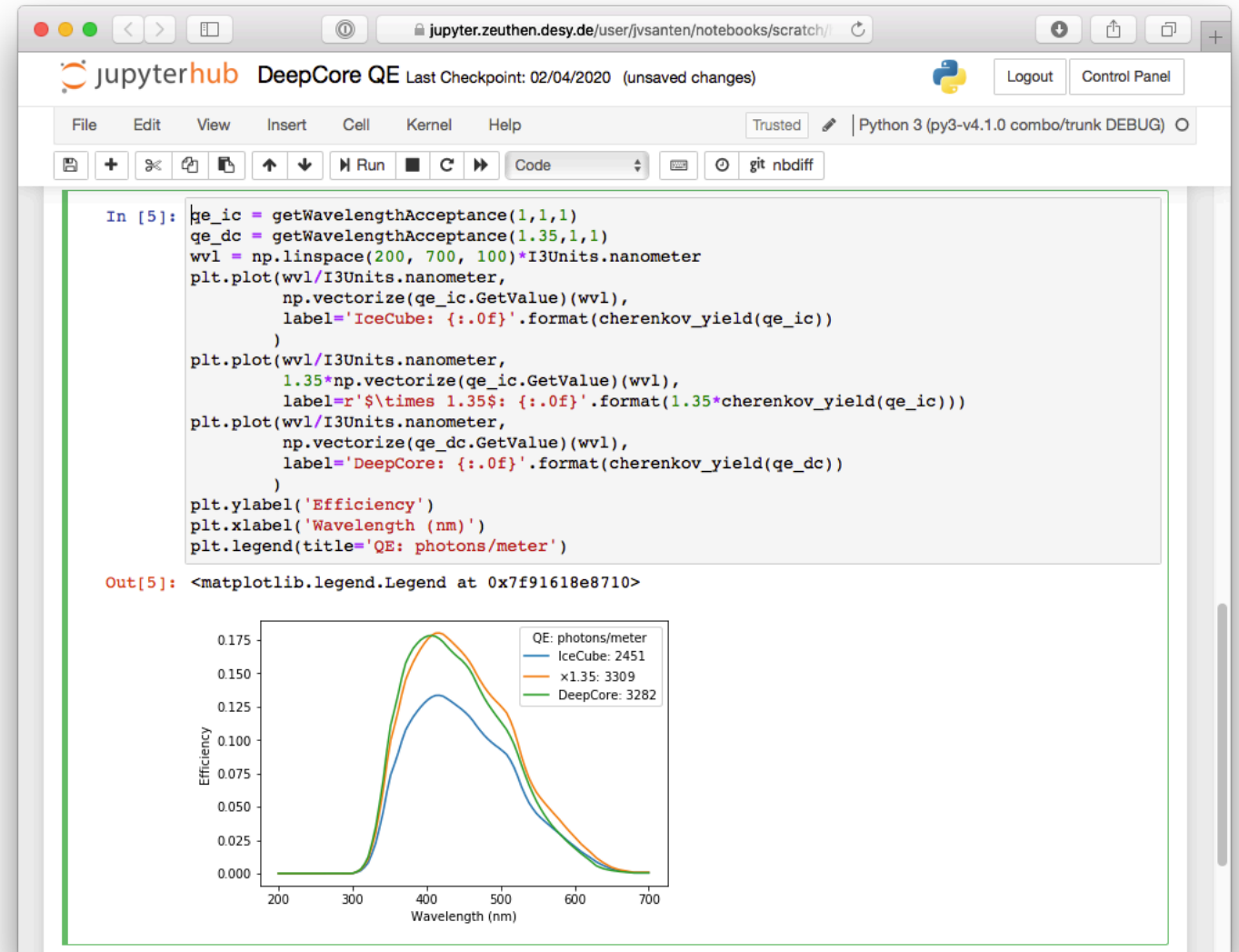


Introduction to Jupyter

Jakob van Santen
2020-02-25

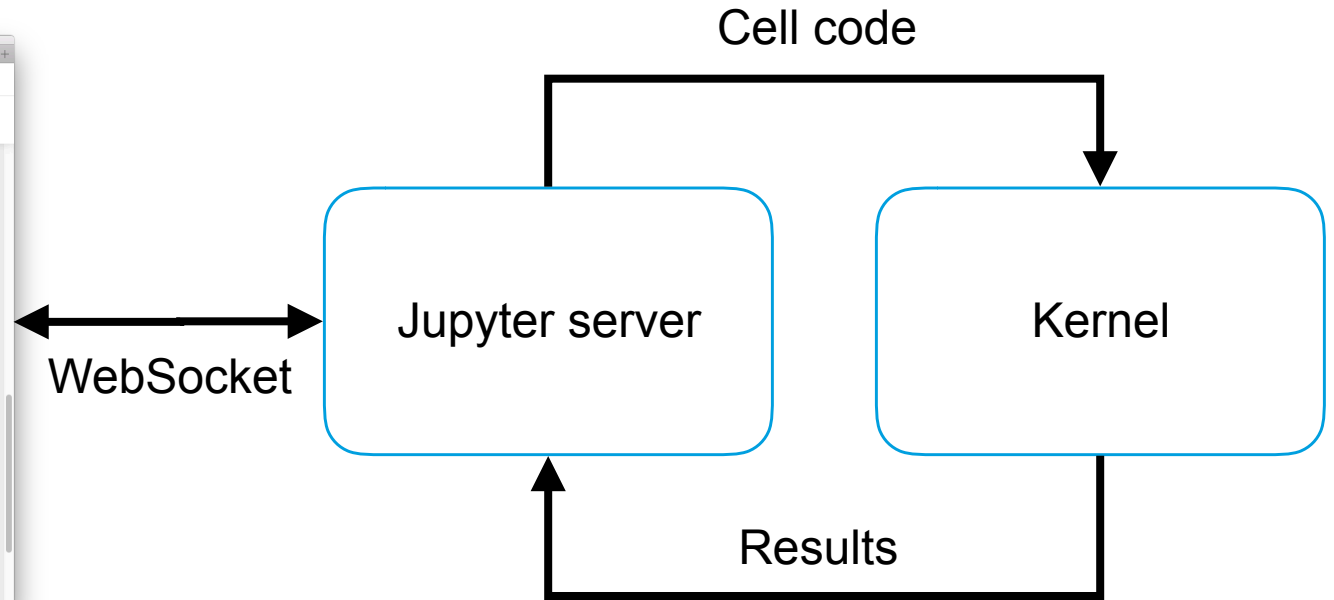
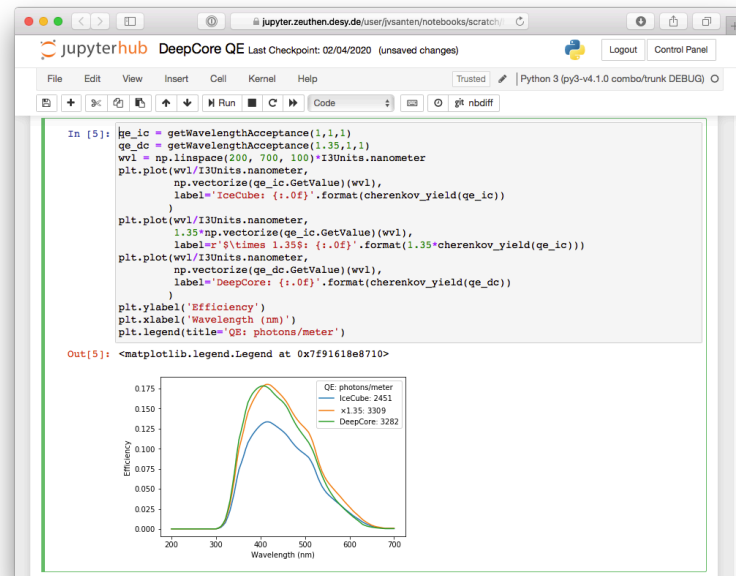
Jupyter in a nutshell

- **Jupyter Notebook** is a web application that lets you write, edit, and execute code snippets (cells) and view the results in the browser
- Results can be anything a web browser can display:
 - static images
 - richly formatted tables
 - [interactive] animations
- Built-in help
 - Context-aware tab-completion for variables, functions, file names
 - Access to function docstrings
- Open source (BSD)



Jupyter components

Client



Server can be local (on your laptop) or remote (on a WGS)

Kernels exist for nearly any language

Notebook elements

Code and results

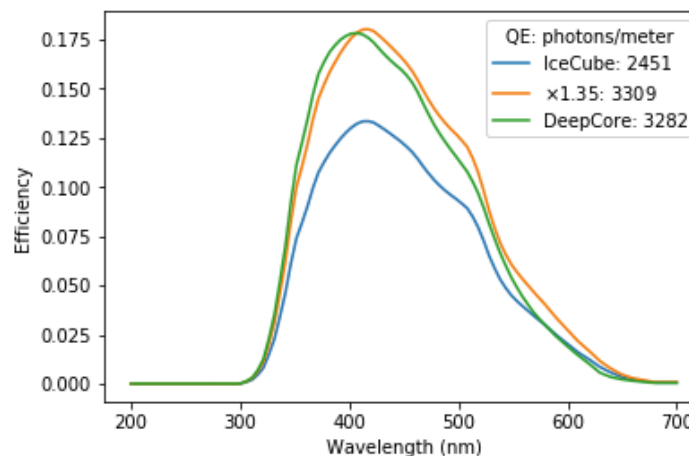
Input
(can include magics)

Evaluation sequence

Matplotlib figure (inline backend)
These can also be animated!
(see demo)

```
In [5]: qe_ic = getWavelengthAcceptance(1,1,1)
qe_dc = getWavelengthAcceptance(1.35,1,1)
wvl = np.linspace(200, 700, 100)*I3Units.nanometer
plt.plot(wvl/I3Units.nanometer,
         np.vectorize(qe_ic.GetValue)(wvl),
         label='IceCube: {:.0f}'.format(cherenkov_yield(qe_ic))
        )
plt.plot(wvl/I3Units.nanometer,
         1.35*np.vectorize(qe_ic.GetValue)(wvl),
         label=r'$\times 1.35$: {:.0f}'.format(1.35*cherenkov_yield(qe_ic))
        )
plt.plot(wvl/I3Units.nanometer,
         np.vectorize(qe_dc.GetValue)(wvl),
         label='DeepCore: {:.0f}'.format(cherenkov_yield(qe_dc))
        )
plt.ylabel('Efficiency')
plt.xlabel('Wavelength (nm)')
plt.legend(title='QE: photons/meter')
```

Out[5]: <matplotlib.legend.Legend at 0x7f91618e8710>



Notebook elements

Inline text

- Explain and document your steps in text cells
- Formatting with [GitHub-flavored] Markdown (headings, bullet lists, links, etc)
- Wrap LaTeX expressions in $...$ for inline rendering with MathJax

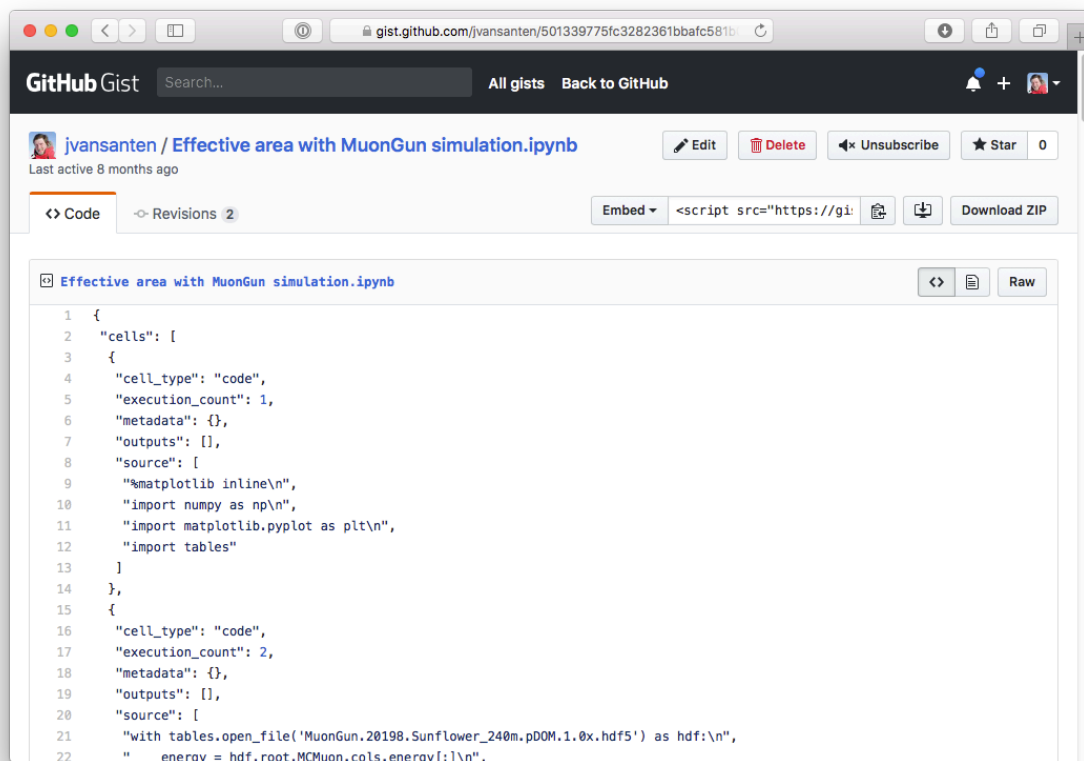
```
In [2]: with tables.open_file('MuonGun.20198.Sunflower_240m.pDOM.1.0x.hdf5') as hdf:
        energy = hdf.root.MCMuon.cols.energy[:]
        cos_zen = np.cos(hdf.root.MCMuon.cols.zenith[:])
        area_weight = hdf.root.MuonEffectiveArea.cols.value[:]
```

Events in these datasets were grouped by muon energy for purposes of resource efficiency. You do not necessarily need to care about this if all files are present. However, when some files are missing, you see very obvious holes in the energy distribution. If you didn't account for them, it would appear that the effective area is smaller in bins that partially overlap with a missing energy range.

You can correct for this by ensuring that your bins never have this partial overlap, so the number of events in the bin will either be the expected number or 0. The simplest solution is just to use the production binning, but this is too coarse, especially at low energies. The `snap_to_valid_regions` function below takes a user-supplied binning and removes regions that are expected to have zero events.

NB: this is exactly the same problem as constructing a charge-vs-time histogram for a DOM with some time windows excluded, as implemented in Millipede.

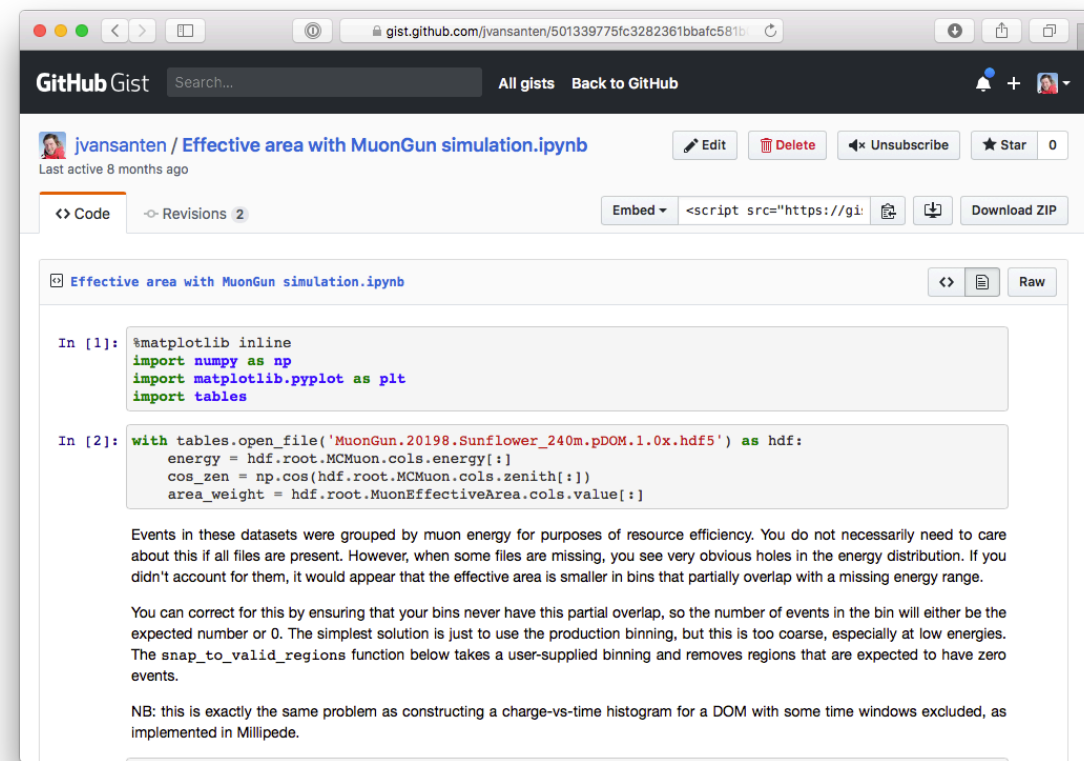
Notebooks are just JSON



The screenshot shows a GitHub Gist page for a file named "Effective area with MuonGun simulation.ipynb". The "Code" tab is selected, displaying the JSON structure of the notebook cell. The JSON includes metadata like "cell_type": "code", "execution_count": 1, and "source" which contains the Python code for importing libraries and opening a file.

```
1 {
2   "cells": [
3     {
4       "cell_type": "code",
5       "execution_count": 1,
6       "metadata": {},
7       "outputs": [],
8       "source": [
9         "%matplotlib inline\n",
10        "import numpy as np\n",
11        "import matplotlib.pyplot as plt\n",
12        "import tables"
13      ]
14    },
15    {
16      "cell_type": "code",
17      "execution_count": 2,
18      "metadata": {},
19      "outputs": [],
20      "source": [
21        "with tables.open_file('MuonGun.20198.Sunflower_240m.pDOM.1.0x.hdf5') as hdf:\n",
22        "    energy = hdf.root.MCMuon.cols.energy[:]\n",
```

Keep them in version control!



The screenshot shows the same GitHub Gist page, but with the "Code" tab selected. The JSON is rendered into a readable Python code block. Below the code, there is explanatory text about the data and a note about the binning process.

```
In [1]: %matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import tables

In [2]: with tables.open_file('MuonGun.20198.Sunflower_240m.pDOM.1.0x.hdf5') as hdf:
        energy = hdf.root.MCMuon.cols.energy[:]
        cos_zen = np.cos(hdf.root.MCMuon.cols.zenith[:])
        area_weight = hdf.root.MuonEffectiveArea.cols.value[:]
```

Events in these datasets were grouped by muon energy for purposes of resource efficiency. You do not necessarily need to care about this if all files are present. However, when some files are missing, you see very obvious holes in the energy distribution. If you didn't account for them, it would appear that the effective area is smaller in bins that partially overlap with a missing energy range.

You can correct for this by ensuring that your bins never have this partial overlap, so the number of events in the bin will either be the expected number or 0. The simplest solution is just to use the production binning, but this is too coarse, especially at low energies. The `snap_to_valid_regions` function below takes a user-supplied binning and removes regions that are expected to have zero events.

NB: this is exactly the same problem as constructing a charge-vs-time histogram for a DOM with some time windows excluded, as implemented in Millipede.

Display them on GitHub/GitLab
(or from any URL with <https://nbviewer.jupyter.org>)

Use cases

- **Rapid prototyping:** explore the API of an unfamiliar library, test code snippets
- **Exploratory data analysis:** load data, make plots of all imaginable combinations
- **Plot tweaking:** load data once, refine figures iteratively
- **Education:** distribute tutorials as notebooks with inline documentation and expected results
- **Interactive dashboards:** run code and display results based on user input ([examples](#))
- likely many more...

Where did this come from?

- 2001: iPython 0.0.1, a Python shell startup script
 - some predefined physical constants and units
 - line return values stored in special variables (`_`, `_1`, `_2`, etc)
- 2011: iPython 0.12 introduces iPython Notebook
 - single process -> client/server model
 - Inline graphics in iPython Notebook and Qt console
- 2014: Notebook front-end, file format, and kernel protocol factored out into Project Jupyter
 - Multi-language support: R, Julia, C++, etc (even Mathematica!)
- 2018: JupyterLab
 - Multi-window user interface
 - Plug-in system (git integration, HDF5 browser, etc)

iPython 0.0.1

```
8. python
(standard)[jakob@znb34:2020/Data Science Club]$ PYTHONSTARTUP=ipython-0.0.1.py python
Python 2.7.16 (default, Dec 3 2019, 02:03:47)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.31)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.

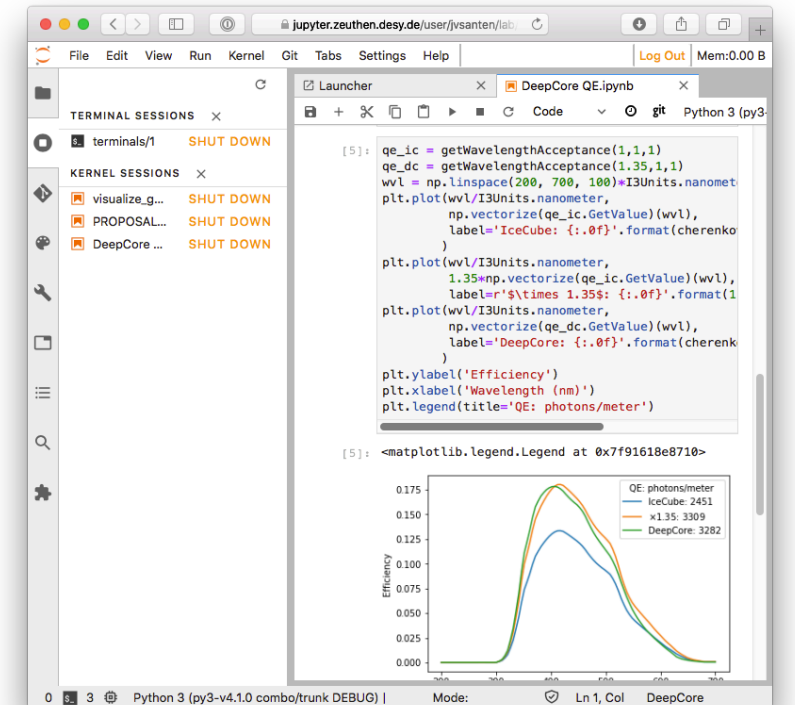
Python 2.7.16 (default, Dec 3 2019, 02:03:47)

Interactive Python -- Type intro() for a brief explanation.

In[1]:=
```



JupyterLab 1.2.6



Best practices

- Distinguish between “scratch pad” and “notebook”
 - A scratch pad is temporary: experiment, evaluate cells out of order, throw away
 - Notebooks are software: should be documented, readable, and reproducible
 - A notebook is not always the best tool for the job
- For notebooks you want to keep and share:
 - Reevaluate cells in order before saving
 - Check in to version control!

Where to get it

- Self-hosted: install miniconda, then jupyterlab
- Hosted at DESY: jupyter.zeuthen.desy.de (see demo by Philipp)

Questions & demo