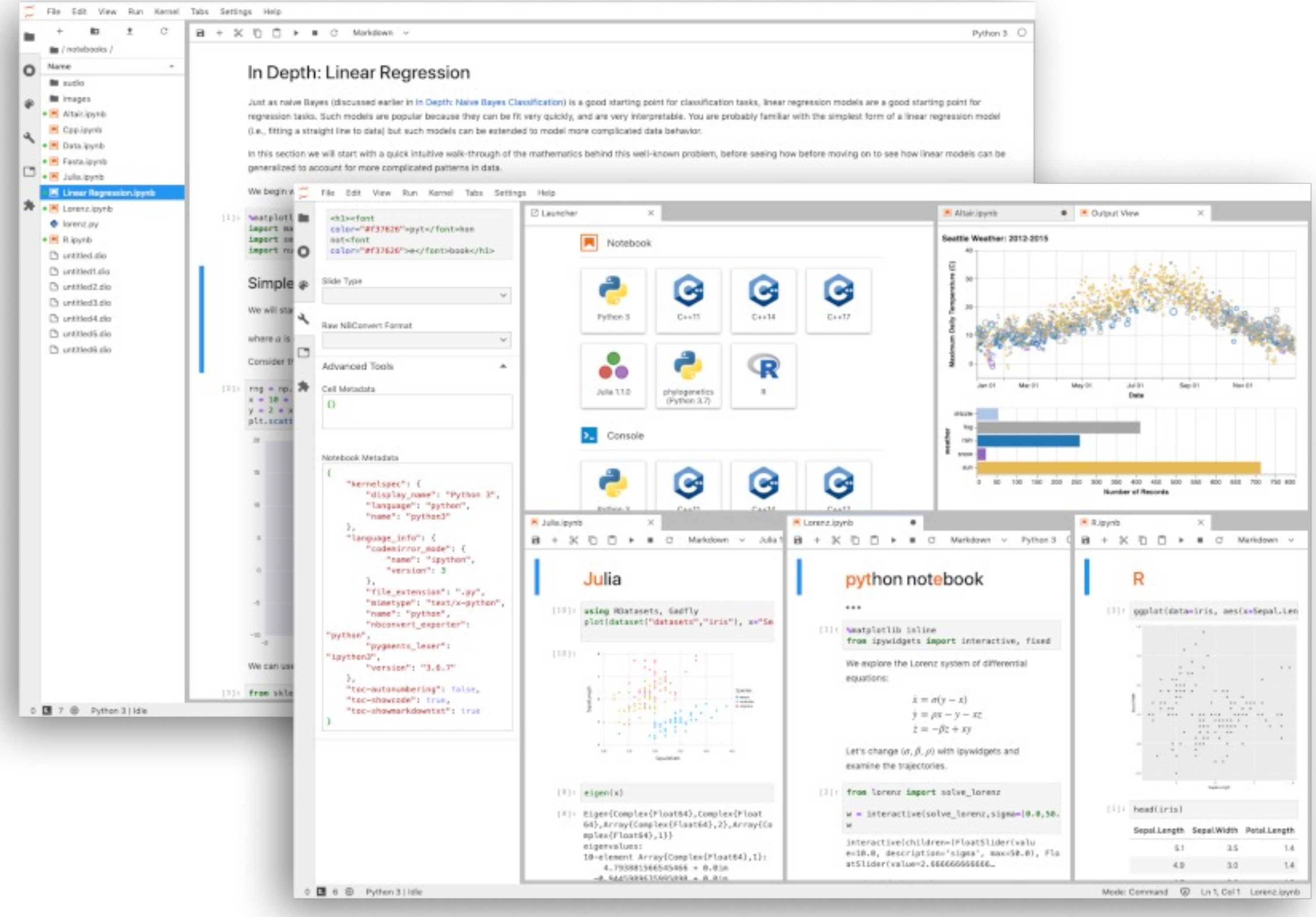
Productionizing Jupyter Notebooks

Antoni Ivanov

Versatile Data Kit Open Source Project

VERSATILE
DATA KIT

The role of Jupyter in the data world



In Depth: Linear Regression

Just as naive Bayes (discussed earlier in In Depth: Naive Bayes Classification) is a good starting point for classification tasks, linear regression models are a good starting point for regression tasks. Such models are popular because they can be fit very quickly, and are very interpretable. You are probably familiar with the simplest form of a linear regression model (i.e., fitting a straight line to data) but such models can be extended to model more complicated data behavior.

In this section we will start with a quick intuitive walk-through of the mathematics behind this well-known problem, before seeing how before moving on to see how linear models can be generalized to account for more complicated patterns in data.

```
<h1><font
color="#f37626">pyt</font>hon
not<font
color="#f37626">e</font>book</h1>
```

Slide Type

Raw NBConvert Format

Advanced Tools

Cell Metadata

```
{}
```

Notebook Metadata

```
{
    "kernelspec": {
        "display_name": "Python 3",
        "language": "python",
        "name": "python3"
    },
    "language_info": {
        "codemirror_mode": {
            "name": "ipython",
            "version": 3
        },
        "file_extension": ".py",
        "mimetype": "text/x-python",
        "name": "python",
        "nbconvert_exporter":
"python",
        "pygments_lexer":
"ipython3",
        "version": "3.6.7"
    },
    "toc-autonumbering": false,
    "toc-showcode": true,
    "toc-showmarkdowntxt": true
}
```

Seattle Weather: 2012-2015

Julia

```
[10]: using RDatasets, Gadfly
      plot(dataset("datasets","iris"), x="Se
```

```
[8]: eigen(x)
```

```
[8]: Eigen{Complex{Float64},Complex{Float
64},Array{Complex{Float64},2},Array{Co
mplex{Float64},1}}
eigenvalues:
10-element Array{Complex{Float64},1}:
    4.793881566545466 + 0.0im
   -0.9445989635995998 + 0.0im
```

python notebook

```
[1]: %matplotlib inline
     from ipywidgets import interactive, fixed
```

We explore the Lorenz system of differential equations:

$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = \rho x - y - xz$$
$$\dot{z} = -\beta z + xy$$

Let's change (σ, β, ρ) with ipywidgets and examine the trajectories.

```
[2]: from lorenz import solve_lorenz

     w = interactive(solve_lorenz,sigma=(0.0,50.
     w
```

```
interactive(children=(FloatSlider(valu
e=10.0, description='sigma', max=50.0), Flo
atSlider(value=2.6666666666666…
```

R

```
[3]: ggplot(data=iris, aes(x=Sepal.Len
```

```
[1]: head(iris)
```

Sepal.Length	Sepal.Width	Petal.Length
5.1	3.5	1.4
4.9	3.0	1.4

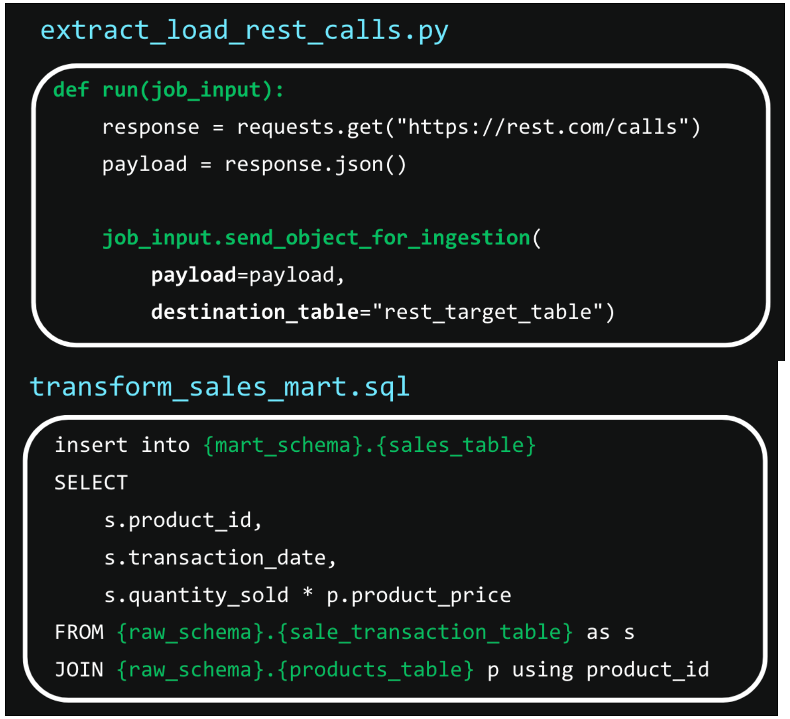
https://analyticsindiamag.com/why-jupyter-notebooks-are-so-popular-among-data-scientists/

https://odsc.medium.com/why-you-should-be-using-jupyter-notebooks-ea2e568c59f2

The role of Versatile Data Kit (VDK) in the data world

Develop

VDK SDK



extract_load_rest_calls.py

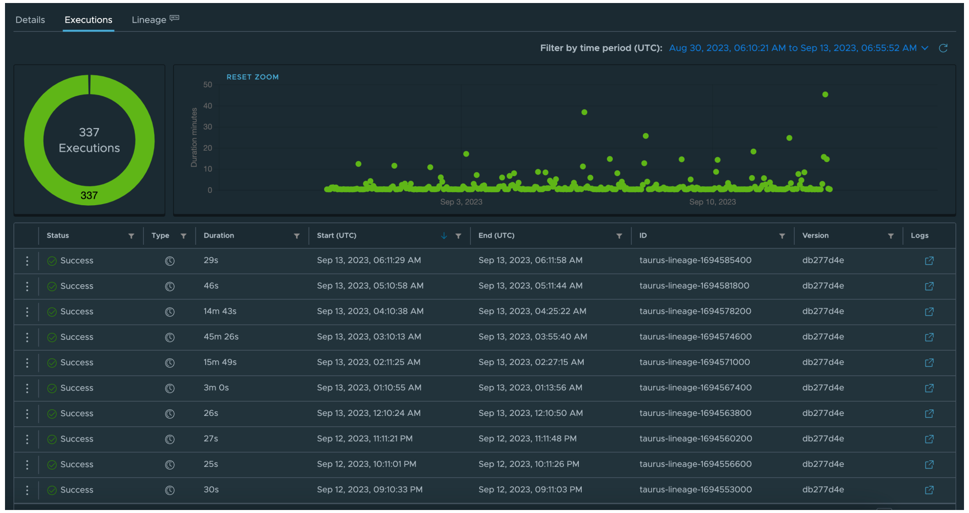
```python
def run(job_input):
    response = requests.get("https://rest.com/calls")
    payload = response.json()

    job_input.send_object_for_ingestion(
        payload=payload,
        destination_table="rest_target_table")
```

transform_sales_mart.sql

```sql
insert into {mart_schema}.{sales_table}
SELECT
    s.product_id,
    s.transaction_date,
    s.quantity_sold * p.product_price
FROM {raw_schema}.{sale_transaction_table} as s
JOIN {raw_schema}.{products_table} p using product_id
```

Deploy and Monitor

Control Plane and Operations UI



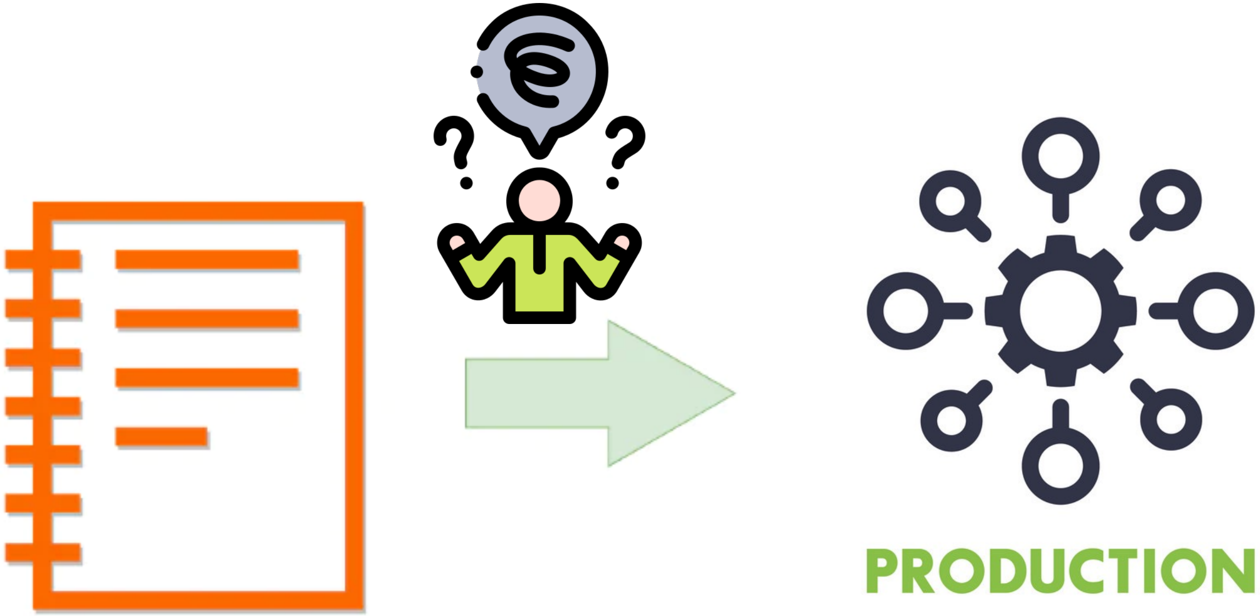
Filter by time period (UTC): Aug 30, 2023, 06:10:21 AM to Sep 13, 2023, 06:55:52 AM

Status	Type	Duration	Start (UTC)	End (UTC)	ID	Version	Logs
✓ Success		29s	Sep 13, 2023, 06:11:29 AM	Sep 13, 2023, 06:11:58 AM	taurus-lineage-1694585400	db277d4e	
✓ Success		46s	Sep 13, 2023, 05:10:58 AM	Sep 13, 2023, 05:11:44 AM	taurus-lineage-1694581800	db277d4e	
✓ Success		14m 43s	Sep 13, 2023, 04:10:38 AM	Sep 13, 2023, 04:25:22 AM	taurus-lineage-1694578200	db277d4e	
✓ Success		45m 26s	Sep 13, 2023, 03:10:13 AM	Sep 13, 2023, 03:55:40 AM	taurus-lineage-1694574600	db277d4e	
✓ Success		15m 49s	Sep 13, 2023, 02:11:25 AM	Sep 13, 2023, 02:27:15 AM	taurus-lineage-1694571000	db277d4e	
✓ Success		3m 0s	Sep 13, 2023, 01:10:55 AM	Sep 13, 2023, 01:13:56 AM	taurus-lineage-1694567400	db277d4e	
✓ Success		26s	Sep 13, 2023, 12:10:24 AM	Sep 13, 2023, 12:10:50 AM	taurus-lineage-1694563800	db277d4e	
✓ Success		27s	Sep 12, 2023, 11:11:21 PM	Sep 12, 2023, 11:11:48 PM	taurus-lineage-1694560200	db277d4e	
✓ Success		25s	Sep 12, 2023, 10:11:01 PM	Sep 12, 2023, 10:11:26 PM	taurus-lineage-1694556600	db277d4e	
✓ Success		30s	Sep 12, 2023, 09:10:33 PM	Sep 12, 2023, 09:11:03 PM	taurus-lineage-1694553000	db277d4e	

https://github.com/vmware/versatile-data-kit

Jupyter

From Data Exploration to Production

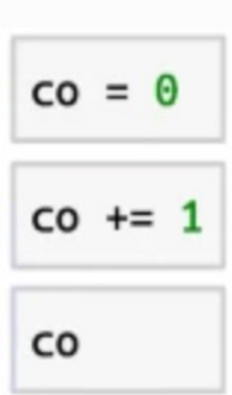


PRODUCTION

Challenges

- Reproducibility: Non-Linear Execution and Hidden State Risks
- Code Organization: Irrelevant or debugging code
- Execution model: interactive kernel vs automated flow
- Automated Testing and CICD
- Version Control

➢ Reproducibility: Non-Linear Execution and Hidden State Risks

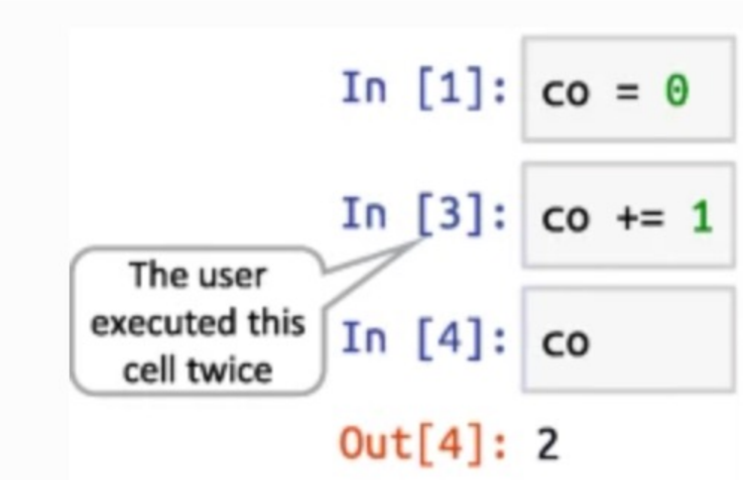


```
co = 0
```

```
co += 1
```

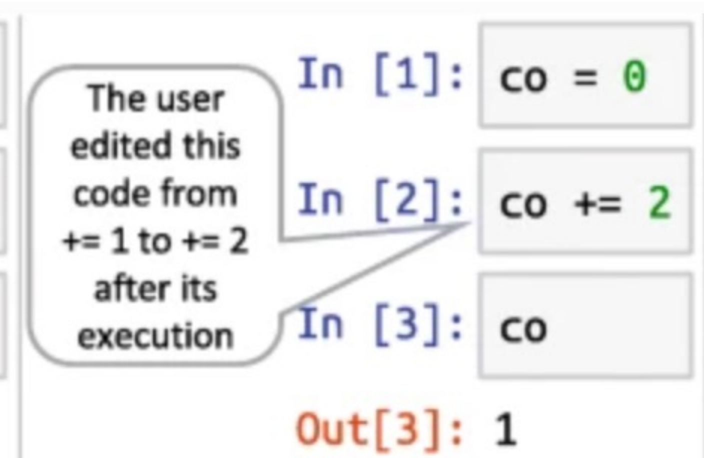
```
co
```

➢ Reproducibility: Non-Linear Execution and Hidden State Risks



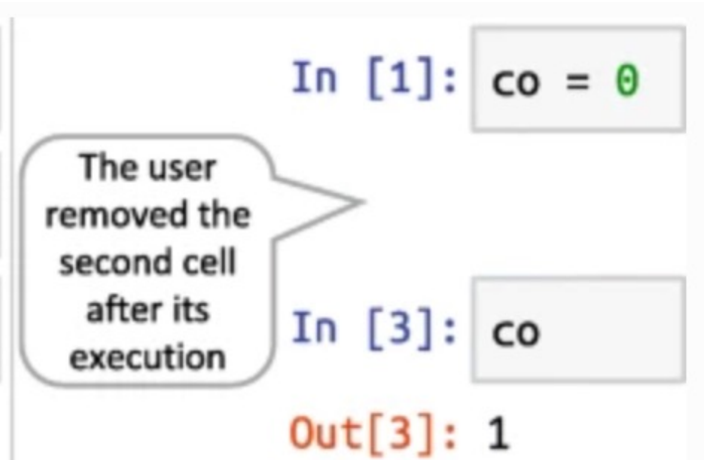
```
In [1]: co = 0

In [3]: co += 1

In [4]: co

Out[4]: 2
```

The user executed this cell twice



```
In [1]: co = 0

In [2]: co += 2

In [3]: co

Out[3]: 1
```

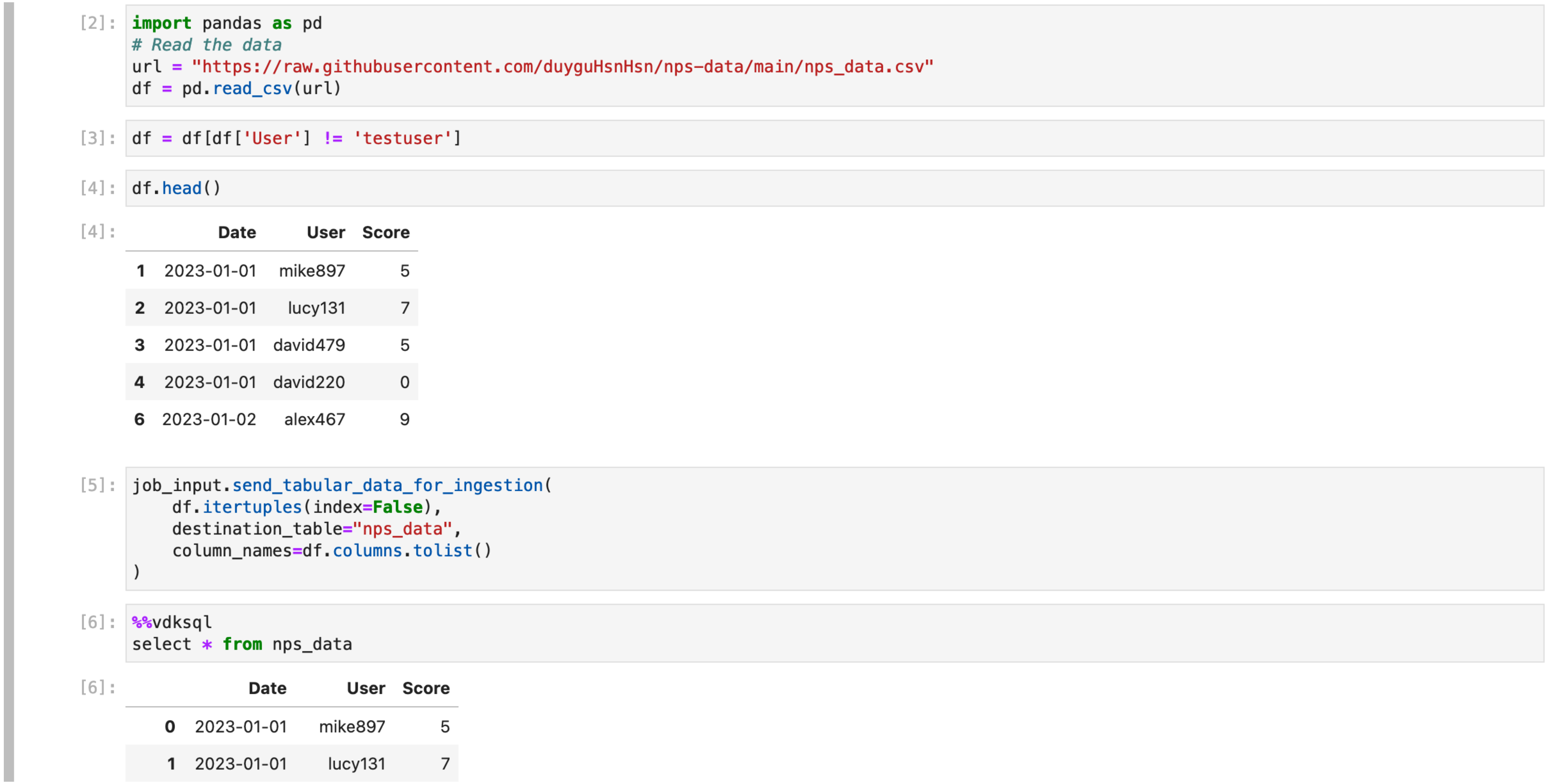
The user edited this code from += 1 to += 2 after its execution



```
In [1]: co = 0

In [3]: co

Out[3]: 1
```

The user removed the second cell after its execution

What can we do?



[2]:

```
import pandas as pd
# Read the data
url = "https://raw.githubusercontent.com/duyguHsnHsn/nps-data/main/nps_data.csv"
df = pd.read_csv(url)
```

[3]:

```
df = df[df['User'] != 'testuser']
```

[4]:

```
df.head()
```

[4]:

	Date	User	Score
1	2023-01-01	mike897	5
2	2023-01-01	lucy131	7
3	2023-01-01	david479	5
4	2023-01-01	david220	0
6	2023-01-02	alex467	9

[5]:

```
job_input.send_tabular_data_for_ingestion(
    df.itertuples(index=False),
    destination_table="nps_data",
    column_names=df.columns.tolist()
)
```

[6]:

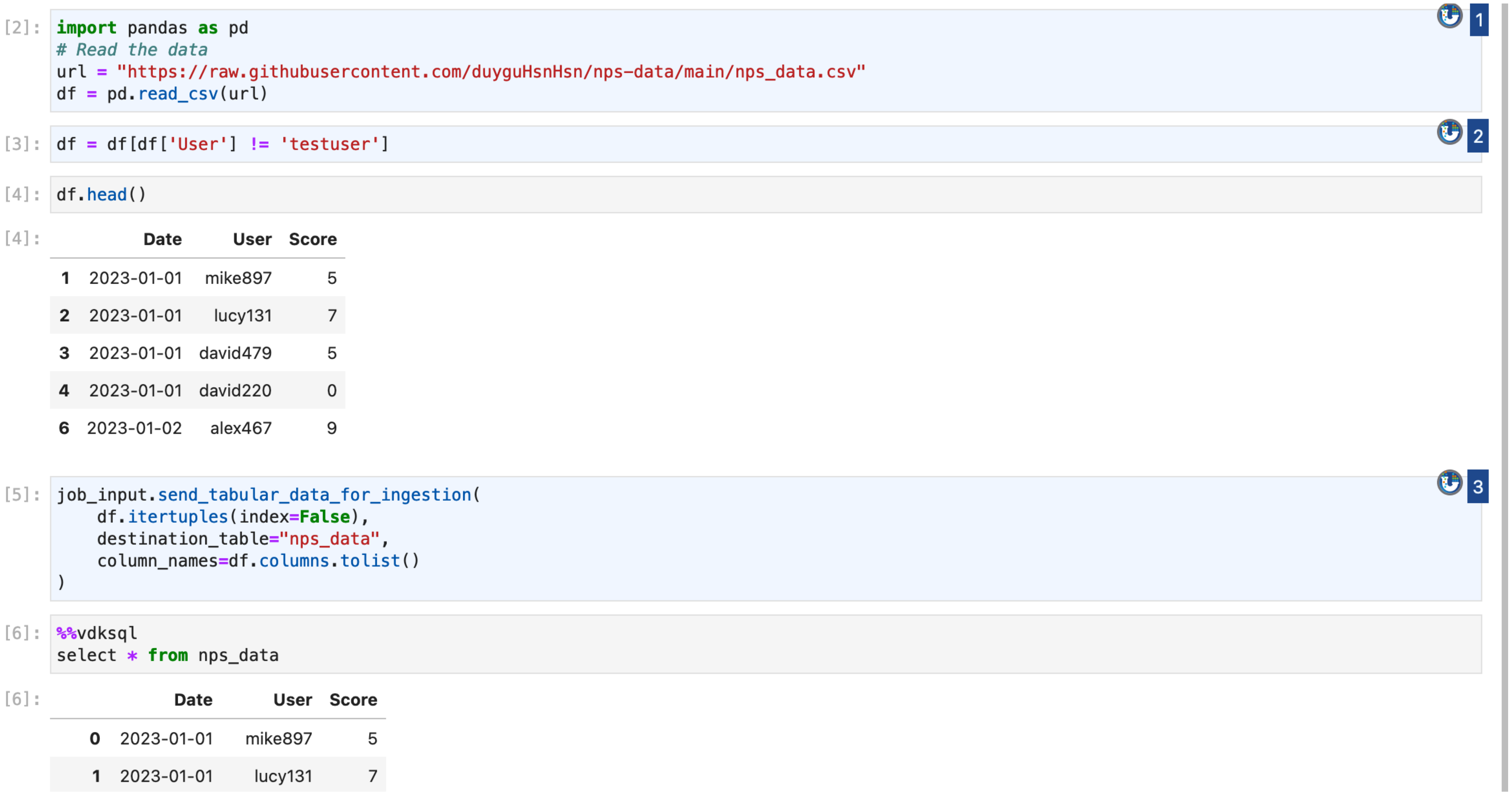
```
%%vdksql
select * from nps_data
```

[6]:

	Date	User	Score
0	2023-01-01	mike897	5
1	2023-01-01	lucy131	7

What can we do?

Tagging VDK Cells



[2]:
```
import pandas as pd
# Read the data
url = "https://raw.githubusercontent.com/duyguHsnHsn/nps-data/main/nps_data.csv"
df = pd.read_csv(url)
```

[3]:
```
df = df[df['User'] != 'testuser']
```

[4]:
```
df.head()
```

[4]:

	Date	User	Score
1	2023-01-01	mike897	5
2	2023-01-01	lucy131	7
3	2023-01-01	david479	5
4	2023-01-01	david220	0
6	2023-01-02	alex467	9

[5]:
```
job_input.send_tabular_data_for_ingestion(
    df.itertuples(index=False),
    destination_table="nps_data",
    column_names=df.columns.tolist()
)
```

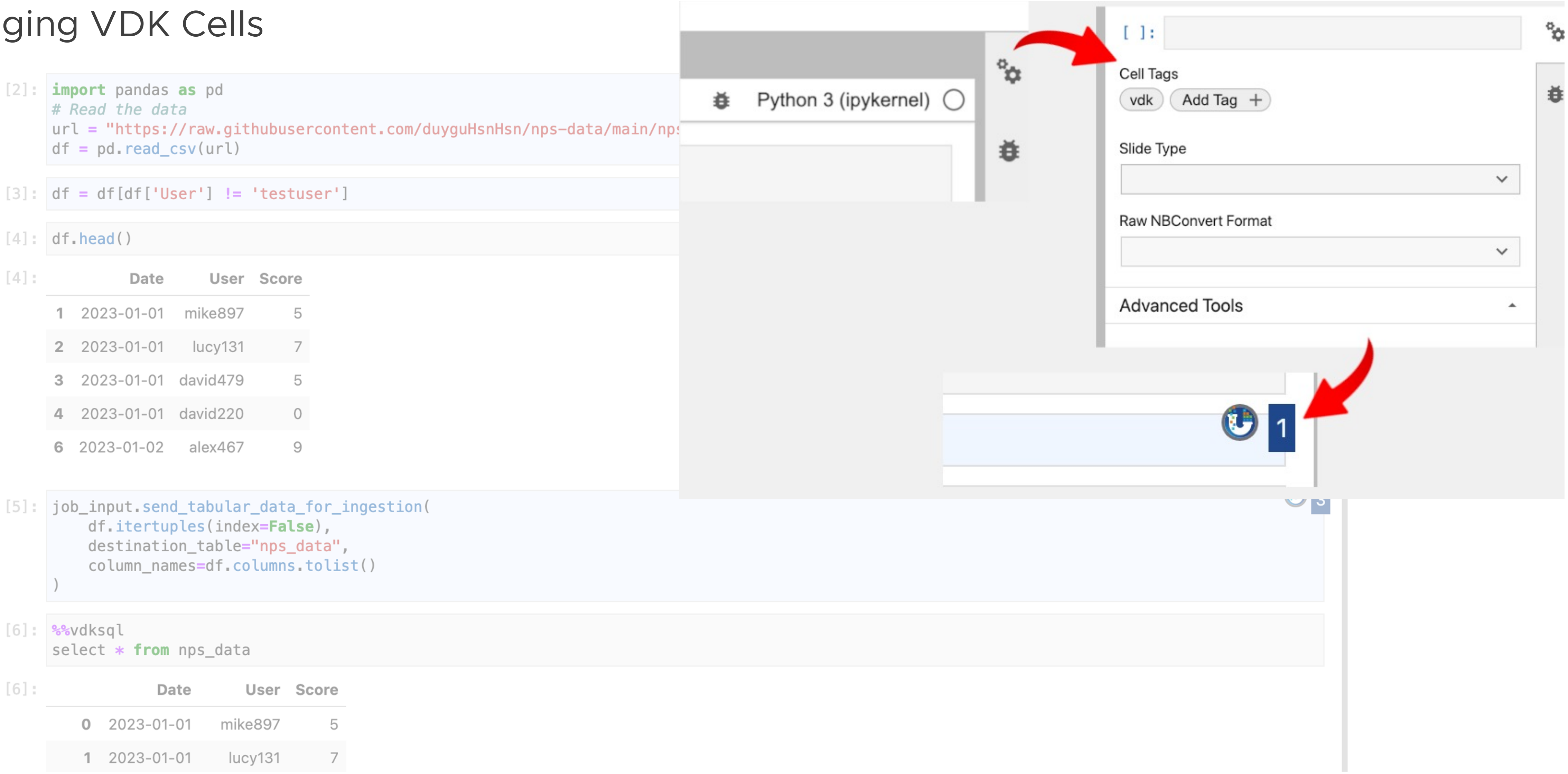
[6]:
```
%%vdksql
select * from nps_data
```

[6]:

	Date	User	Score
0	2023-01-01	mike897	5
1	2023-01-01	lucy131	7

What can we do?

Tagging VDK Cells



[2]:
```
import pandas as pd
# Read the data
url = "https://raw.githubusercontent.com/duyguHsnHsn/nps-data/main/nps
df = pd.read_csv(url)
```

[3]:
```
df = df[df['User'] != 'testuser']
```

[4]:
```
df.head()
```

[4]:

	Date	User	Score
1	2023-01-01	mike897	5
2	2023-01-01	lucy131	7
3	2023-01-01	david479	5
4	2023-01-01	david220	0
6	2023-01-02	alex467	9

[5]:
```
job_input.send_tabular_data_for_ingestion(
    df.itertuples(index=False),
    destination_table="nps_data",
    column_names=df.columns.tolist()
)
```

[6]:
```
%%vdksql
select * from nps_data
```

[6]:

	Date	User	Score
0	2023-01-01	mike897	5
1	2023-01-01	lucy131	7

Reproducibility: Non-Linear Execution and Hidden State Risks

- Assign a "vdk" tag and a specific number to a cell.
- The number dictates the order in which the cell will be executed in production.

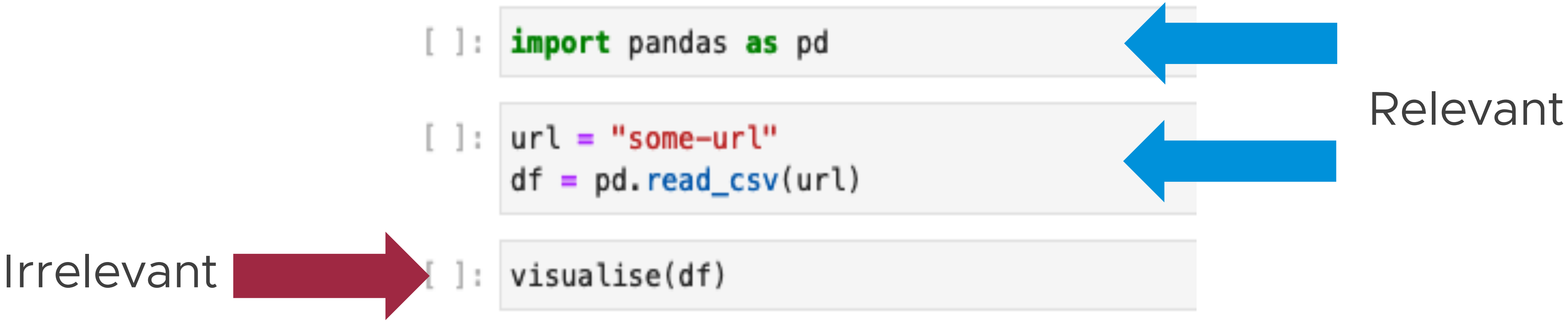
Benefits:

- Ensures only the tagged cells are executed, and in the determined sequence.
- Clearly defining the execution order.
- Detect when the current state is diverging from expected order.
- Test easily end-to-end before deployment (as we will see)

Challenges

- ✓ Reproducibility: Non-Linear Execution and Hidden State Risks
- ➢ Code Organization: Irrelevant or debugging code
- ➢ Execution model: interactive kernel vs automated flow
- ➢ Automated Testing and CICD
- ➢ Version Control

Code Organization: Irrelevant or debugging code



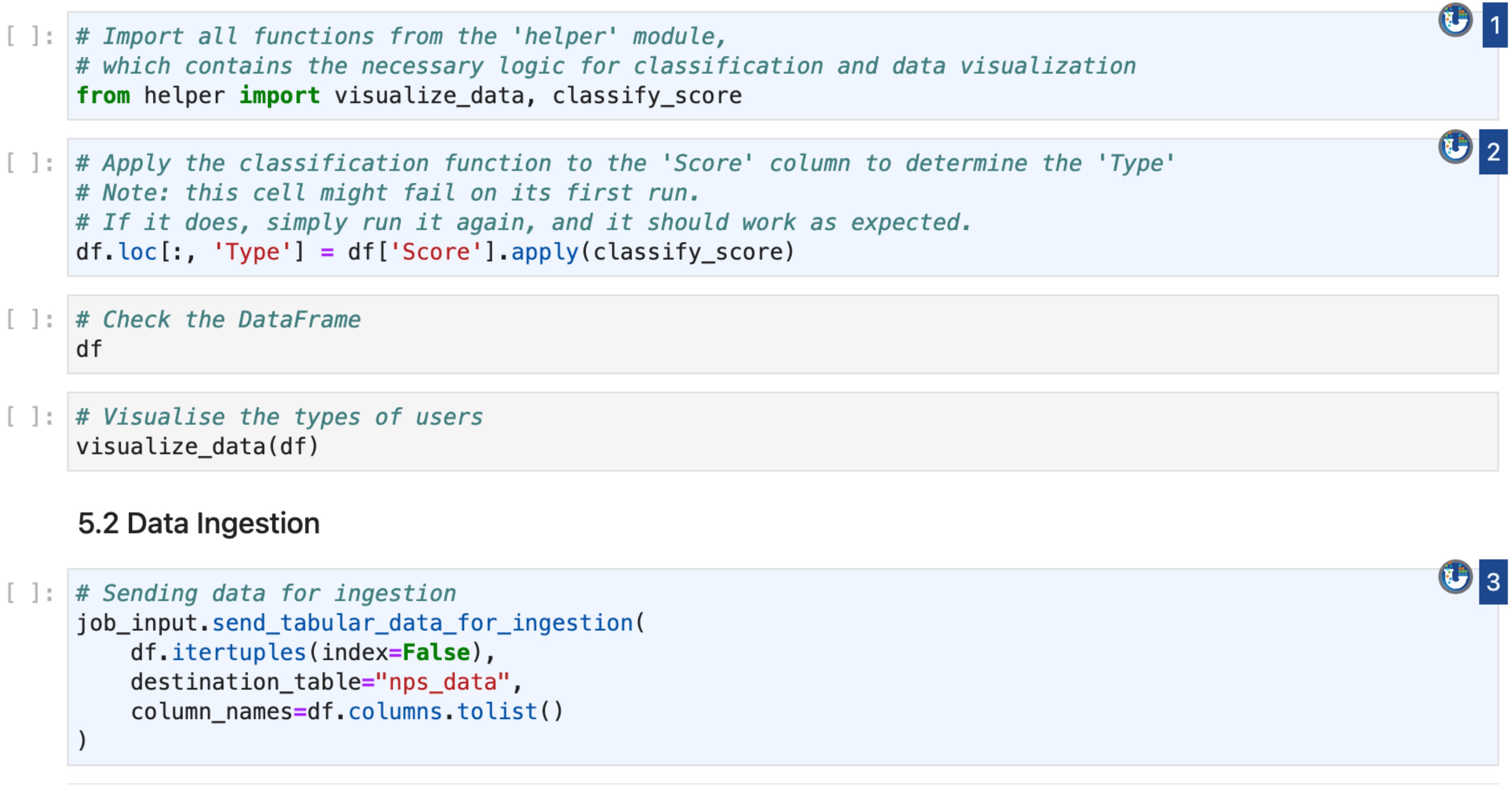
```
[ ]: import pandas as pd

[ ]: url = "some-url"
     df = pd.read_csv(url)

[ ]: visualise(df)
```

Relevant

Irrelevant

Code Organization: Irrelevant or debugging code

VDK tags to the rescue again



```
[ ]: # Import all functions from the 'helper' module,
     # which contains the necessary logic for classification and data visualization
     from helper import visualize_data, classify_score
```

```
[ ]: # Apply the classification function to the 'Score' column to determine the 'Type'
     # Note: this cell might fail on its first run.
     # If it does, simply run it again, and it should work as expected.
     df.loc[:, 'Type'] = df['Score'].apply(classify_score)
```

```
[ ]: # Check the DataFrame
     df
```

```
[ ]: # Visualise the types of users
     visualize_data(df)
```

5.2 Data Ingestion

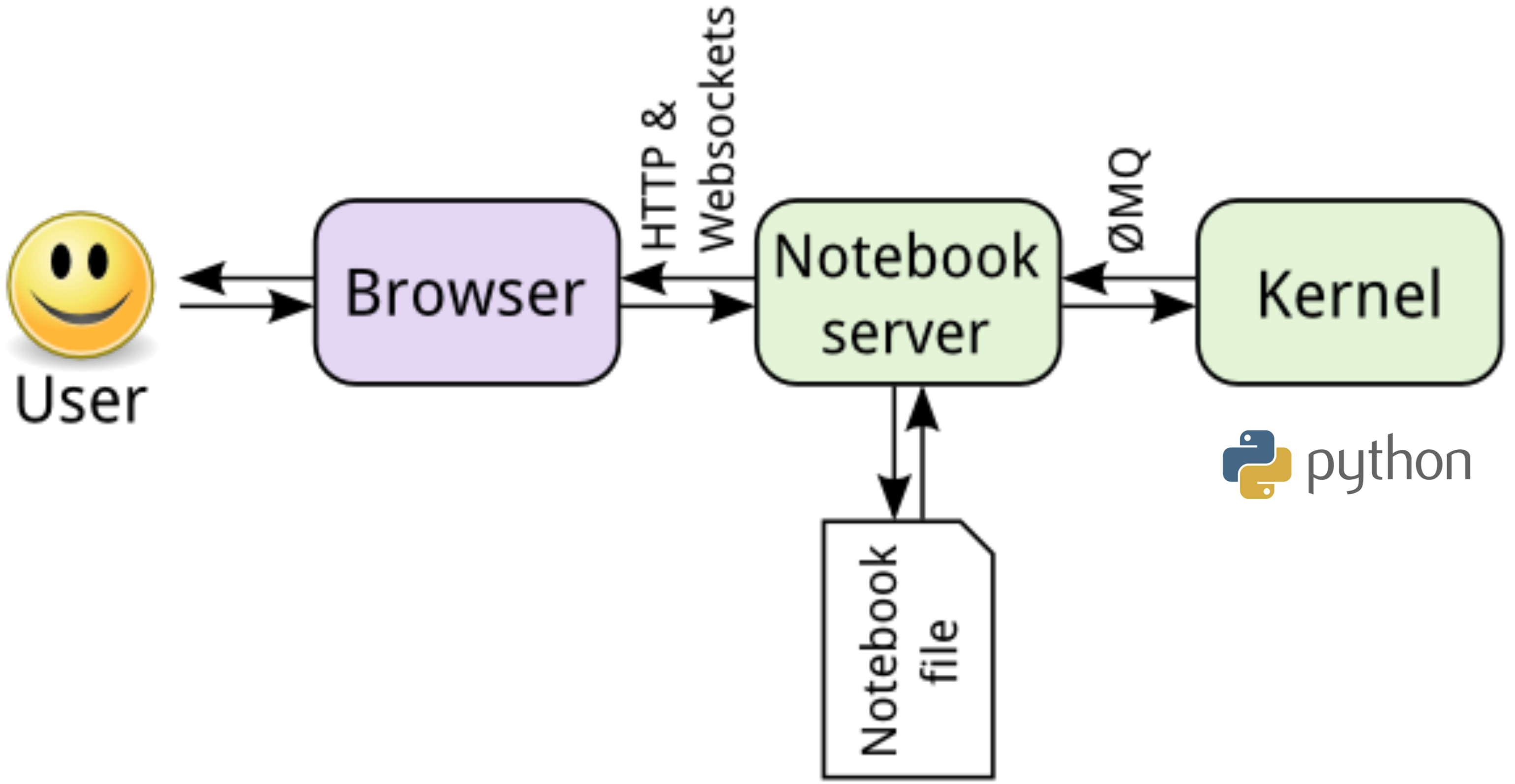
```
[ ]: # Sending data for ingestion
     job_input.send_tabular_data_for_ingestion(
         df.itertuples(index=False),
         destination_table="nps_data",
         column_names=df.columns.tolist()
     )
```

Challenges

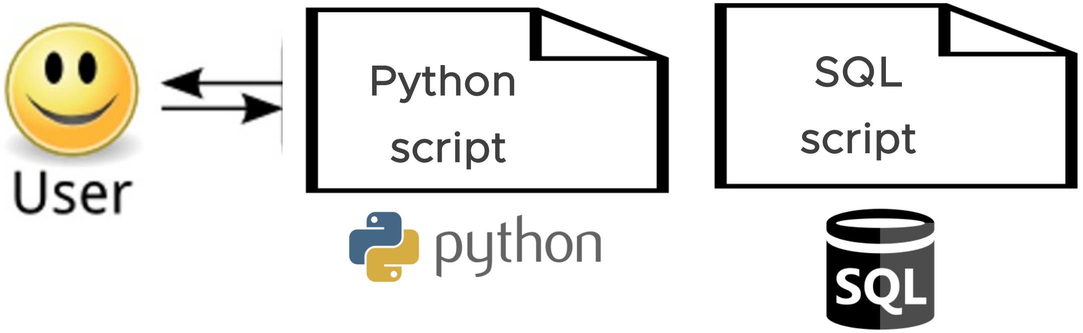
- ✓ Reproducibility: Non-Linear Execution and Hidden State Risks
- ✓ Code Organization: Irrelevant or debugging code
- ➢ Execution model: interactive kernel vs automated flow
- ➢ Automated Testing and CICD
- ➢ Version Control

Execution model: interactive kernel vs automated flow

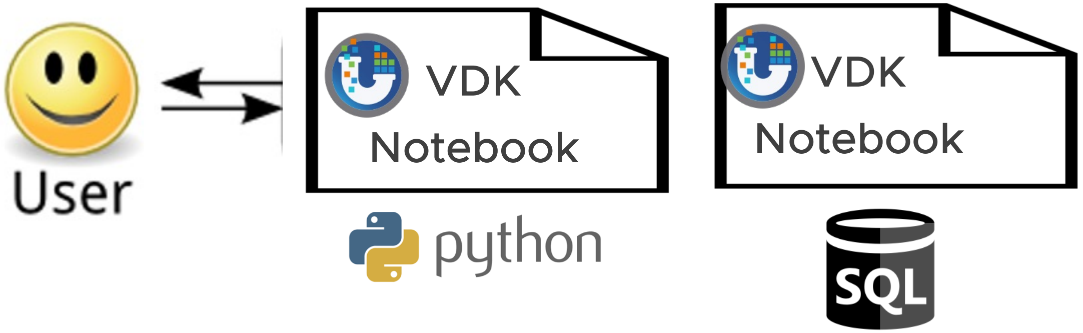
Bad for automation, bad for being part of a workfow



Execution model: interactive kernel vs automated flow

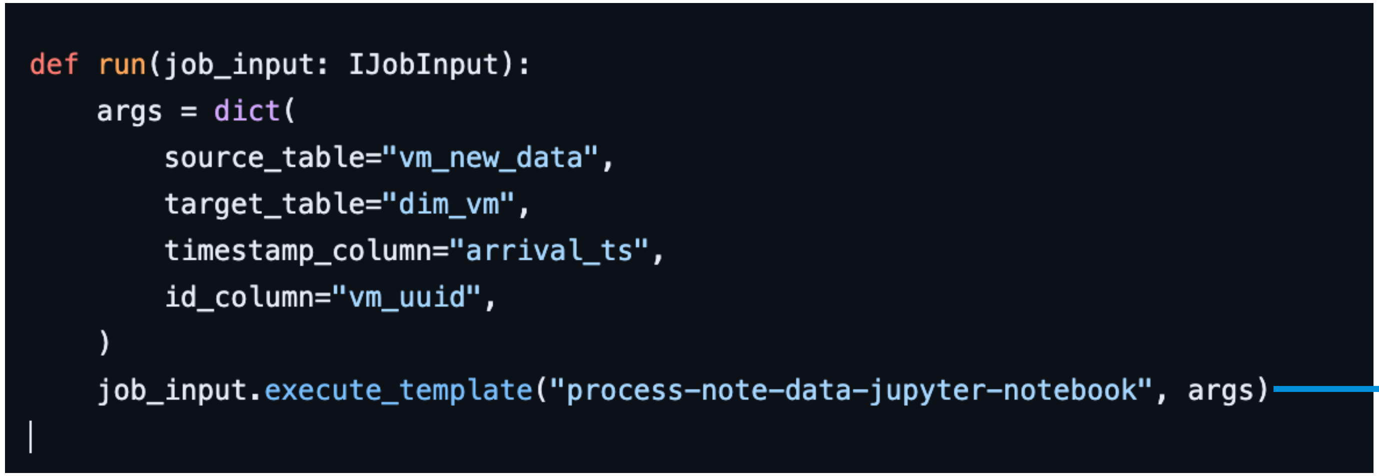


Execution model: interactive kernel vs automated flow

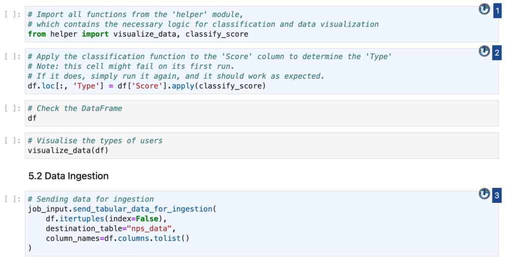


Execution model: interactive kernel vs automated flow

✓ Reuse another notebook as a template (function)



```
def run(job_input: IJobInput):
    args = dict(
        source_table="vm_new_data",
        target_table="dim_vm",
        timestamp_column="arrival_ts",
        id_column="vm_uuid",
    )
    job_input.execute_template("process-note-data-jupyter-notebook", args)
```



```
[ ]: # Import all functions from the 'helper' module,
     # which contains the necessary logic for classification and data visualization
     from helper import visualize_data, classify_score
```

```
[ ]: # Apply the classification function to the 'Score' column to determine the 'Type'
     # Note: this cell might fail on its first run.
     # If it does, simply run it again, and it should work as expected.
     df.loc[:, 'Type'] = df['Score'].apply(classify_score)
```

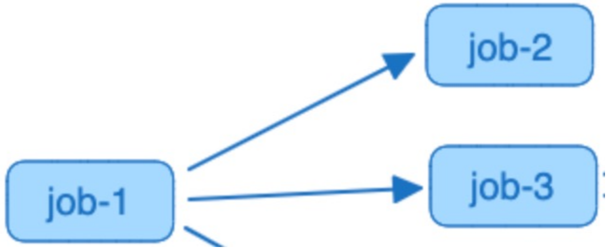
```
[ ]: # Check the DataFrame
     df
```

```
[ ]: # Visualise the types of users
     visualize_data(df)
```

5.2 Data Ingestion

```
[ ]: # Sending data for ingestion
     job_input.send_tabular_data_for_ingestion(
         df.itertuples(index=False),
         destination_table="nps_data",
         column_names=df.columns.tolist()
     )
```

✓ Execute within a workflow

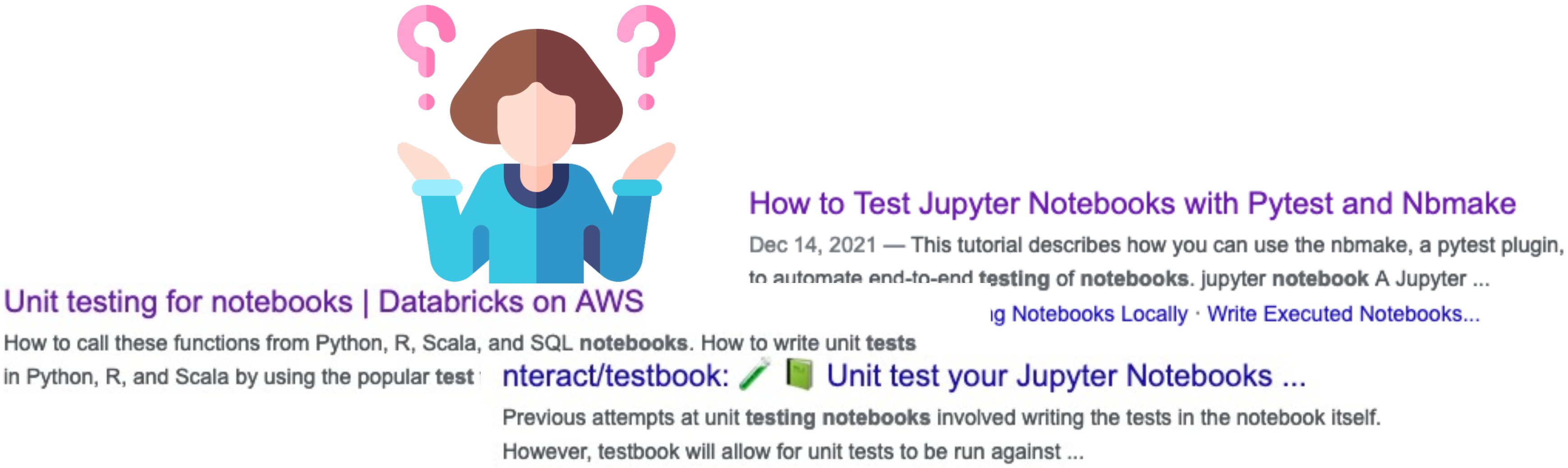


✓ Run automated tests (example coming later)

Challenges

- ✓ Reproducibility: Non-Linear Execution and Hidden State Risks
- ✓ Code Organization: Irrelevant or debugging code
- ✓ Execution model: interactive kernel vs automated flow
- ➢ Automated Testing and CICD
- ➢ Version Control

Automated Testing and CICD



How to Test Jupyter Notebooks with Pytest and Nbmake

Dec 14, 2021 — This tutorial describes how you can use the nbmake, a pytest plugin, to automate end-to-end **testing** of **notebooks**. jupyter **notebook** A Jupyter ...

g Notebooks Locally · Write Executed Notebooks...

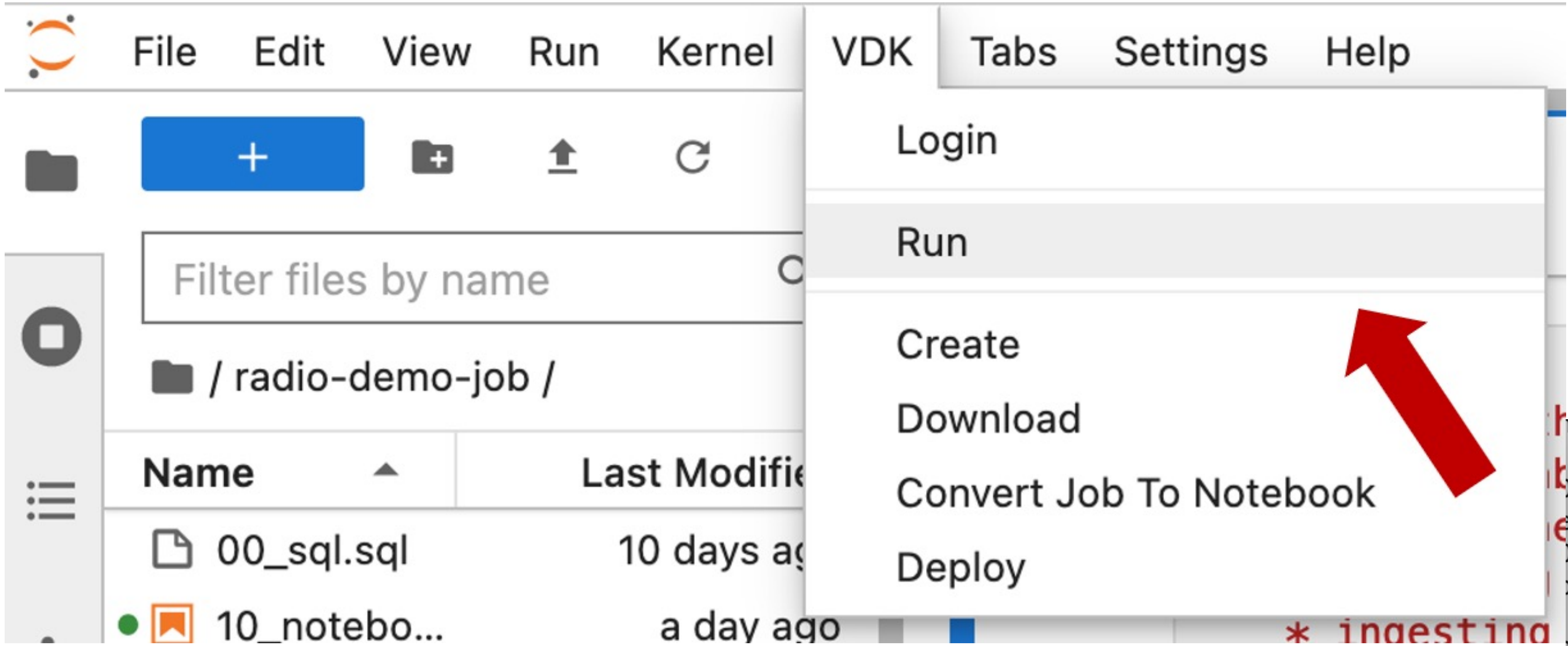
Unit testing for notebooks | Databricks on AWS

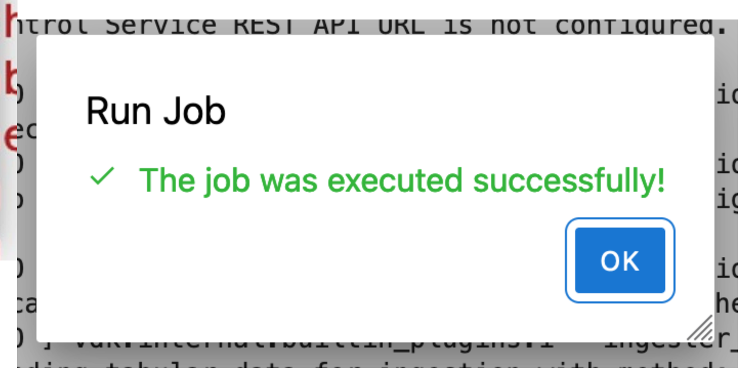
How to call these functions from Python, R, Scala, and SQL **notebooks**. How to write unit **tests** in Python, R, and Scala by using the popular **test**

nteract/testbook: 🧪 📗 Unit test your Jupyter Notebooks ...

Previous attempts at unit **testing notebooks** involved writing the tests in the notebook itself. However, testbook will allow for unit tests to be run against ...

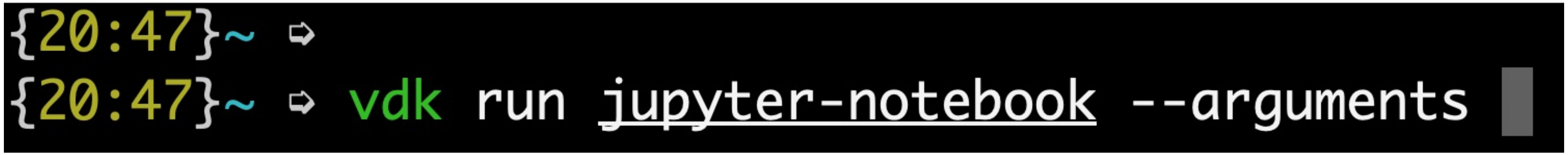
Smoke (end-to-end) testing





Run Job

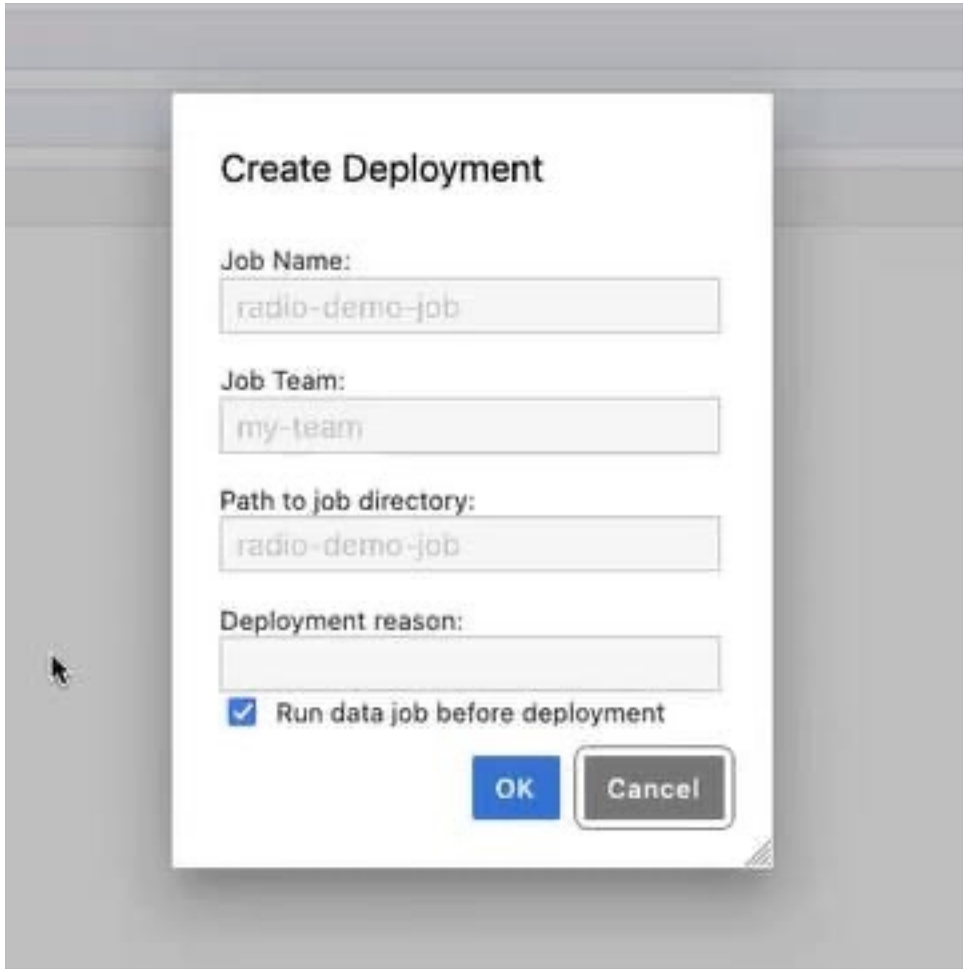
✓ The job was executed successfully!



```
{20:47}~ ➪
{20:47}~ ➪ vdk run jupyter-notebook --arguments
```

On deploy VDK requires passing smoke test first

Opt out possible.



Create Deployment

Job Name: radio-demo-job

Job Team: my-team

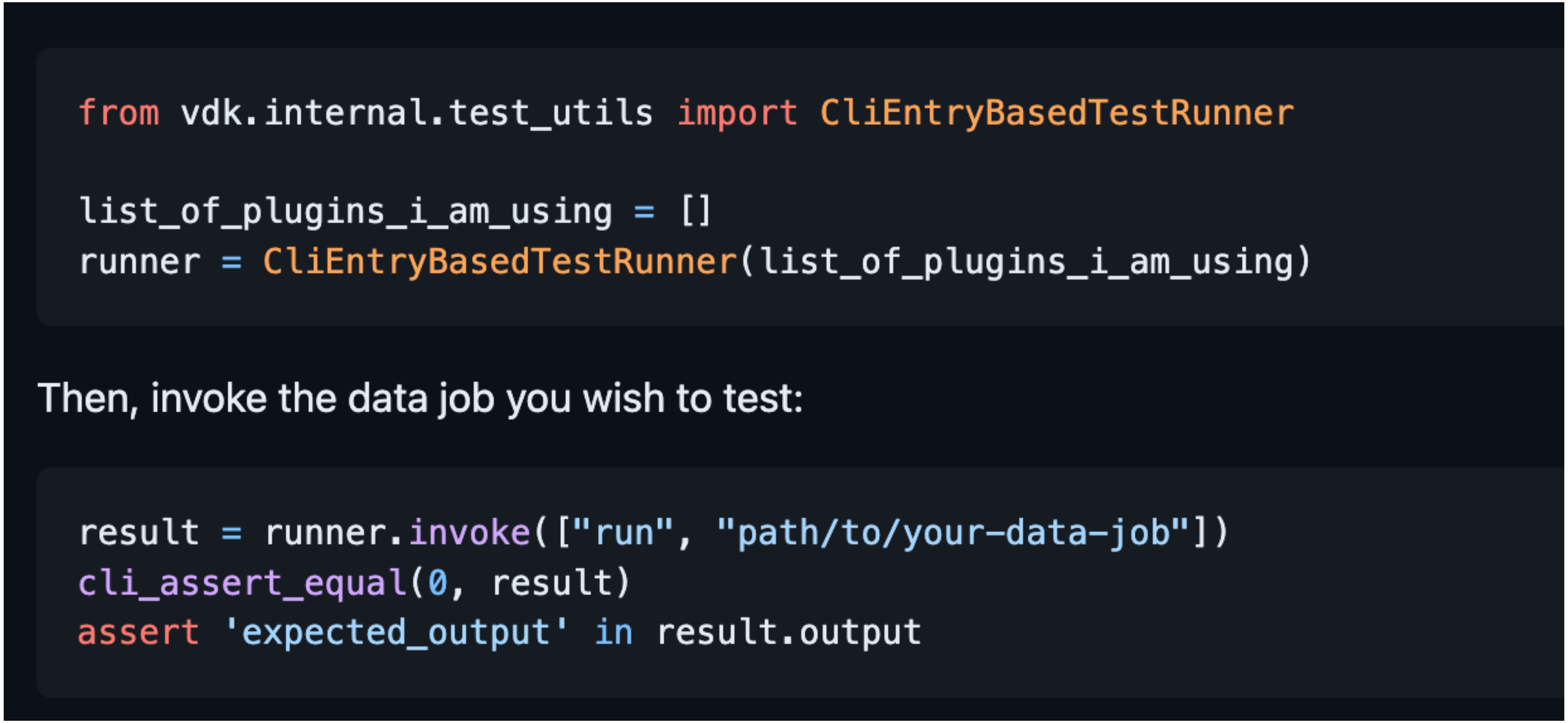
Path to job directory: radio-demo-job

Deployment reason:

☑ Run data job before deployment

Automated testing with pytest

Using VDK testing library “vdk-test-utils”



```
from vdk.internal.test_utils import CliEntryBasedTestRunner

list_of_plugins_i_am_using = []
runner = CliEntryBasedTestRunner(list_of_plugins_i_am_using)
```

Then, invoke the data job you wish to test:

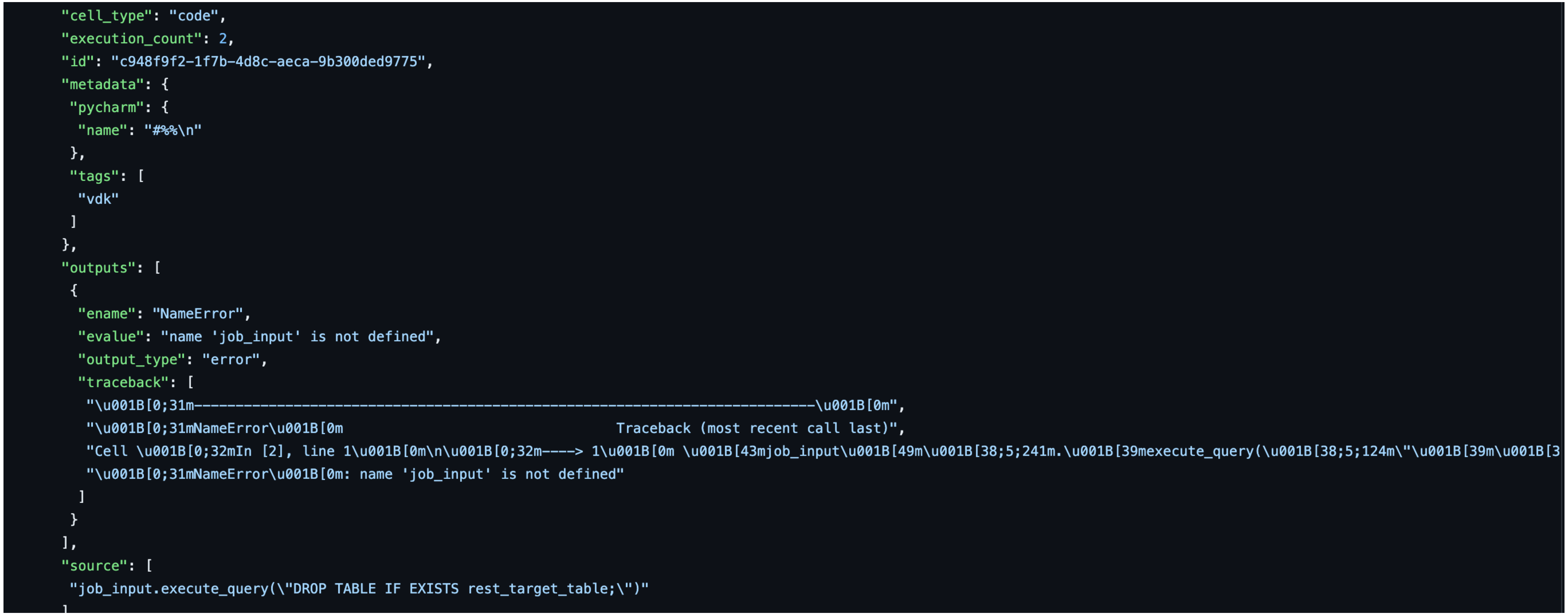
```
result = runner.invoke(["run", "path/to/your-data-job"])
cli_assert_equal(0, result)
assert 'expected_output' in result.output
```

https://github.com/vmware/versatile-data-kit/wiki/Test-VDK-Data-Jobs-with-pytest

Challenges

- ✓ Reproducibility: Non-Linear Execution and Hidden State Risks
- ✓ Code Organization: Irrelevant or debugging code
- ✓ Execution model: interactive kernel vs automated flow
- ✓ Automated Testing and CICD
- ➢ Version Control

Version Control Challenges

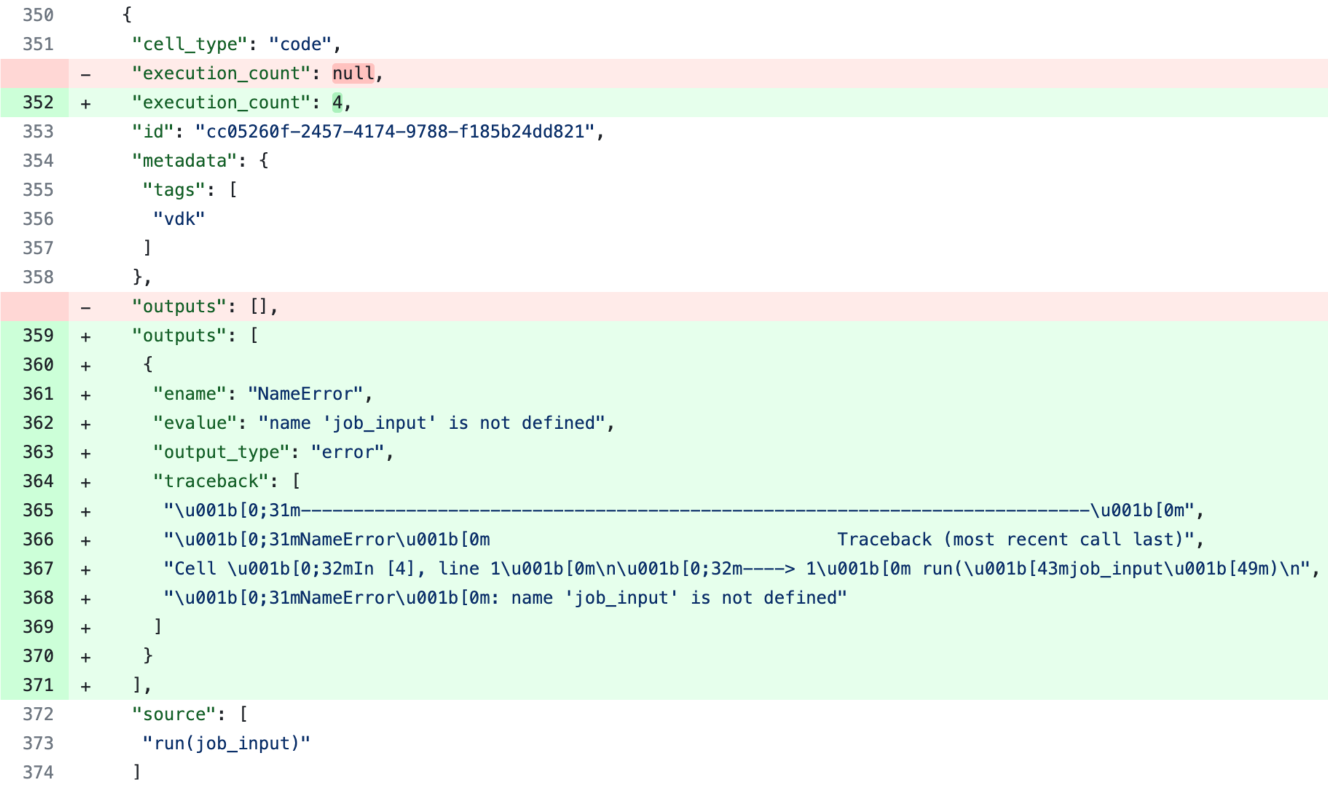


```
   "cell_type": "code",
   "execution_count": 2,
   "id": "c948f9f2-1f7b-4d8c-aeca-9b300ded9775",
   "metadata": {
    "pycharm": {
     "name": "#%%\n"
    },
    "tags": [
     "vdk"
    ]
   },
   "outputs": [
    {
     "ename": "NameError",
     "evalue": "name 'job_input' is not defined",
     "output_type": "error",
     "traceback": [
      "\u001B[0;31m---------------------------------------------------------------------------\u001B[0m",
      "\u001B[0;31mNameError\u001B[0m                                 Traceback (most recent call last)",
      "Cell \u001B[0;32mIn [2], line 1\u001B[0m\n\u001B[0;32m----> 1\u001B[0m \u001B[43mjob_input\u001B[49m\u001B[38;5;241m.\u001B[39mexecute_query(\u001B[38;5;124m\"\u001B[39m\u001B[3
      "\u001B[0;31mNameError\u001B[0m: name 'job_input' is not defined"
     ]
    }
   ],
   "source": [
    "job_input.execute_query(\"DROP TABLE IF EXISTS rest_target_table;\")"
   ]
```

VERSATILE DATA KIT

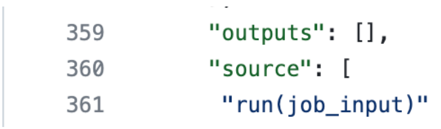
Version Control Challenges

Without VDK



```
 {
  "cell_type": "code",
- "execution_count": null,
+ "execution_count": 4,
  "id": "cc05260f-2457-4174-9788-f185b24dd821",
  "metadata": {
   "tags": [
    "vdk"
   ]
  },
- "outputs": [],
+ "outputs": [
+  {
+   "ename": "NameError",
+   "evalue": "name 'job_input' is not defined",
+   "output_type": "error",
+   "traceback": [
+    "\u001b[0;31m---------------------------------------------------------------------------\u001b[0m",
+    "\u001b[0;31mNameError\u001b[0m                                 Traceback (most recent call last)",
+    "Cell \u001b[0;32mIn [4], line 1\u001b[0m\n\u001b[0;32m----> 1\u001b[0m run(\u001b[43mjob_input\u001b[49m)\n",
+    "\u001b[0;31mNameError\u001b[0m: name 'job_input' is not defined"
+   ]
+  }
+ ],
  "source": [
   "run(job_input)"
  ]
```

With VDK



```
 "outputs": [],
 "source": [
  "run(job_input)"
```

From Data Exploration to Production

- ✓ Reproducibility: Non-Linear Execution and Hidden State Risks
- ✓ Code Organization: Irrelevant or debugging code
- ✓ Execution model: interactive kernel vs automated flow
- ✓ Automated Testing and CICD
- ✓ Version Control

Self-paced tutorial

https://bit.ly/vdk-product-notebooks

Try it yourself

Please take the survey.

Thank You

https://www.linkedin.com/in/antoni-ivanov