Supporting Reproducibility in Jupyter through Dataflow Notebooks

David Koop

University of Massachusetts Dartmouth
Two Kinds of Notebooks
Exploratory Notebooks

D. Koop

Dataflow Notebooks
Jupyter Support for Rapid Exploration

- Flexible environment
  - Edit any cell whenever you want
  - Execute whichever cells you want
- Inline views of outputs
- No context switch
- Easily compare and trace outputs
- Execution shortcuts: run all, run above, run below
Explanatory Notebooks

Gene Expression Data

We obtained gene expression data from the Cancer Cell Line Encyclopedia (CCLE) for 37 lung cancer cell lines assayed by our collaborators at CST. This independent dataset can be used to find novel correlations between differentially expressed genes and PTMs as well as determine whether lung cancer cell lines behave similarly in gene-expression-space and PTM-space. The gene expression data was processed in the CST_Data_Processing.ipynb notebook that: kept the top 1000 genes with the greatest variance across the cell lines, and Z-score normalized the genes across the cell lines to highlight differential expression across the lung cancer cell lines.

In [4]: net.load_file('./lung_cellline_3_1_16/lung_cl_all_ptm/precals_processed/CST_CCLE_exp.txt')
    print('Expression data shape: ' + str(net.dat['mat'].shape))

Expression data shape: (1000, 37)

In [5]: net.set_cat_color('row', 1, 'Data-Type: Exp', 'yellow')
    net.cluster(views={})
    net.widget()
Jupyter Support for Clear Explanation

- Textual explanation: markdown cells
- Graphical explanation: inline figures
- Interactive explanation: Jupyter Widgets
- Publishing: nbconvert can generate Web pages, LaTeX, etc.
- Structure: clear, linear cell layout
- Reproducible
Two Kinds of Notebooks

- Different notebooks for different stages of work
  - Exploratory notebooks
  - Explanatory notebooks
- Parallels a general approach to research
  - Brainstorm questions and explore answers
  - Distill initial results into something understandable and verifiable
  - Publish polished text and (interactive) visualizations
Rapid exploration → Clear explanation?
Reproducibility

- Reproducibility vs. replicability
- Given code & data, check results
- What does this mean for notebooks?
  - Environment: have installed correct versions of libraries/packages
  - Data: have access to all of the necessary data
  - Execution: am able to rerun the cells as intended
Linearizing the Notebook

• “Make sure it runs straight from the beginning to the end” — J. VanderPlas, *Reproducible Data Analysis in Jupyter*

• Rearrange, organize, merge cells

• Collect imports

• Add explanatory text

• Restart the kernel and run all

• Some notebooks put auxiliary code to be run first at the end of the notebook
Reuse

• Run-all, linear solutions can be great
• Can I understand how the original code works?
• Published notebooks
• Your own code
• Do I know how to change the code for my own work?
Exploration Potholes

• Have you ever redefined or mutated a variable and found that you broke another cell?
• Have you ever had a problem getting back to the state you were at before closing a notebook?
• Have you ever edited a cell and forgot to run it?
Solution: Links (dependencies) between cells
IPython History

• "Hold state and capture previous results for reuse" — F. Perez
• Allows users to reference:
  • Previous cell **outputs** (e.g. `Out[9]`, ``, ` `, ` ```)
  • Transcript of all executed code (In[5], `%history`)
• Such commands exist in IPython and Jupyter Notebook (ipykernel)
...but they are not used much
Current IPython Kernel
Current IPython Kernel
Referencing Intermediate Results

• Strategy: Out dictionary stores results using numeric cell ids
• Problem: Cell ids update when a cell is re-executed

• Strategy: Underscore variables (\_) reference last results
• Problem: Out-of-order execution

• Strategy: Global variables
• Problem: Out-of-order execution, mutation
Solution: Persistent, Unique Identifiers

• Each cell is assigned a unique id when it is created, and it maintains that id forever
• This way, we can reference old cells in a way that won't change if we restart the notebook
• Don't imply order to IDs (cells may move)
• ID Scheme: Random integers in hex (like UUIDs)
• Example:
  In [51d03e]: 12 ** 2
  Out[51d03e]: 144
Identifiers

• Problem: No one will remember identifiers like 51d03e
• Solution: Users specify the names of variables to be output from cells and use those names as references in other cells
Recursive Execution

• We don't need linear (top-down) notebooks!
• How to run cells in the right order?
  • Have unique references since each variable is output once (by a specific cell)
  • We know which cells need to be run to allow a cell to execute
  • Recurse!
• Similar to dataflows (or functional programming)—a cell waits for its inputs to be up-to-date (may involve executing other modules/cells) and then executes
A dataflow is a set of computational modules with connections between outputs of one module and inputs of another.

Dataflow execution is bottom-up; all dependencies must be up-to-date before a module is executed.

Upstream dependencies: modules that must be updated first.

Downstream modules: modules whose results may be affected when/if another module's output changes.
Dataflow Notebooks

- A **dataflow notebook** is a set of computational **cells** with connections between outputs of one **cell** and inputs of another.

- Dataflow notebook execution is **bottom-up**; all dependencies must be up-to-date before a **cell** is executed.

- **Upstream** dependencies: **cells** that must be updated before a **cell** is run.

- **Downstream** cells: **cells** whose results may be affected when/if another **cell's** output changes.
How? Wall off cells from each other

- Variables are local to a single cell…
- …but each cell can define **multiple** named outputs via an **expression** or **assignment** on the last line
- Duplicate outputs are not allowed: no more than one cell can output the same name
- Unnamed outputs are ok and you can still connect through ids
How? Create precise connections between cells

- Each cell can access any other cell's outputs by name
- A connection includes both the cell id and the name of the variable
- This allows the notebook to know when any cell is stale (something upstream changed)
Example

In [0a358]:
raw_df = pd.read_csv("fifa17-top20-women.txt", sep="-", header=None)

raw_df:

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>Country</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Caroline Seger</td>
<td>Sweden</td>
<td>85</td>
</tr>
<tr>
<td>1</td>
<td>Wendie Renard</td>
<td>France</td>
<td>85</td>
</tr>
<tr>
<td>2</td>
<td>Steph Houghton</td>
<td>England</td>
<td>85</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

In [aaa3c6]:
column_names = {0: "Name", 1: "Country", 2: "Rating"}
column_names: {0: 'Name', 1: 'Country', 2: 'Rating'}

In [a249ea]:
named_df = raw_df.rename(columns=column_names)
named_df:

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Caroline Seger</td>
<td>Sweden</td>
</tr>
<tr>
<td>1</td>
<td>Wendie Renard</td>
<td>France</td>
</tr>
<tr>
<td>2</td>
<td>Steph Houghton</td>
<td>England</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

In [aab079]:
named_df.groupby("Country").size().sort_values(ascending=False)

Out[aab079]:

<table>
<thead>
<tr>
<th>Country</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>USA</td>
<td>6</td>
</tr>
<tr>
<td>Canada</td>
<td>3</td>
</tr>
<tr>
<td>Brazil</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Example

```python
In [a0a358]:
raw_df = pd.read_csv("fifa17-top20-women.txt", sep="-", header=None)
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Caroline Seger</td>
<td>Sweden</td>
</tr>
<tr>
<td>1</td>
<td>Wendie Renard</td>
<td>France</td>
</tr>
<tr>
<td>2</td>
<td>Steph Houghton</td>
<td>England</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```

```python
In [aaa3c6]:
column_names = {0: "Name", 1: "Country", 2: "Rating"}
column_names = {0: 'Name', 1: 'Country', 2: 'Rating'}
```

```python
In [a249ea]:
named_df = raw_df.rename(columns=column_names)
named_df = raw_df.rename(columns=column_names)
```

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caroline Seger</td>
<td>Sweden</td>
<td>85</td>
</tr>
<tr>
<td>Wendie Renard</td>
<td>France</td>
<td>85</td>
</tr>
<tr>
<td>Steph Houghton</td>
<td>England</td>
<td>85</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

```python
In [aab079]:
named_df.groupby("Country").size().sort_values(ascending=False)
```

```
<table>
<thead>
<tr>
<th>Country</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>USA</td>
<td>6</td>
</tr>
<tr>
<td>Canada</td>
<td>3</td>
</tr>
<tr>
<td>Brazil</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
Example

• Persistent Identifiers
• Named Outputs

In [0a358]:
raw_df = pd.read_csv("fifa17-top20-women.txt",sep="-",header=None)

In [aaa3c6]:
column_names = {0: "Name", 1: "Country", 2: "Rating"}

In [a249ea]:
named_df = raw_df.rename(columns=column_names)

In [aab079]:
named_df.groupby("Country").size().sort_values(ascending=False)

Out[aab079]:
Country  
USA      6
Canada   3
Brazil   3
...      ...

D. Koop
Example

```
In [a0a358]:
raw_df = pd.read_csv("fifa17-top20-women.txt", sep="-", header=None)

raw_df:

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>Country</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Caroline Seger</td>
<td>Sweden</td>
<td>85</td>
</tr>
<tr>
<td>1</td>
<td>Wendie Renard</td>
<td>France</td>
<td>85</td>
</tr>
<tr>
<td>2</td>
<td>Steph Houghton</td>
<td>England</td>
<td>85</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

In [aaa3c6]:
column_names = {0: "Name", 1: "Country", 2: "Rating"}

column_names:

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>Country</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Name 1</td>
<td>Country 1</td>
<td>Rating 1</td>
</tr>
<tr>
<td>1</td>
<td>Name 2</td>
<td>Country 2</td>
<td>Rating 2</td>
</tr>
<tr>
<td>2</td>
<td>Name 3</td>
<td>Country 3</td>
<td>Rating 3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

In [a249ea]:
named_df = raw_df.rename(columns=column_names)

named_df:

<table>
<thead>
<tr>
<th>Name</th>
<th>Country</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caroline Seger</td>
<td>Sweden</td>
<td>85</td>
</tr>
<tr>
<td>Wendie Renard</td>
<td>France</td>
<td>85</td>
</tr>
<tr>
<td>Steph Houghton</td>
<td>England</td>
<td>85</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

In [aab079]:
named_df.groupby("Country").size().sort_values(ascending=False)

Out[aab079]:

<table>
<thead>
<tr>
<th>Country</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>USA</td>
<td>6</td>
</tr>
<tr>
<td>Canada</td>
<td>3</td>
</tr>
<tr>
<td>Brazil</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
Example

In [a0a358]:
raw_df = pd.read_csv("fifa17-top20-women.txt",sep="-",header=None)

<table>
<thead>
<tr>
<th>raw_df:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Country</td>
</tr>
<tr>
<td>Sweden</td>
</tr>
<tr>
<td>85</td>
</tr>
<tr>
<td>Wendie</td>
</tr>
<tr>
<td>Rena</td>
</tr>
<tr>
<td>85</td>
</tr>
<tr>
<td>Steph</td>
</tr>
<tr>
<td>Houghton</td>
</tr>
<tr>
<td>85</td>
</tr>
</tbody>
</table>

In [aaa3c6]:
column_names = {0: "Name", 1: "Country", 2: "Rating"}

<table>
<thead>
<tr>
<th>column_names:</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0: 'Name', 1: 'Country', 2: 'Rating'}</td>
</tr>
</tbody>
</table>

In [a249ea]:
named_df = raw_df.rename(columns=column_names)

<table>
<thead>
<tr>
<th>named_df:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Country</td>
</tr>
<tr>
<td>Rating</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Caroline Seger</td>
</tr>
<tr>
<td>Sweden</td>
</tr>
<tr>
<td>85</td>
</tr>
<tr>
<td>Wendie Rena</td>
</tr>
<tr>
<td>France</td>
</tr>
<tr>
<td>85</td>
</tr>
<tr>
<td>Steph Houghton</td>
</tr>
<tr>
<td>England</td>
</tr>
<tr>
<td>85</td>
</tr>
</tbody>
</table>

In [aab079]:
named_df.groupby("Country").size().sort_values(ascending=False)

<table>
<thead>
<tr>
<th>Out[aab079]:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Country</td>
</tr>
<tr>
<td>USA</td>
</tr>
<tr>
<td>Canada</td>
</tr>
<tr>
<td>Brazil</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

• Persistent Identifiers
• Named Outputs
• Unnamed Outputs
• Connection by Variable Reference

D. Koop
Implementation

- dfkernel (dataflow kernel)
  - Modifies both the interface and the execution
  - Updates to ipykernel
  - Hacks and extensions to the notebook
More Details

- Use a custom namespace (user_ns) to intercept variable access
- When a variable is accessed:
  - look up its associated cell in the kernel's dependency graph
  - run any upstream dependencies
  - run the cell
  - pull out the specific output
- We cache outputs so they do not need to be recomputed
- Augment run_ast_nodes to wrap the cell in a closure and customize the execution of the last line so that we can catch and register the variables a user chose to output
Understanding and Reusing Notebooks
Understanding a Notebook

- Add status icons for cells that indicate state
  - 🟢 New
  - 🟢 New with content
  - ⏳ Executing
  - ✔ Successfully executed and up-to-date
  - ✗ Executed but with an exception
  - ⚠ Stale (either edited or upstream was edited)
  - ✔ Successfully executed in last session
  - ✗ Had exception in last session
Cell Toolbars

In [baae9d]:

```python
import pandas as pd
...
```

In [a0a358]:

```python
raw_df = pd.read_csv("fifa17-top20-women.txt", sep=" ", header=None)
...
```

In [asa3c5]:

```python
column_names = [0: "Name", 1: "Country", 2: "Rating"]
...
```

In [a249ea]:

```python
named_df = raw_df.rename(columns=column_names)
...
```

In [aab079]:

```python
named_df.groupby("Country").size().sort_values(ascending=False)
...
```
Cell Toolbar

- Select all upstream/downstream cells
- Input/output variable names
- Force cached: use a cached value unless this cell is explicitly executed
- Auto-refresh: always re-run this cell if it becomes stale (reactive)
Interactive Dependency Graph

- Shows connections between cells
- Can see which cells would be affected by a change
- Same colors indicate which parts of the graph are stale
- Linked to the notebook
  - Hover to show a cell's code
  - Can also execute in the graph
Demo

Dataflow Notebooks

D. Koop
Widgets

Boston Housing: Percent Built Before 1940

In [f2d972]:
```python
plt.hist(data[:, 6], bins=5)
```

In [c51ef8]:
```python
import ipywidgets
import matplotlib.pyplot as plt

plt: <module 'matplotlib.pyplot' from '/Users/dkoop/anaconda/envs/dfkernel/lib/python3.6/site-packages/matplotlib/pyplot.py'>

ipywidgets: <module 'ipywidgets' from '/Users/dkoop/Research/notebook-prov/src/ipywidgets/ipywidgets/__init__.py'>
```

In [c8e921]:
Widgets

Boston Housing: Percent Built Before 1940

```
In [f2d972]:
plt.hist(data[:,6], bins=5)
```

```
In [c51ef8]:
import ipywidgets
import matplotlib.pyplot as plt
plt: <module 'matplotlib.pyplot' from '/Users/dkoop/anaconda/envs/dfkernel/lib/python3.6/site-packages/matplotlib/pyplot.py'>
ipywidgets: <module 'ipywidgets' from '/Users/dkoop/research/notebook-prov/src/ipywidgets/ipywidgets/__init__.py'>
In [c8e921]:
```
Known Issues

- If you mutate an output, you do so at your own peril
- Similarly, nested references can introduce issues
- Anything that assumes it can access globals via CPython can be problematic if you're trying to reference an output (static class variable assignment)
- Multiple interfaces to the same kernel (a la JupyterLab change kernel)

- Can convert your dfkernel notebook to a standard ipykernel notebook
Related Work

- **reactivePy**: kernel that does reactive computation
- **Nodebook**: cell order defines execution order
- **Observable**: reactive JavaScript notebooks
- **JetBrains Datalore**: incremental computation that follows dependencies
- **Mathematica's Dynamic**
Acknowledgments

• Contributors: Colin Brown, Hieu (Henry) Ngo, Jay Patel

• Jupyter and IPython Development Teams

• This research was partially supported by a generous donation by the State Street Foundation to the Data Science program at the University of Massachusetts Dartmouth, USA.

• Photos from unsplash.com, Photographers: Linus Sandvide, Benjamin Gremler, Perry Grone, Jon Tyson, Bernard Hermant, Marc Olivier Jodoin, Gradikaa
github.com/dataflownb/dfkernel