dfpy_kernel Documentation

Release 0.0.1

Dataflow Kernel Developers

Apr 12, 2019

Contents

1 Installation **1**
1.1 Contents 1

CHAPTER 1

Installation

To install the Dataflow Python Kernel (Package)

>> pip install dfkernel

From Source

>> git clone

>> cd dfkernel

>> pip install -e .

>> python -m dfkernel install [–user|–sys-prefix]

Note: –sys-prefix works best for conda environments

1.1 Contents

1.1.1 Dataflow Kernel Tutorial

Controlling Output references

In the Dataflow Kernel there are several different ways to export your variables into the current namespace that you're working in. Any unnamed type will simply be given a persistent identifier that matches the persistent cell identifier that the input is given.

```
In [e90038]: 2+3
```

```
Out[e90038]: 5
```

You can also give an output a tag by assigning a name to it, as long as the variable has a name we can just export that into the namespace.

```
In [aabe16]: a = 3
```

```
a: 3
```

All names are considered persistent throughout the notebook, this means if you try and reassign it in a different cell you'll get an error however, this can be resolved by simply deleting that cell and you'll be able to redefine that name in any other cell.

```
In [f2146e]: a = 4
```

```
---------------------------------------------------------------------------
DuplicateNameError                        Traceback (most recent call last)
DuplicateNameError: name 'a' has already been defined in Cell 'aabe16'
```

You can also assign multiple variables at once as long as you pass multiple references.

```
In [cff2e4]: c,d = 3,4
```

```
c: 3
```

```
d: 4
```

```
In [b7f27b]: e,f = 5,6
             e,f
```

```
e: 5
```

```
f: 6
```

In the case of mixed named and unnamed variables the named variables will be pulled out by their variable names and any unnamed variables will be given the ability to be referenced through bracket notation.

```
In [aaa071]: g,h = 5,6
             g,h,[2,3]
```

```
g: 5
```

```
h: 6
```

```
Out[aaa071][2]: [2, 3]
```

Output Magics

Since a user might want to split out a dictionary into multiple variables we've provided methods for that.

NOTE: In versions of Python below 3.6 tags are not guaranteed to be in order as dict keys have no guaranteed ordering and if you desire your keys to be in order we suggest the user of ''OrderedDict''.

```
In [fff27b]: %split_out {'j':1,'k':1}
```

```
j: 1
```

```
k: 1
```

Accessing Outputs

The Dataflow kernel allows for several different methods of accessing exported variables in the notebook. Completion is enabled for only the last cell in the most current revision, despite this cells and tags that have been exported can all be autocompleted by using the completer.

```
In [c3f778]: m = 4
```

```
m: 4
```

_ + <tab> produces

```
In [d04d7a]: m
```

```
Out[d04d7a]: 4
```

```
In [f2cc87]: n,o = 1,2

n: 1

o: 2
```

When a cell has multiple tags the completer will produce a tuple of the outputs

```
In [f21db2]: (n,o)

Out[f21db2]: (1, 2)
```

A cell can also still be addressed as it's original output tag if desired but this behavior is discouraged unless the user needs to reference outputs this way.

In the example below typing `f2c` and hitting `<tab>` will result in the following

```
In [def65e]: Out[f2cc87]

Out[def65e]: (1, 2)
```

Auto-Parse Library functionality

To make life easier on users we came to the conclusion that libraries should be parsed out at a local level because writing extra code to specifically export libraries slows users down.

So even when you write only assignments the library will be parsed out and become attached to that cell.

```
In [f2e1a8]: import sys
             p = 3

sys: <module 'sys' (built-in)>

p: 3
```

However, when you access the cell or try to reference it, it will perform in the same way you expected before so it doesn't become a problem when referencing objects.

```
In [efd9e7]: Out[f2e1a8]

Out[efd9e7]: 3
```

It also performs the same way when you have multiple tags but is instead referenced as a tuple.

```
In [bcaf80]: import os
             q,r = 1,2

os: <module 'os' from '/home/colin/anaconda3/envs/noglobals/lib/python3.6/os.py'>

q: 1

r: 2

In [f0a682]: Out[bcaf80]

Out[f0a682]: (1, 2)
```

1.1.2 Jupyter Notebook Interactions

To be able to properly accomodate both UUIDs and our namespace into the notebook we've had to make some changes to the way normal notebooks work.

In the vanilla notebook an action like copying a cell and pasting it twice has no reprecussions, for us however these interactions had to be changed to incorporate things like ensuring that no Cell IDs is the same.

Let's step through some of these interactions.

Delete Cell

A cell that has never been executed is the same interaction you'd see in a normal notebook, it removes the cell and you never have to deal with that cell ever again.

However, with `CodeCells` that have already been executed that is not the case, a horizontal red bar will show up where that cell was.

```
In [d7a7c8]: from IPython.display import Image
```

```
Image: IPython.core.display.Image
```

```
In [aed959]: Deletee14d16 = Image(url='https://rawgit.com/colinjbrown/dfkernel/documentation-update/
```

```
Deletee14d16: <IPython.core.display.Image object>
```

It's best to think of this cell as a "Soft Delete", it's completely reversible without any consequences. The reason why this is the case is because until a new cell is executed the Python kernel doesn't even know you've deleted a cell. Upon executing another cell though this bar will be completely removed and will be considered "hard deleted".

Note: We think this is the best way to handle these deletions because when you delete any cell in the Notebook you might not be aware of the cells that are impacted by it.

Copy, Cut and Paste Cell

Copy

Copy behaves just as you imagine it would so there are no special interactions that happen here.

Cut

On cutting a cell, much like when we delete a cell performing a soft delete. However there are now two copies of the cell that exist, one exists in the undelete stack and one exists on the clipboard. Both of these are considered equally valid references so until one of two events happens they are all valid.

- Paste Event: Pasted Cell is now considered the authentic reference to that cell and the deleted cell maintains a different Cell ID and has output wiped if it is undeleted.
- Undelete Event: Undeleted Cell is now considered the authentic reference to that cell and the cell on clipboard maintains a different Cell ID and has output wiped if it is pasted.

Paste

This behaves relatively the same other than two references to the same cell cannot exist, so if you try and for example paste twice they will have two different Cell IDs and the second one will not have any output attached to it.

Split and Merge

Split

This is the same as a typical split, one side gets the output tags and the other does not, determining the proper way to split requires code introspection so the onus falls on the user to not make poor choices.

Merge As text is collapsed together so is the output, we only retain the outputs from the bottom cell.

1.1.3 Cell Statuses

```
In [e84c3f]: from IPython.display import Image
```

```
Image: IPython.core.display.Image
```

```
In [b87ea7]: Success =          Image(url='https://cdn.rawgit.com/colinjbrown/dfkernel/29b7b020/dfkern
             SavedSuccess =     Image(url='https://cdn.rawgit.com/colinjbrown/dfkernel/29b7b020/dfkern
             Running =          Image(url='https://cdn.rawgit.com/colinjbrown/dfkernel/29b7b020/dfkern
             Unverified =       Image(url='https://cdn.rawgit.com/colinjbrown/dfkernel/29b7b020/dfkern
             NewCell =          Image(url='https://cdn.rawgit.com/colinjbrown/dfkernel/29b7b020/dfkern
             EditedNewCell =    Image(url='https://cdn.rawgit.com/colinjbrown/dfkernel/29b7b020/dfkern
             Error =            Image(url='https://cdn.rawgit.com/colinjbrown/dfkernel/29b7b020/dfkern
             UnverifiedError =  Image(url='https://cdn.rawgit.com/colinjbrown/dfkernel/29b7b020/dfkern
             Success,SavedSuccess,Running,Unverified,NewCell,EditedNewCell,Error,UnverifiedError
```

```
Success: <IPython.core.display.Image object>
```

```
SavedSuccess: <IPython.core.display.Image object>
```

```
Running: <IPython.core.display.Image object>
```

```
Unverified: <IPython.core.display.Image object>
```

```
NewCell: <IPython.core.display.Image object>
```

```
EditedNewCell: <IPython.core.display.Image object>
```

```
Error: <IPython.core.display.Image object>
```

```
UnverifiedError: <IPython.core.display.Image object>
```

Success:

This represents the best case for a cell, this represents a cell that has had no change to it's text and everything upstream of it is also in this state.

SavedSuccess:

The last run of this cell was successful, there's a good chance that this will succeed too, but if however you were missing a file that the original author had the cell would fail.

Running:

This state is only reached when a cell continues to run, as our UUIDs are persistent we've gotten rid of the typical `[*]` state one would see in a vanilla notebook and replaced it with this icon as a feedback mechanism for users.

Unverified:

This represents a stale state, this state can be reached by a number of ways. If a user were to edit an upstream cell not only would we mark that cell as stale because that cell has not yet been executed, but everything that is downstream from that cell has reached a stale state as they depend on that cell.

NewCell:

A new empty cell.

EditedNewCell:

A new cell that has had it's contents modified.

Error:

The opposite of a Success, this is a verified error state where the error has happened locally.

UnverifiedError:

Most of the time an error is an error, however that is not always the case if whoever saved the file was missing a library and you have it. Without knowing the cause of an error we must mark it as unverified when the Notebook is saved out.

1.1.4 Dependency Viewer Actions and Interactions

```
In [ca5d7e]: from IPython.display import Image
```

```
Image: IPython.core.display.Image
```

Graph Updates

To best describe what happens when a graph updates we must first open the dependency viewer.

```
In [ea1465]: DepViewerOpen = Image(url='https://rawgit.com/colinjbrown/dfkernel/documentation-update/
```

```
DepViewerOpen: <IPython.core.display.Image object>
```

Once our viewer is open we can execute code to create some nodes. #### Note: The viewer does not have to be open, changes that happen while the viewer is not open will be rendered upon opening of the viewer.

```
In [b89f91]: a = 3
```

```
a: 3
```

After our cell finishes executing the viewer will now update the graph.

```
In [e775eb]: Executeb89f91 = Image(url='https://rawgit.com/colinjbrown/dfkernel/documentation-update/
```

```
Executeb89f91: <IPython.core.display.Image object>
```

```
In [d91de0]: b = a+3
```

```
b: 6
```

When we execute a second cell the viewer will move out and the edge between `a` and `Cell[d91de0]` will be shown.

```
In [a9ad40]: Executed91de0 = Image(url='https://rawgit.com/colinjbrown/dfkernel/documentation-update/
```

```
Executed91de0: <IPython.core.display.Image object>
```

```
In [e14d16]: c = b+5
```

```
c: 11
```

This process then can again be repeated for another cell.

```
In [d5ce14]: Executee14d16 = Image(url='https://rawgit.com/colinjbrown/dfkernel/documentation-update/
```

```
Executee14d16: <IPython.core.display.Image object>
```

When we decide to delete `Cell[b92938]` the changes are not rendered immediatly because the kernel does not know about these changes, instead we opt to provide a red horizontal cell that acts as a place holder until that cell is actually removed. Thus inside of the viewer the cell is also rendered in red.

```
In [d7faa2]: Deletee14d16 = Image(url='https://rawgit.com/colinjbrown/dfkernel/documentation-update/c
             GraphUpdate = Image(url='https://rawgit.com/colinjbrown/dfkernel/documentation-update/do
             Deletee14d16,GraphUpdate
```

```
Deletee14d16: <IPython.core.display.Image object>
```

```
GraphUpdate: <IPython.core.display.Image object>
```

When we decide to click on the red bar to undelete the cell the viewer restores the color of that cell to green as if nothing had happened to that cell at all.

```
In [b782f3]: Restoree14d16 = Image(url='https://rawgit.com/colinjbrown/dfkernel/documentation-update/
```

```
Restoree14d16: <IPython.core.display.Image object>
```

Now we edit `In[b89f91]` so that instead of `a=3` we set `a=2`, this results in the graph being updated again.

```
In [c2ca48]: Editb89f91 = Image(url='https://rawgit.com/colinjbrown/dfkernel/documentation-update/doc
```

```
Editb89f91: <IPython.core.display.Image object>
```

Toggle Cell Sinks

To be able to demonstrate exactly what this does a more complicated graph needs to be shown, so we will use the graph from a fully executed example of ours that can be found here.

```
In [a80e4d]: CellSinksVisible = Image(url='https://cdn.rawgit.com/colinjbrown/dfkernel/documentation-
```

```
CellSinksVisible: <IPython.core.display.Image object>
```

Upon hitting the toggle to remove cell sinks, we can see that any sink nodes in our graph are now removed and our graph is significantly more compact.

```
In [e88bb4]: NoCellSinks = Image(url='https://cdn.rawgit.com/colinjbrown/dfkernel/documentation-updat
```

```
NoCellSinks: <IPython.core.display.Image object>
```

1.1.5 Dependencies and the Cell Toolbar

Dependencies in the notebook will execute upstreams if the code in an upstream cell changes. Code caches are checked to ensure that everything stays up to date.

```
In [f9cfcd]: from IPython.display import Image
```

```
Image: IPython.core.display.Image
```

```
In [c4ae59]: a = 3
```

```
a: 3
```

```
In [e26943]: b = a+2
```

```
b: 5
```

Upon changing `a` to be a different value such as `a=2` and executing `Cell[e26943]`, `Cell[c4ae59]` will immediatly execute to ensure that `b` now returns `6` instead.

How this works: To ensure that the ‘‘a‘‘ being referenced is the proper ‘‘a‘‘ we have enforced a variable name lockdown. That means that any attempts to reassign a will result in failure. However this only happens at the final output stage, every cell in the Dataflow kernel is a closure.

```
In [d7c413]: a = 4
```

```
---------------------------------------------------------------------------
DuplicateNameError                        Traceback (most recent call last)
DuplicateNameError: name 'a' has already been defined in Cell 'c4ae59'
```

Defining every cell as a closure allows for more interesting behavior but the user has to be careful as the `a` in the folowing case is considered to be a local variable.

```
In [dd9629]: a = 5
             c = a+4
```

```
c: 9
```

However you can ensure that you are referring to the correct a by the use of the following keywords `global` and `nonlocal`.

```
In [ccf913]: global a
             a = 6
             d = a+3
```

```
d: 6
```

In the following case a user may want to refer to a locally declared variable `a` inside a function or class and switch between this locally declared variable and one that is in the global namespace.

```
In [c08a91]: a = 6

             class g():
                 def __init__(self):
                     self.f()
                     self.g()
                 def f(self):
                     nonlocal a
                     print(a)
                 def g(self):
                     global a
                     print(a)
             g()
```

```
6
3
```

```
Out[c08a91]: <__main__.__closure__.<locals>.g at 0x7f49dc5fa6a0>
```

Cell Toolbars

To allow users an easy way to get an overview of the upstream and downstream dependencies we've provided a cell toolbar that is enabled from the view dropdown.

```
In [df4af9]: CellToolBar = Image(url='https://rawgit.com/colinjbrown/dfkernel/documentation-update/do
```

```
CellToolBar: <IPython.core.display.Image object>
```

Upon enabling the dataflow toolbar and executing the cells below we get an overview of the dependencies in each cell as well as the ability to toggle refresh states.

```
In [ed0889]: x = a+10
             y = b+4
             x,y
```

```
x: 13
```

```
y: 9
```

```
In [a6cb70]: z = x+y
```

```
z: 22
```

```
In [d8a75f]: CellToolBarOpen = Image(url='https://rawgit.com/colinjbrown/dfkernel/documentation-updat
```

```
CellToolBarOpen: <IPython.core.display.Image object>
```

In this menu by clicking on you are able to use the cached version of this cell, anything that is retrieved from this cell will be from the last run.

While clicking on will make sure that if a cell that is upstream of this cell executes that you want this downstream cell to also execute.

If a user clicks on all of the cells that are upstream from this cell will be selected in the notebook.

Where as if a user clicks on all of the cells that are downstream from this cell will be selected instead.

Note: The Auto-Refresh on downstream is setup this way because it's not explicit that a user would want a downstream cell to be re-executed or not, all cells upstream of a cell have to be re-executed to be considered valid.

1.1.6 Dataflow IPykernel Convert

Installation

Package

>> pip install dfipy_convert

>> jupyter bundlerextension enable –sys-prefix –py dfipy_convert

From Source

>> git clone

>> pip install .

>> jupyter bundlerextension enable –sys-prefix –py dfipy_convert

Convert to IPython kernel compliant notebook

Open Dataflow Python Kernel Notebook File

Open File Menu

Go under Download menu

Click “Convert to IPykernel Compliant Notebook”

Convert to Dataflow kernel compliant notebook

Open IPykernel Notebook File

Open File Menu

Go under Download menu

Click “Convert to Dataflow Compliant Notebook”

1.1.7 Contact Information

David Koop

dkoop@umassd.edu

Colin Brown

cbrown12@umassd.edu

Hieu Ngo

hngo1@umassd.edu

- genindex
- modindex
- search