jupyter_notebook_pyfred_tutorial

July 30, 2017

1 Run from Jupyter Notebook

A Jupyter Notebook (such as this one) may be used to interact with the FRED kernel providing a very handy interactive graphical notebook/documentation environment.

Jupyter Notebook may have already been installed along with your python dsitribution. For more information on installation, see here: http://jupyter.readthedocs.io/en/latest/install.html

Change into the directory where you want the notebook to have file access and start with:

```
$ jupyter notebook
```

Alternatively launch your notebook as usual and make sure your project is accesible below your top level notebook path.

Running `%qtconsole` from a notebook cell will pop-up an IPython GUI terminal with access to the notebook kernel.

Run the following cells individually in order (use SHIFT-ENTER on a highlighted cell to run it). Read through the accompanying documentation as well. If you find your working environment getting "out of control", close out FRED (and your qtconsole) and click the "Kernel" menu up top and then "Restart". To pick up where you left off, navigate to and click "Cell :: Run All Above".

In [1]:
```
# To get remote console connection info use:
#%connect_info
# To open a GUI console:
%qtconsole
# Embed plots in the notebook
%matplotlib inline

# NumPy is nice to have around
import numpy as np
np.set_printoptions(precision=4) # 4 decimal places for printing is OK

import time # For time delay
# Get IPython's pretty printer
from IPython.lib.pretty import pprint
# Typeset arbitrary latex math with display(Math('y=x^2')) where y=x^2 is any LaTeX ma
from IPython.display import display, Math, Latex, HTML, SVG
```

In [2]:
```
# Make a convenience function for printing out instance attributes
# This is of minimal value in any actual application but I use
```

```
        # it here in the demo as a means of information display
        def probj(obj):
            attrs = [x for x in dir(obj) if not x.startswith('_')]
            outstr = ''
            for i in attrs:
                if 60 < len(outstr):
                    print(outstr)
                    outstr = ''
                outstr += "{:<16s}".format(i)
            print(outstr)
```

1.1 Introduction

1.1.1 Core Modules

Run the following cell to import pyfred into your namespace:

```
In [3]: # Imports for getting started with pyfred
        from pyfred import core as pyfred
        from pyfred import apicmds
```

In the IPython shell (qtconsole), the help facilty is very nice. For instance, to see all of the available attributes/methods of pyfred you could type:

```
In [1]: pyfred.<TAB>
```

and get a list of available methods:

```
In [4]: probj(pyfred) # Emulate IPython output
```

```
CWD             Camera          ComLib          DocBase
DocCollection   DocInit         DocProperties   Entities
FunctGetter     MODNAME         SCRIPTPATH      ScriptLib
api             math            np              os
u               w32
```

Use TAB completion to expand .DocInit

```
In [1]: pyfred.DocI<TAB>
```

Get the available help documentation by typing a '?' and hitting ENTER

```
In [1]: pyfred.DocInit?
```

The help facility also works in the Jupyter Notebook. Try running the following cell:

```
In [5]: pyfred.DocInit?
```

Click the 'x' in the top right corner of the help window to hide it.
Now use `.DocInit` to launch FRED by running the next cell:

```
In [6]: FDOC = pyfred.DocInit('testdoc')
```

FRED should launch and open a new document named `testdoc`. Our `FDOC` variable is now an object we can use to interact with the FRED document. Check out the available methods from the IPython console:

```
In [1]: FDOC.<TAB>
```

```
In [7]: probj(FDOC) # Emulate IPython output
```

```
app             comment         coprint         dobj
entities        oprint          struct          units
```

The `.dobj` attribute provides access to the raw COM interface and it's handy to keep that around in a global variable:

```
In [8]: DOBJ = FDOC.dobj # Provides raw access to the COM interface
```

It's not advised to try TAB completion on `DOBJ.` as IPython will dutifully suggest *all* of FRED's available commands. If you do this and then get lost in a sea of suggested commands, just hit ESC or `q` to back out.

Completion on DOBJ can still be quite handy for searching out a command you want. For instance, you could get all of the `Init` commands using:

```
In [1]: DOBJ.Init<TAB>
```

You might see the command you want is `InitEntity` so you just add an `E` and hit TAB again:

```
In [1]: DOBJ.InitE<TAB>
```

IPython fills out `DOBJ.InitEntity` and suggests:

```
DOBJ.InitEntity  DOBJ.InitEntityArray
```

Or, maybe you don't remember the command you want start with, but you know a string that it contains. Say for instance `Qbfs`. Try this:

```
In [1]: DOBJ.*Qbfs*?
```

And you'll get back all the commands with substrings that match:

```
DOBJ.AddQbfsSurf
DOBJ.GetQbfsSurf
DOBJ.GetQbfsSurfCoefCount
DOBJ.GetQbfsSurfIthCoef
DOBJ.SetQbfsSurf
DOBJ.SetQbfsSurfIthCoef
```

Help documentation is not available on raw COM objects. That is accomplished with the pyfred API wrapper which we'll introduce next.

1.1.2 Python wrapper

Controlling FRED through the COM iterface has some issues. Many things work, some things do not and behavior can be inconsistent. pyfred adresses these issues with the `apicmds` module. This module is built directly from the HTML documentation written by Photon Engineering for FRED. This module will not exist until you have completed running the install scripts included with pyfred. These "wrapped" commands provide pythonic access to FRED consistent with their documented implentation and avoid the quirks of the COM interface. Additionally this provides inline access to FRED's documentation for these commands.

In some cases where the documentation is not consistent with the command implementation, the wrapped command will fail. However, broken commands may be fixed using the `api_overrides.yaml`. If you find a broken command that requires an override please let me know so I can merge it into the mainline distribution of pyfred.

Also if the wrapped command is not functional, it may work just fine through the raw COM interface. Some times it may be better to use the raw COM interface for performance reasons anyway. Qualitatively I have found better performance when using raw COM commands vs. the python wrapped API.

Instantiate the python wrapped API by wrapping the raw document object:

```
In [9]: # The python wrapped version of FRED's API will be available in 'api':
        api = apicmds.Wrap(DOBJ)
```

Again TAB completion works as before and it is not suggested to attempt completion using `api.<TAB>` since you'll get the whole library of available commands back.

Now we do have access to full command documentation at our fingertips!

```
In [1]: api.GetEntity?
```

```
In [10]: help(api.GetEntity)

Help on method GetEntity in module pyfred.apicmds:

GetEntity(n, entity) method of pyfred.apicmds.Wrap instance
    Python API documentation:
    =========================
    Wrapper for FRED GetEntity SUBROUTINE.
    Requires all parameters to be set when invoked.

    Retrieves the generic entity data for a specified FRED
    entity.

    Parameters
    ----------
    n: <class 'int'>
    entity: <com_record 'T_ENTITY'>

    Returns
    -------
    [n: <class 'int'>, entity: <com_record 'T_ENTITY'>]
```

```
FRED documentation:
===================

Description:
------------
   Retrieves the generic entity data for a specified
FRED entity.

Examples:
---------
   Entity Info Functions

Parameters:
-----------
   n As Long
     Node number of the FRED entity.
   entity As T_ENTITY
     The generic entity data for the specified
FRED entity.

Remarks:
--------
   This subroutine retrieves the generic entity
data for a specified FRED entity.ă If there is a problem,
the subroutine sets an error and returns without modifying
entity.

See Also:
---------
   T_ENTITY, FindFullName, InitEntity, SetEntity

Syntax:
-------
   GetEntity n, entity
```

1.1.3 COM vs pyfred API example

Get functions that return FRED data structures (such as GetEntity) do not work through the raw COM interface. Also functions that don't take any arguments cause problems.

Take for example the FRED `GetEntity()` subroutine:

```
In [11]: # Raw GetEntity() does not work through the w32 API
         geom_w32 = DOBJ.GetEntity(2, FDOC.struct('T_ENTITY'))

         try:
```

```
            assert('Geometry' == geom_w32.name)
        except AssertionError:
            print("If the raw GetEntity() had worked, we would've gotten:\n"
                  "  name:'{}'\nInstead we got:\n  name:'{}'".
                  format('Geometry', geom_w32.name))
        else:
            print("w32 raw function worked properly!!")
```

```
If the raw GetEntity() had worked, we would've gotten:
  name:'Geometry'
Instead we got:
  name:''
```

```
In [12]: # Using api.GetEntity however works as expected:
         geom_id, geom_api = api.GetEntity(2, FDOC.struct('T_ENTITY'))
         # We get out a tuple of: (<ID>, <DATASTRUCT>) as per
         # the GetEntity() documentation
         try:
             assert('Geometry' == geom_api.name)
         except AssertionError:
             print("Expecting name:'{}'\nInstead we got:\n  name:'{}'".
                   format('Geometry', geom_api.name))
         else:
             print("Expecting name:'{}'\nAnd we got name:'{}'".
                   format('Geometry', geom_api.name))
```

```
Expecting name:'Geometry'
And we got name:'Geometry'
```

1.2 Usage

Here's a simple example of looping over the active FRED document entities:

```
In [13]: outstr = ''
         for i in range(DOBJ.GetEntityCount()):
             outstr += "Entity {}: {}\n".format(i, DOBJ.GetFullName(i))
         print(outstr)
```

```
Entity 0: System
Entity 1: Optical Sources
Entity 2: Geometry
Entity 3: Analysis Surface(s)
```

1.2.1 Printing to the output window

An occasional inconvenience with controlling FRED from an external application is that access to FRED's output window is lost. `FDOC.oprint` will send string output to FRED's console and `FDOC.coprint` sends output to FRED and the local python standard output:

```
In [14]: # We can send output to the FRED output window.
         # Beware linefeed/carriage returns do not advance the cell row.
         # All of the output goes into the next cell including lf/cr's.
         # It's best to call oprint per line of text:
         FDOC.oprint("Output from FDOC.oprint():")
         for line in outstr.split('\n'):
             FDOC.oprint(line)

         # Also the coprint method is available to send output
         # to the FRED output window as well as the local
         # python buffer
         FDOC.coprint("Output from FDOC.coprint():")
         for line in outstr.split('\n'):
             FDOC.coprint(line)
```

```
Output from FDOC.coprint():
Entity 0: System
Entity 1: Optical Sources
Entity 2: Geometry
Entity 3: Analysis Surface(s)
```

1.2.2 Active update/retrieval of the General File Comments

`FDOC.comment` will also instantly set/retrieve the document comment field:

```
In [15]: FDOC.comment = '''Here we have some file
         comments that might be used for documentation
         and other useful descriptive text'''
```

Check in the GUI: 'Edit :: General File Comments' If you change the General File Comments in the GUI it will be reflected here "up to date". Try changing the text and then run the next cell

```
In [16]: # After modifying the File Comments: Edit -> General File Comments
         # Run this cell and the changes in the GUI are reflected here
         print(FDOC.comment)
```

```
Here we have some file
comments that might be used for documentation
and other useful descriptive text

... some comments added from the GUI.
```

1.2.3 Active update/retrieval of the Document Units

Document units are also available. Use `FDOC.units`:

```
In [17]: print("Current units are: {}".format(FDOC.units))
         FDOC.units='nm'
         print("We can change the document units: {}".format(FDOC.units))
```

```
Current units are: mm
We can change the document units: nm
```

In the GUI, check that the SysUnit value has changed in the bottow right of the workspace window. Also it is updated in 'Tools :: Units & Scaling'.

Change the value in 'Tools :: Units & Scaling' back to 'mm' and run the next cell. Changes in the GUI are updated here.

```
In [18]: print("Current units are: {}".format(FDOC.units))
```

```
Current units are: mm
```

1.2.4 Access to the Entities

`FDOC.entities` can be used to query the state of the document entities but it cannot update them:

```
In [19]: ents = FDOC.entities # create a shortcut
         print('FDOC.entities.count informs us there are {} '
               'entities in the document.'.format(ents.count))
         print("Their names and descriptions are available from the .names and .descriptions at
         for i in range(FDOC.entities.count):
             print("\tName: {} (ID:{})\n\t  Description: {}".format(ents.names[i], i, ents.desc
```

```
FDOC.entities.count informs us there are 4 entities in the document.
Their names and descriptions are available from the .names and .descriptions attributes:
        Name: System (ID:0)
          Description: System
        Name: Optical Sources (ID:1)
          Description:
        Name: Geometry (ID:2)
          Description:
        Name: Analysis Surface(s) (ID:3)
          Description:
```

1.2.5 struct

The `FDOC.struct` method provides access to all of the FRED datastructures (i.e. `T_<STRUCTNAME>`). They'll be passed to FRED by value so it's fine to define a few global instances and modify them as need be before sending them to FRED.

```
In [20]: # Entity data structure
         ENT = FDOC.struct('T_ENTITY')
         # Primitive operations data structure
         OP = FDOC.struct('T_OPERATION')
```

Member access is accomplished using python 'dotted' notation. The following is a typical process for adding an element with a surface.

```
In [21]: # Add a custom element
         ENT.traceable = True
         ENT.name = 'Custom Element'
         elem_id, elem_ent = DOBJ.AddCustomElement(ENT)
         DOBJ.Update() # Sync the GUI with the API

In [22]: # Add a conic surface
         ENT.name = 'Conic Surface'
         ENT.parent = elem_id
         OP.Type = 'ShiftZ'
         OP.val1 = 1.0
         conic_id, conic_ent = DOBJ.AddConic(ENT, -0.5, 0)
         shift_op = DOBJ.AddOperation(conic_id, OP)
         DOBJ.Update()
```

Hit `ALT-Z` in your FRED window to zoom fit the view.

1.2.6 SimplePlane

You may encapsualte geometry creation into python libraries as you see fit. For example, the `geom` module demonstrates a `SimplePlane` class:

```
In [23]: from pyfred import geom

         # Use geom.SimplePlane to make a plane surface
         plane1 = geom.SimplePlane(FDOC, parent=elem_id, color='red')
```

Now with this pythonic instance of the plane, you may dynamically update its properties at a high level without having to call `DOBJ.Update()`:

```
In [24]: plane1.width = 4.
         plane1.height = 2.
         plane1.color = 'yellow'
```

1.2.7 Application objects

`FDOC.app` provides raw access to the w32 Application object. Browse the available commands from the IPython shell:

```
In [1]: FDOC.app.<TAB>
```

It's often handy to have it as a global variable:

```
In [25]: APP = FDOC.app

         # Test out APP.Asin against numpy arcsin
         x0 = APP.Asin(0.5)
         x1 = np.arcsin(0.5)
         print("FRED function Asin(0.5):    {}".format(x0))
         print("Numpy function arcsin(0.t): {}".format(x1))
         assert(np.isclose(APP.Asin(0.5), np.arcsin(0.5)))
         print("PASS")

FRED function Asin(0.5):    0.5235987755982989
Numpy function arcsin(0.t): 0.5235987755982989
PASS
```

1.3 utils module

Have a look through the pyfred\utils.py module for some potentially useful utilities:

```
In [26]: from pyfred import utils as u

In [1]: u.<TAB>

In [27]: probj(u)

acos            acosdg          asin            asindg
atan            atan2           atan2dg         atandg
cos             cosdg           d2r             degrees
magnitude       math            move_x          move_y
move_z          negate_vect     norm            normvect
np              pi              r2d             radians
sin             sindg           sqrt            tan
tandg           vectangle       xy_tilt         xz_tilt
yz_tilt


In [28]: print("There are some trig shortcuts:")
         ang = 60.
         print("  u.cosdg({}) = {:g}".format(ang, u.cosdg(ang)))
         assert(np.isclose(u.cosdg(ang), np.cos(np.deg2rad(ang))))
         print("PASS\n")

         print("... some vector operations:")
         v = (2., 10., 11.)
         print("  u.magnitude({}) = {:g}".format(v, u.magnitude(v)))
         assert(np.isclose(u.magnitude(v), np.sqrt(v[0]**2+v[1]**2+v[2]**2)))
         print("PASS\n")

         print("... and translation convenience functions:")
```

```
        u.move_z(FDOC, plane1.objid, 1.0)
        print("  Plane1 shifted by Z={}".format(plane1.OPS.elements[-1].val1))
        assert('ShiftZ' == plane1.OPS.elements[-1].Type)
        assert(np.isclose(1.0, plane1.OPS.elements[-1].val1))
        print("PASS\n")

There are some trig shortcuts:
  u.cosdg(60.0) = 0.5
PASS

... some vector operations:
  u.magnitude((2.0, 10.0, 11.0)) = 15
PASS

... and translation convenience functions:
  Plane1 shifted by Z=1.0
PASS
```

1.4 Camera handling

Initialize a Camera instance for manipulating the 3D View:

```
In [29]: cam = pyfred.Camera(FDOC)
         DOBJ.Update() # Update the document

In [30]: # Underneath the Camera() instance is a
         # FRED T_CAMERA datastructure
         print("The camera datastructure is:\n{}\n".format(repr(cam)))

         # The string representation of the instance
         # reports the properties nicely formatted
         print("Current camera settings:\n{}".format(cam))

The camera datastructure is:
com_struct(xLoc=-11.523056553262094, yLoc=3.3654297275890204, zLoc=1.4254518547729207, xAim=0.0

Current camera settings:
xLoc: -11.523056553262094
yLoc: 3.3654297275890204
zLoc: 1.4254518547729207
xAim: 0.0
yAim: 0.0
zAim: 0.5
xUp: 0.2692417586861648
yUp: 0.9553470644548315
zUp: -0.12174096195248998
```

```
In [31]: # You can muck around with the camera in the GUI.
         # Then we can see the camera (x,y,z) location, set it
         # directly with a triplet and it updates immediately
         print("Current camera coordinates are: {}".format(cam.location))
         cam.location=(-.25, .1, .25)
         print("New camera coordinates are: {}".format(cam.location))
```

```
Current camera coordinates are: (-11.523056553262094, 3.3654297275890204, 1.4254518547729207)
New camera coordinates are: (-0.25, 0.1, 0.25)
```

Use ALT-Z to re-zoom as necessary.

```
In [32]: # Same for the camera aiming
         print("Current camera aims at: {}".format(cam.aim))
         cam.aim=(0., .05, .05)
         print("New camera aim: {}".format(cam.aim))
         # Also the pointing vector from the aim point to the
         # camera is available
         print("Camera vector: {}".format(cam.pointvect))
```

```
Current camera aims at: (0.0, 0.0, 1.0)
New camera aim: (0.0, 0.05, 0.05)
Camera vector: [-11.8152   4.6761 -10.8652]
```

```
In [33]: # .aim_origin will conveniently re-aim the camera at the origin
         cam.aim_origin
         # .dist reports the camera distance from it's aim
         print("Camera distance from aim point: {:0.4f}".format(cam.dist))
```

```
Camera distance from aim point: 16.7004
```

```
In [34]: # We can also retrieve/adjust the up-vector
         print("Current camera up vector is: {}".format(cam.upvect))
         cam.upvect=(1, 0, 0)
         print("New camera up-vector: {}".format(cam.upvect))
```

```
Current camera up vector is: (0.38222399589359135, 0.9239665661340697, -0.01380585490115721)
New camera up-vector: (0.7067332032878372, 0.28329116393115483, -0.6482856591110464)
```

```
In [35]: # Since 'y is up' is so common, .view_yup is available
         cam.view_yup
         print("New camera up-vector: {}".format(cam.upvect))
```

```
New camera up-vector: (0.20874426764168183, 0.9591222504460617, 0.19107678934402225)
```

In [36]:
```
# There's additional convenience properties for standard views:
# .view_front .view_back .view_top .view_bottom
# .view_left .view_right
views = ['front', 'back', 'top', 'bottom', 'left', 'right']
for v in views:
    time.sleep(1)
    getattr(cam, 'view_{}'.format(v))
```

In [37]:
```
# .view_iso() will provide 8 possible isometric views
for v in range(8):
    time.sleep(1)
    cam.view_iso(v + 1)
```

In [38]:
```
# .view_zen() lets us set the view in spherical coordinates
for az in range (0, 180, 40):
    for zen in range(5, 85, 10):
        cam.view_sph(zen=zen, az=az)
```