Introduction to Jupyter and IPython

Week 1 Wednesday

Miles Chen, PhD

Jupyter Lab and Jupyter Notebooks

There are many ways to run Python.

You can run it directly in the Python interpreter in interactive mode. This is generally not recommended for doing anything other than checking a few values or expressions.

Another option is to use a text editor to write scripts and to run the scripts from the command line.

Others use an integrated development environment (IDE) like PyCharm.

Jupyter Lab is an browser-based IDE centered around Notebooks. Jupyter Notebooks have combine markdown and Python code to create a document, much like an R-Markdown file and is a popular choice for data science work.

Jupyter comes pre-installed with the Anaconda distribution. You can launch Jupyter Lab from the prompt by typing in `jupyter lab` . Alternatively, you can launch Jupyter Notebook alone with `jupyter notebook`

Jupyter has two modes:

command mode

- Type `Esc` to enter command mode

edit mode

- Type `Enter`/`Return` to enter edit mode

Don’t mix up command mode and edit mode.

Keyboard Shortcuts

You really need to learn and use the keyboard shortcuts if you want to gain speed and proficiency. Try to do as much as you can without touching your mouse or trackpad.

Adding Cells (command mode)

- Type **b** to add a new cell below your current cell.
- Type **a** to add a new cell above your current cell.

Deleting Cells (command mode)

- Type **dd** to delete a cell. That is type the letter d twice.

Keyboard Shortcuts

Cut, Copy, Paste (command mode)

While you have a cell selected in command mode, you can use

- **x** to cut
- **c** to copy
- **v** to paste (it will paste the cell below the selected cell)

Don't hold ctrl, just type the letter.

Navigating (command mode)

- You can use the up or down arrows to switch cells.
- If you don't want to leave your home row, you can also use **j or k** to move up and down.

Jupyter has three types of cells:

- Markdown Cells
    - used for text
- Code Cells
    - used to run code
- raw cells (used infrequently)

While in command mode, you can convert a cell to markdown by typing **m**

You can convert a cell to code by typing **y**

heading level 1 starts with `#`

heading level 2 starts with `##`

heading level 3 starts with `###`

heading level 4 starts with `####`

heading level 5 starts with `#####`

heading level 6 starts with `######`

# if you try to use 7 `#` symbols, it becomes normal text

Markdown basics

https://commonmark.org/help/tutorial/

Use # symbols to indicate headings.

```
# When rendered, this would be a top level heading.
```

Make bulleted lists by using hyphens

```
- item 1
- item 2
```

If you want to include code, you can indicate to markdown not to render by using he back accent ` and the beginning and end of one line, or use three tildes ~ to indicate a code chunk.

Use asterisks *for emphasis. Put on asterisk around a word or phrase to* \`italicize* `it. Use two asterisks to make it` **bold**\`.

Include math by using dollar signs. One dollar sign for in-line math symbols. Two dollar signs for stand-alone math equations.

For example, you can talk about π using in-line math.

You write an equation like

```
$$E = mc^2$$
```

Or a more complex equation like:

```
$$\phi(x) = \frac{1}{\sigma \sqrt{2 \pi}} \exp{
    \left( \frac{-1}{2} \left ( \frac{x - \mu}{\sigma} \right )^2 \right)} $$
```

Equation:

$$E = mc^2$$

$$\phi(x) = \frac{1}{\sigma \sqrt{2 \pi}} \exp{ \left(\frac{-1}{2} \left (\frac{x - \mu}{\sigma} \right)^2 \right)}$$

Running or Rendering Cells

You can run a code cell with **Shift+Enter**. This will run the selected cell and then advance to the next cell. If you are at the last cell, it will insert another cell below it.

If you want to run the current cell but do not want to advance to the next cell, use **Ctrl+Enter**. This will run the currently selected cell only.

Alt-Enter or **Option+Enter** renders/runs the current cell and inserts a new cell below it.

IPython

Jupyter runs on IPython.

As you write code in a code cell, you can use some of the nice features of IPython

In [1]:

```
# basic operations
2 + 3
```

Out[1]:

```
5
```

In [1]:

```
# basic operations
2 + 3
```

Out[1]:

```
5
```

In [2]:

```
# if you have multiple operations in a cell, only the last expression will be printed
4 + 5
5 * 5
```

Out[2]:

```
25
```

In [1]:

```
# basic operations
2 + 3
```

Out[1]:

```
5
```

In [2]:

```
# if you have multiple operations in a cell, only the last expression will be printed
4 + 5
5 * 5
```

Out[2]:

```
25
```

In [3]:

```
# if you want to the other stuff to appear, you need to use print() commands
print(4 + 5)
print(5 * 5)

# note that when you do this, there is no Out[ ] for the cell.
```

```
9
25
```

Features of IPython: `In` and `Out`

IPython cells are preceeded by an In[1] or an Out[1].

These show the sequence you write code, but also allows you to access past entries and values

Features of IPython: `In` and `Out`

IPython cells are preceeded by an In[1] or an Out[1].

These show the sequence you write code, but also allows you to access past entries and values

```
In [4]:
In[1]
```

```
Out[4]:
'# basic operations\n2 + 3'
```

Features of IPython: `In` and `Out`

IPython cells are preceeded by an In[1] or an Out[1].

These show the sequence you write code, but also allows you to access past entries and values

```
In [4]:  In[1]
```

```
Out[4]: '# basic operations\n2 + 3'
```

```
In [5]:  Out[2]
```

```
Out[5]: 25
```

In [6]:

```
# Out is a dictionary
Out
```

Out[6]:

```
{1: 5, 2: 25, 4: '# basic operations\n2 + 3', 5: 25}
```

In [6]:

```
# Out is a dictionary
Out
```

Out[6]:

```
{1: 5, 2: 25, 4: '# basic operations\n2 + 3', 5: 25}
```

In [7]:

```
# In is a list
In
```

Out[7]:

```
['',
 '# basic operations\n2 + 3',
 '# if you have multiple operations in a cell, only the last
expression will be printed\n4 + 5\n5 * 5',
 '# if you want to the other stuff to appear, you need to us
e print() commands\nprint(4 + 5)\nprint(5 * 5)\n\n# note tha
t when you do this, there is no Out[ ] for the cell.',
 'In[1]',
 'Out[2]',
 '# Out is a dictionary\nOut',
 '# In is a list\nIn']
```

”math operations” with strings

”math operations” with strings

In [8]:

```
# multiplication repeats the string
3 * 'hello!'
```

Out[8]:

```
'hello!hello!hello!'
```

”math operations” with strings

In [8]:

```
# multiplication repeats the string
3 * 'hello!'
```

Out[8]:

```
'hello!hello!hello!'
```

In [9]:

```
# addition concatenates strings
"hi" + "goodbye"
```

Out[9]:

```
'higoodbye'
```

Executing Scripts from Jupyter

You can use the `%run` to execute python scripts stored in separate files. For example, I have a simple script that simply prints hello world stored in a script

Executing Scripts from Jupyter

You can use the `%run` to execute python scripts stored in separate files. For example, I have a simple script that simply prints hello world stored in a script

In [10]:
```
%run script01.py
```

```
hello world
```

Accessing variables defined in the notebook:

If you want the script to have access to variables that you have defined in the notebook, use `%run -i`

Accessing variables defined in the notebook:

If you want the script to have access to variables that you have defined in the notebook, use `%run -i`

In [11]:

```
name = 'Miles'
```

Accessing variables defined in the notebook:

If you want the script to have access to variables that you have defined in the notebook, use `%run -i`

```
In [11]: name = 'Miles'
```

```
In [12]: %run script02.py
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most re
cent call last)
~\OneDrive\Teaching\21\2022-fa-stats21\script02.py in <modul
e>
----> 1 print('hello ' + name)
      2 print("It's Friday!")

NameError: name 'name' is not defined
```

Accessing variables defined in the notebook:

If you want the script to have access to variables that you have defined in the notebook, use `%run -i`

In [11]:
```
name = 'Miles'
```

In [12]:
```
%run script02.py
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\OneDrive\Teaching\21\2022-fa-stats21\script02.py in <module>
----> 1 print('hello ' + name)
      2 print("It's Friday!")

NameError: name 'name' is not defined
```

In [13]:
```
%run -i script02.py
```

```
hello Miles
It's Friday!
```

Using bash/shell/cmd commands within a jupyter notebook

You can use bash commands like `cat` which displays the contents of a file by preceeding the command with an exclamation point. The commands you can use will be different for windows or unix based (mac) machines. For example, there is no `cat` command in windows, and you must use `type`

Using bash/shell/cmd commands within a jupyter notebook

You can use bash commands like `cat` which displays the contents of a file by preceeding the command with an exclamation point. The commands you can use will be different for windows or unix based (mac) machines. For example, there is no `cat` command in windows, and you must use `type`

In [14]:

```
!cat script02.py
```

```
'cat' is not recognized as an internal or external command,
operable program or batch file.
```

Using bash/shell/cmd commands within a jupyter notebook

You can use bash commands like `cat` which displays the contents of a file by preceeding the command with an exclamation point. The commands you can use will be different for windows or unix based (mac) machines. For example, there is no `cat` command in windows, and you must use `type`

In [14]:

```
!cat script02.py
```

```
'cat' is not recognized as an internal or external command,
operable program or batch file.
```

In [15]:

```
!type script02.py
```

```
print('hello ' + name)
print("It's Friday!")
```

The last output value

You can access the last value output using a single underscore character _

The last output value

You can access the last value output using a single underscore character _

```
In [16]:
3 * 9
```

```
Out[16]:
27
```

The last output value

You can access the last value output using a single underscore character _

```
In [16]: 3 * 9
```

```
Out[16]: 27
```

```
In [17]: _ - 2
```

```
Out[17]: 25
```

The last output value

You can access the last value output using a single underscore character _

```
In [16]: 3 * 9

Out[16]: 27

In [17]: _ - 2

Out[17]: 25

In [18]: _

Out[18]: 25
```

Help

Help

```
In [19]:
help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, fl
ush=False)

    Prints the values to a stream, or to sys.stdout by defau
lt.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the curr
ent sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a n
ewline.
    flush: whether to forcibly flush the stream.
```

Important Notes about Python Syntax

based on A Whirlwind Tour of Python by Jake VanderPlas

Comments Are Marked by `#`

Important Notes about Python Syntax

based on A Whirlwind Tour of Python by Jake VanderPlas

Comments Are Marked by `#`

In [20]:

```
# this is a comment and is not run
```

Lines

The end of a line terminates a statement. No need for using a semi-colon to end a statement ; although you can optionally use the semi-colon to write two statements in one line.

If you want to have a single statement cover multiple lines, you can use a backslash \ or encase the statement in parenthesis. If you are defining a list or other data structure that already uses some sort of bracket, this is handled automatically.

Lines

The end of a line terminates a statement. No need for using a semi-colon to end a statement ; although you can optionally use the semi-colon to write two statements in one line.

If you want to have a single statement cover multiple lines, you can use a backslash \ or encase the statement in parenthesis. If you are defining a list or other data structure that already uses some sort of bracket, this is handled automatically.

In [21]:

```
# examples
x = 5
print(x)
```

```
5
```

Lines

The end of a line terminates a statement. No need for using a semi-colon to end a statement ; although you can optionally use the semi-colon to write two statements in one line.

If you want to have a single statement cover multiple lines, you can use a backslash \ or encase the statement in parenthesis. If you are defining a list or other data structure that already uses some sort of bracket, this is handled automatically.

In [21]:

```
# examples
x = 5
print(x)
```

```
5
```

In [22]:

```
# semicolon to include multiple statements in one line
y = 6; z = 7
print(y + z)
```

```
13
```

Lines

The end of a line terminates a statement. No need for using a semi-colon to end a statement ; although you can optionally use the semi-colon to write two statements in one line.

If you want to have a single statement cover multiple lines, you can use a backslash \ or encase the statement in parenthesis. If you are defining a list or other data structure that already uses some sort of bracket, this is handled automatically.

In [21]:

```python
# examples
x = 5
print(x)
```

```
5
```

In [22]:

```python
# semicolon to include multiple statements in one line
y = 6; z = 7
print(y + z)
```

```
13
```

Having multiple statements in one line is generally considered bad style and should be

In [23]:

```
# backslash to continue a statement over multiple lines
a = 1 + 2 + 3 \
    + 4 + 5
print(a)
```

```
15
```

In [23]:

```python
# backslash to continue a statement over multiple lines
a = 1 + 2 + 3 \
    + 4 + 5
print(a)
```

```
15
```

In [24]:

```python
# or use parenthesis
b = (1 + 2 + 3
    + 4 + 5)
print(b)
```

```
15
```

In [23]:

```
# backslash to continue a statement over multiple lines
a = 1 + 2 + 3 \
    + 4 + 5
print(a)
```

```
15
```

In [24]:

```
# or use parenthesis
b = (1 + 2 + 3
    + 4 + 5)
print(b)
```

```
15
```

In [25]:

```
l = ['a', 2, 3, 'd',
    'e', 6]
print(l)
```

```
['a', 2, 3, 'd', 'e', 6]
```

Indentation defines code blocks

Python does not use curly braces `{}` to define code blocks. IPython is smart enough to automatically indent lines after you use a colon `:` which indicates that the following lines are part of a code block.

We haven't covered conditionals yet, but I'll introduce them here briefly to show how code blocks work.

Indentation defines code blocks

Python does not use curly braces `{}` to define code blocks. IPython is smart enough to automatically indent lines after you use a colon `:` which indicates that the following lines are part of a code block.

We haven't covered conditionals yet, but I'll introduce them here briefly to show how code blocks work.

In [26]:

```
# we will learn if statements later, but here's an example
x = 8
if(x > 5):
    print('x is greater than 5')    # the two indented lines only run
    print(x)                        # when the if statement is true
print('hello')    # this line is not indented and will run regardless of if statement
```

```
x is greater than 5
8
hello
```

In [27]:

```
x = 4
if(x > 5):
    print('x is greater than 5')   # the two indented lines only run
    print(x)                       # when the if statement is true
print('hello')    # this line is not indented and will run regardless of if statement
```

```
hello
```

In [28]:

```
x = 4
if(x > 5):
    print('x is greater than 5')
print(x)
print('hello')
```

```
4
hello
```