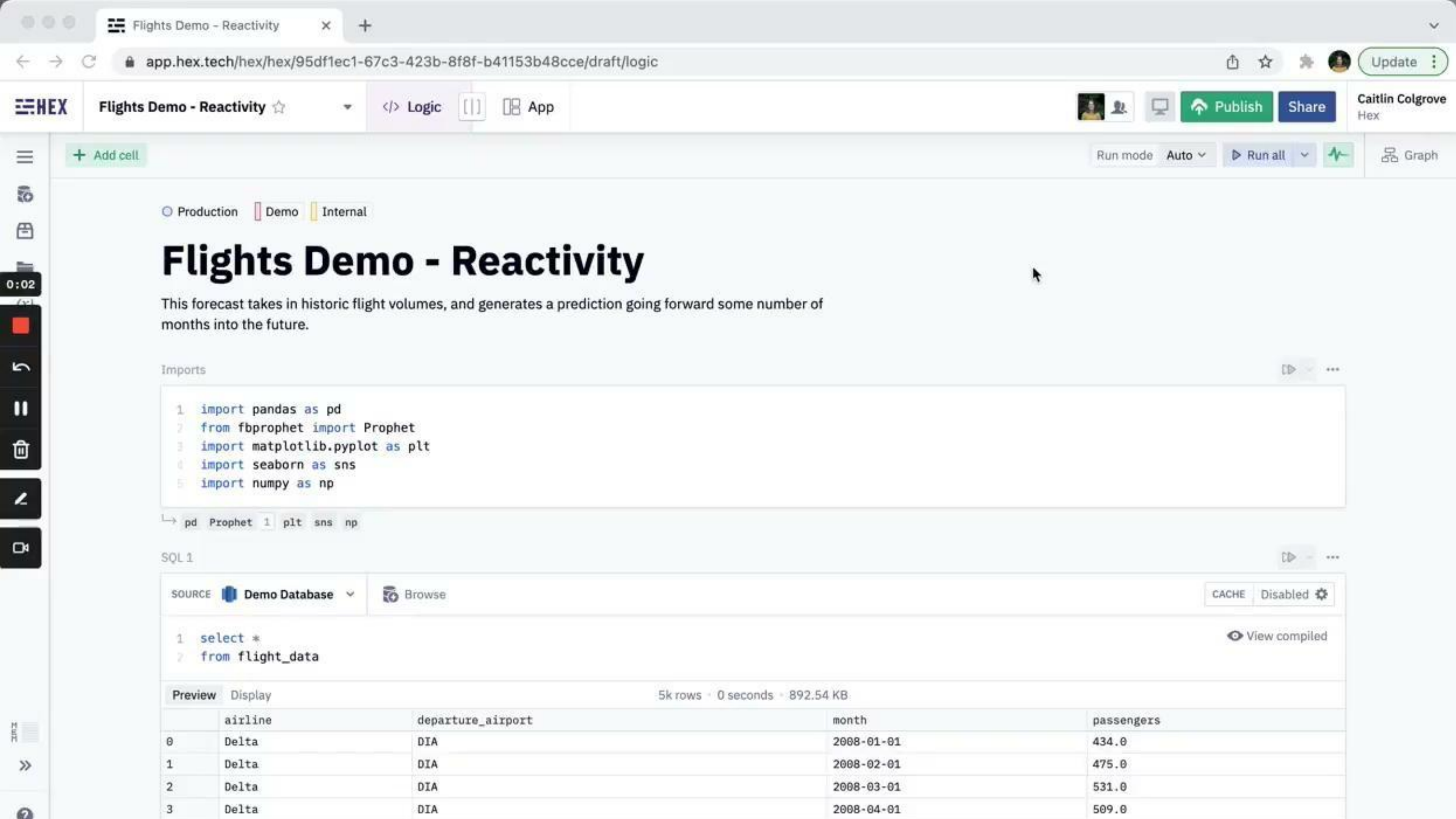
departure_airport
🟦 DIA
🟧 JAX
🟥 JFK
🟦 ORL
🟩 PDX
🟨 SFO
🟪 SJC

	airline	departure_airport	month	passengers
0	Delta	DIA	2008-01-01	434.0

```
airlines = df["airline"].unique(
airports = df["departure_airport
```

Airport: JAX
Airline: United

Demo


Flights Demo - Reactivity
app.hex.tech/hex/hex/95df1ec1-67c3-423b-8f8f-b41153b48cce/draft/logic
Update
HEX
Flights Demo - Reactivity
Logic
App
Publish
Share
Caitlin Colgrove
Hex
Add cell
Run mode Auto
Run all
Graph
0:02
Production
Demo
Internal

Flights Demo - Reactivity

This forecast takes in historic flight volumes, and generates a prediction going forward some number of months into the future.

Imports

```
import pandas as pd
from fbprophet import Prophet
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

pd Prophet 1 plt sns np

SQL 1

SOURCE Demo Database Browse CACHE Disabled

```
select *
from flight_data
```

View compiled

Preview Display 5k rows · 0 seconds · 892.54 KB

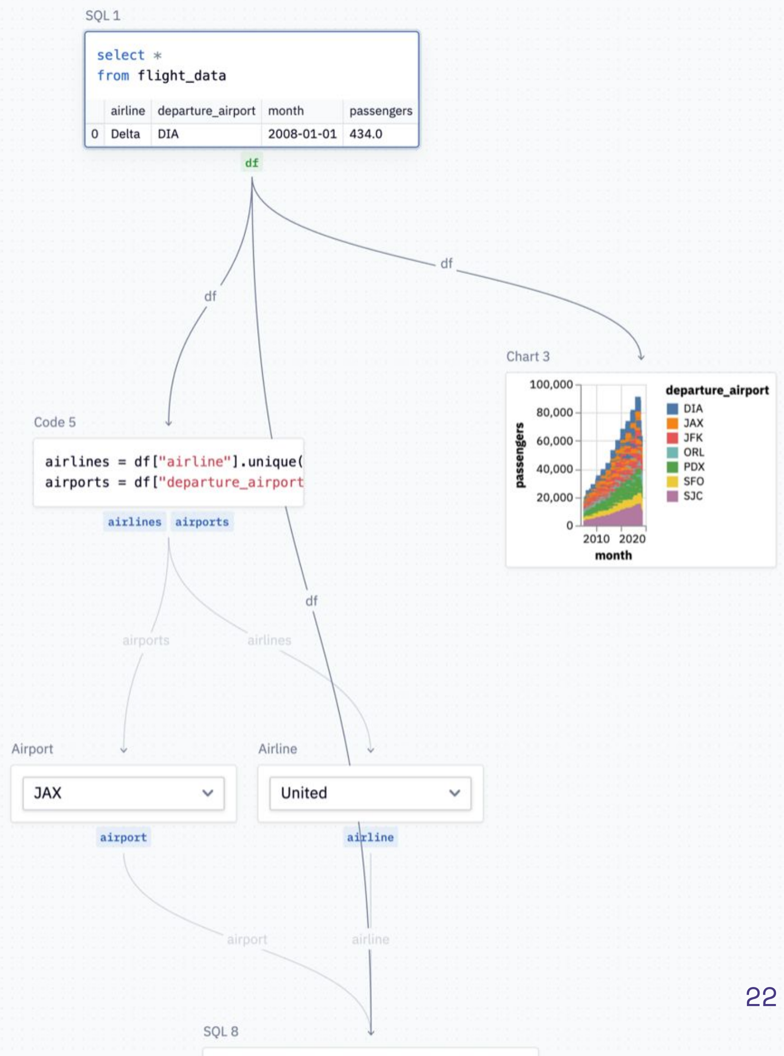
	airline	departure_airport	month	passengers
0	Delta	DIA	2008-01-01	434.0
1	Delta	DIA	2008-02-01	475.0
2	Delta	DIA	2008-03-01	531.0
3	Delta	DIA	2008-04-01	509.0

Under the hood: building the DAGs

Graphs have **Nodes** and **Edges**:

- Nodes = Cells
- In edges: Variable references
- Out edges: Variable assignments

How do we determine relationships?

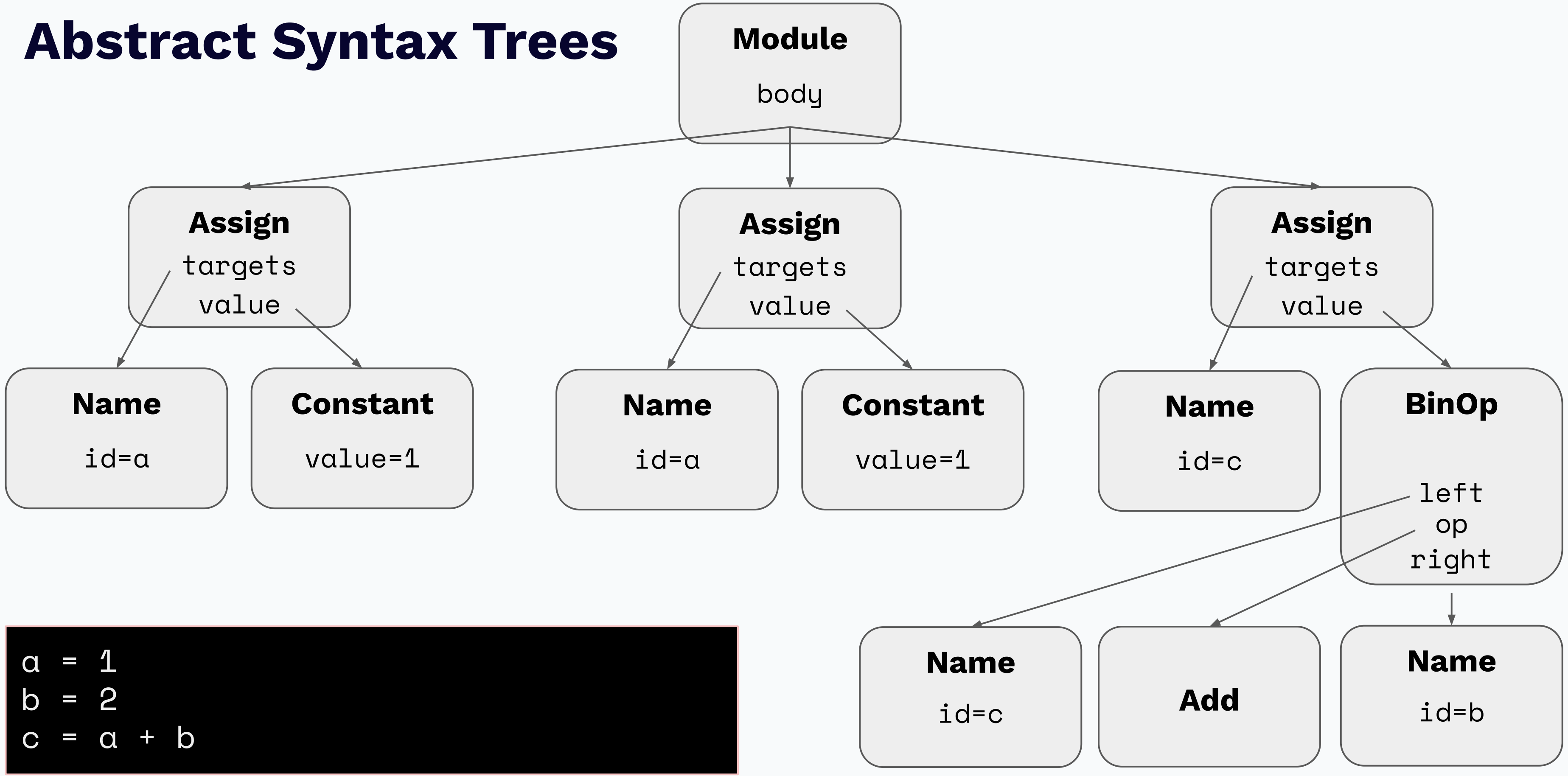


```
select *
from flight_data
```

	airline	departure_airport	month	passengers
0	Delta	DIA	2008-01-01	434.0

```
airlines = df["airline"].unique(
airports = df["departure_airport
```

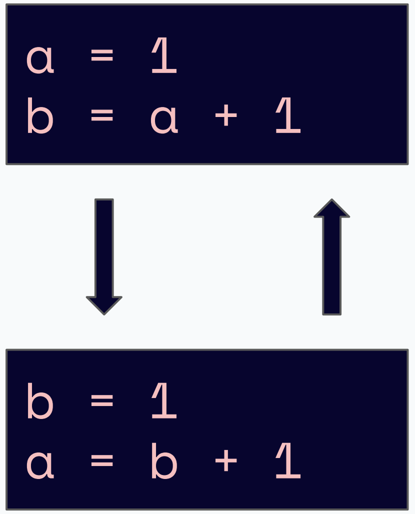
Abstract Syntax Trees



```
a = 1
b = 2
c = a + b
```

Issues with this approach

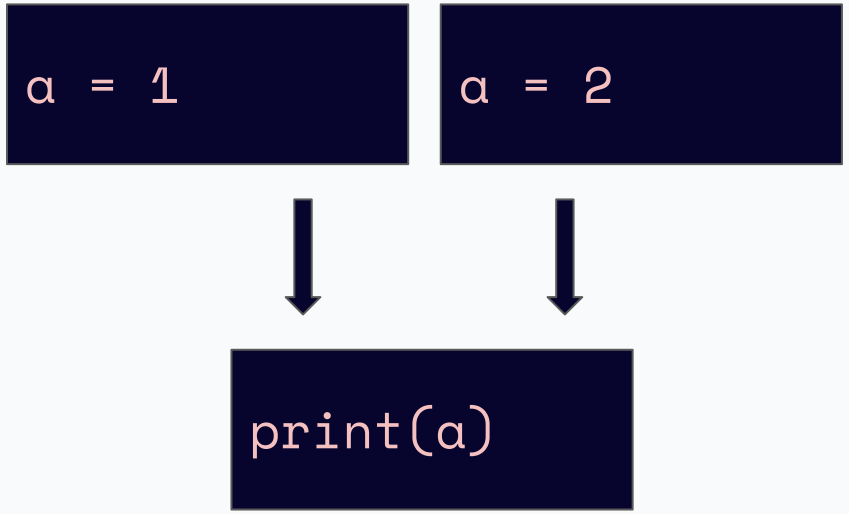
It’s not actually a DAG!



```
a = 1
b = a + 1
```

```
b = 1
a = b + 1
```

The ordering is non-deterministic

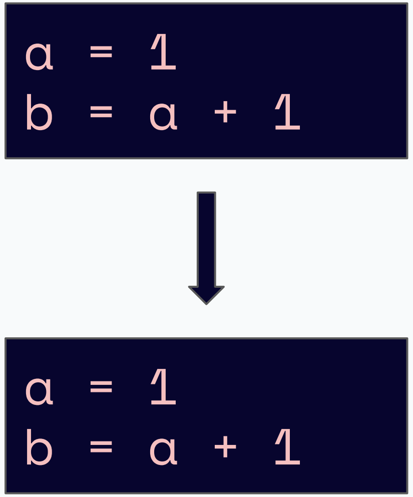


```
a = 1
```

```
a = 2
```

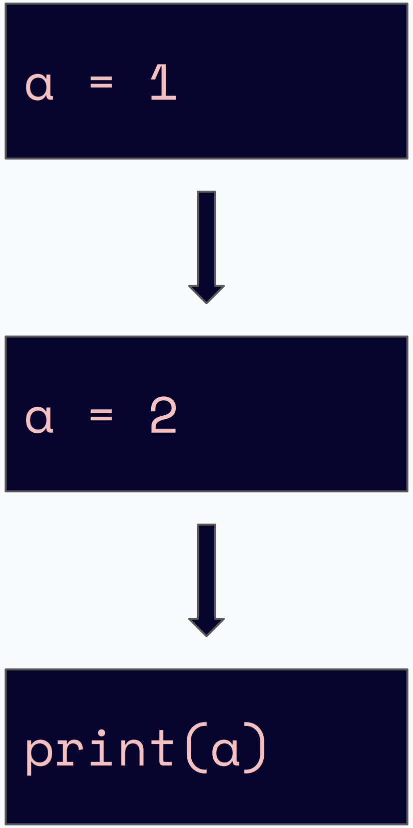
```
print(a)
```

Solution: use notebook ordering



```
a = 1
b = a + 1
```

```
a = 1
b = a + 1
```



```
a = 1
```

```
a = 2
```

```
print(a)
```

Pulling it all together: bringing DAGs into Hex notebooks

Determining “staleness”

In order to know which cells to recompute, we track a condition called *staleness*.

A cell is *stale* if:

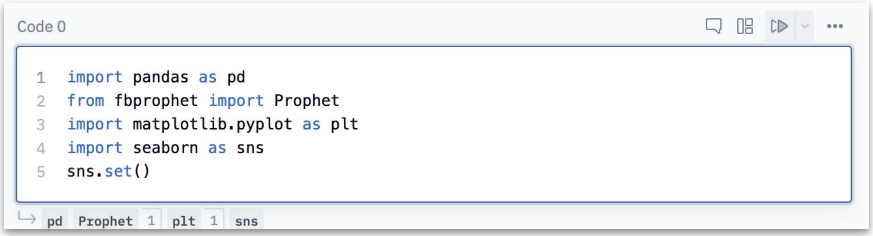
- It hasn’t been run yet this kernel session
- An upstream cell has been **edited** and it hasn’t been re-run
- An upstream cell has been **run** and it hasn’t been re-run
- An upstream cell has **become stale**

Implementing Reactivity with iPython

On each edit:

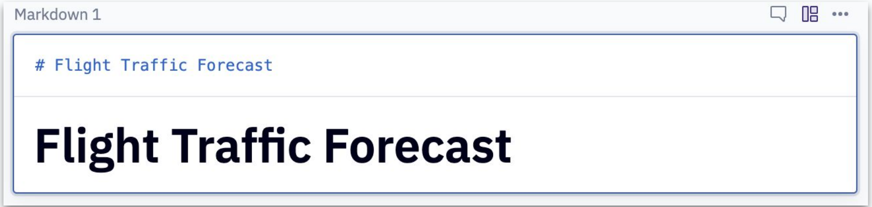
- Run each cell through an AST parser to compute inputs and outputs
- Re-compute the cell DAG
- Traverse graph upstream **and** downstream to determine list of cells needed to be run
  - Upstream, filter out cells that are already “up to date”
  - Downstream, mark as “stale”
- Queue all remaining stale cells in notebook order into the kernel
  - Mark cell as “up to date” after successful run

DAG usability cleanup



Code 0

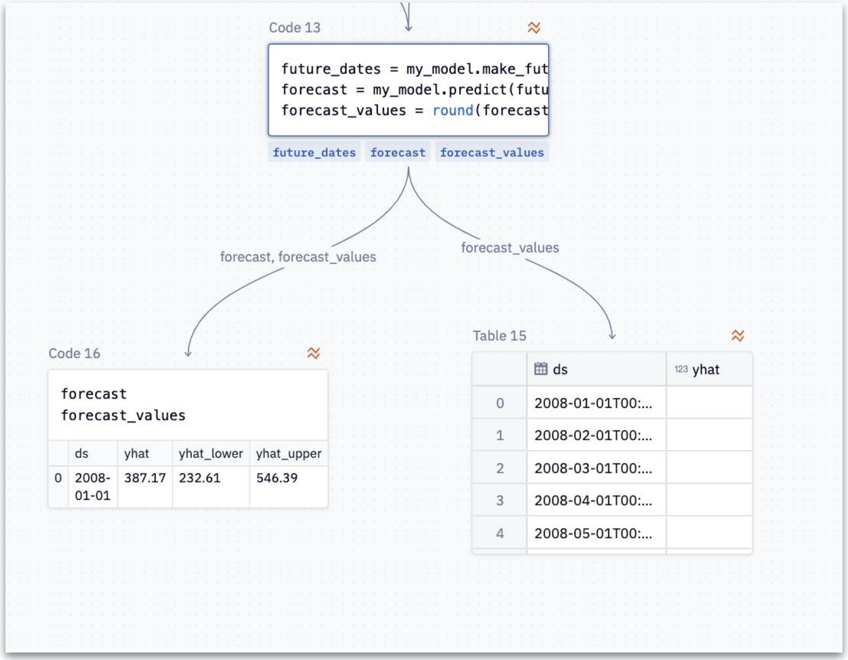
```python
import pandas as pd
from fbprophet import Prophet
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
```



Markdown 1

```
# Flight Traffic Forecast
```

Flight Traffic Forecast



Code 13

```python
future_dates = my_model.make_fut
forecast = my_model.predict(futu
forecast_values = round(forecast
```

Code 16

```
forecast
forecast_values
```

	ds	yhat	yhat_lower	yhat_upper
0	2008-01-01	387.17	232.61	546.39

Table 15

	ds	yhat
0	2008-01-01T00:...	
1	2008-02-01T00:...	
2	2008-03-01T00:...	
3	2008-04-01T00:...	
4	2008-05-01T00:...	

Future exploration

Future exploration

- Lambdas / better isolation
- Cell caching
- Performance & parallelism

Adam Storr
Design Lead

Melissa Carlson
Engineering Lead

Glen Takahashi
Chief Architect

Interested?

Director, Platform Engineering
Backend Engineer
Cloud Engineer
Platform PM
Engineering Lead
... and many more

hex.tech/jobs

Questions?