Jupyter notebook advice

February 22, 2023

1 Efficient coding and collaboration with Jupyter notebooks

This notebook contains four ideas on how to best utilize jupyter notebooks when coding and delivering results.

1.1 Why notebooks are great

Jupyter notebooks are great for quickly testing code to check if it does what it is supposed to do. For example when working with NumPy.

[1]:
```
import numpy as np

n = 4
p = 1
A = np.ones((n,n))
A[p:n-p,p:n-p] = np.zeros((n-2*p,n-2*p))
A
```

[1]:
```
array([[1., 1., 1., 1.],
       [1., 0., 0., 1.],
       [1., 0., 0., 1.],
       [1., 1., 1., 1.]])
```

1.2 Why notebooks are not that great

Jupyter notebooks are not that great when your code reaches a certain level of complexity, with a lot of variables laying around and you are running cells in different orders.

If the second cell below is run twice in a row, things will break down.

[2]:
```
n = 10
p = 3

def get_padded_matrix(n,p):

    """
    Input:
        - n: integer for the dimension of your matrix
        - p: integer for the width of the padding
```

```
    Output:
        - A: (n,n) matrix with a p-width layer of zeros around the boundary
             and ones everywhere else.
    """

    A = np.zeros((n,n))
    A[p:n-p,p:n-p] = np.ones((n-2*p,n-2*p))
    return A
```

[3]:
```
get_padded_matrix(n,p)
p = 6
```

[4]:
```
n = 14
get_padded_matrix(n,p)
```

[4]:
```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

1.3 If others are going to read your code: document it!

The docstring, i.e. the `""" text """` in the function `get_padded_matrix(n,p)` will be printed if you run `help(get_padded_matrix)`

[5]:
```
help(get_padded_matrix)
```

```
Help on function get_padded_matrix in module __main__:

get_padded_matrix(n, p)
    Input:
        - n: integer for the dimension of your matrix
        - p: integer for the width of the padding

    Output:
        - A: (n,n) matrix with a p-width layer of zeros around the boundary
```

and ones everywhere else.

1.4 Move working code to a separate .py file

When you have finised developing your functions, move it to separate `.py` files and import it to the notebook. Some more input on python modules here.

Why is this a goode idea?

1. Assuming you give intuitive names for the `.py` files, this gives your code a better structure that is easier to understand and navigate for others.
2. If you are using git to sync code, it is much easier to handle `.py` files (for merge conflicts etc.) than `.ipynb` files.

Now, if you change something in the external `.py` file, you have to restart the notebook and import the modules to get the updated functions imported to the notebook. However, there is a hack. Importing the jupyter autoreload extension makes sure that new changes are imported every time you run `from my_code import my_function`, without having to reload the whole notebook.

In short, run

```
%reload_ext autoreload
%autoreload
```

before importing functions from external `.py` files.

[6]:
```
%reload_ext autoreload
%autoreload

#Assuming we have a file called "matrix_operations.py"
#in the same folder as the jupyter notebook with a "get_padded_matrix(n,p)"␣
 ↪function

from matrix_operations import get_padded_matrix

n = 8
p = 2

get_padded_matrix(n,p)
```

[6]:
```
array([[0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 1., 1., 1., 0., 0.],
       [0., 0., 1., 1., 1., 1., 0., 0.],
       [0., 0., 1., 1., 1., 1., 0., 0.],
       [0., 0., 1., 1., 1., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0.]])
```