WA2715 Applied Data Science with Python

Student Labs

Web Age Solutions Inc.

Table of Contents

Lab 1 - Using Jupyter Notebook

Jupyter (note the spelling of the word) is a browser-based development environment that is an evolution of the original IPython command-line REPL. It is driven by the built-in web server that can be securely accessed remotely as well (a fantastic feature for collaborative development and work sharing).

Your Jupyter projects are represented by a Notebook which is mix of your code and meta-information associated with the code (like comments) that may help you track your train of thought while you are experimenting with code. You can export your code as a regular Python file that you can then execute from command line. You can load external scripts as well.

Jupyter uses the concept of a kernel which is, essentially, a specific language run-time. In addition to Python, Jupyter supports kernels for R, Go, C#, etc. (https://github.com/jupyter/jupyter/wiki/Jupyter-kernels)

Jupyter environment can be augmented with thousands of additional plugins, like spell checkers, etc.

Like in IPython, you can get immediate help on commands and/or functions using the **?** operator, like so:

```
import numpy as np
np.st*?
```

which will print a list of functions that start with **st**

To get the Docstring help of a function, use the full function name with the **?** next to it, e.g.

```
np.std?
```

Note: Just quickly review (do not execute them for now as we will do it a bit later on) the following steps needed to get started with Jupyter.

To start a Jupyter session, use the command prompt to navigate to the directory from which you want to run it and enter this command:

```
# Do not execute this command for now this is just for illustration!
jupyter notebook
```

The above command will go ahead and start a new Jupyter interactive editor session that is opened in your web browser.

Each notebook you create in your session will be backed up by an Interactive Python Notebook file with extension *.ipynb* saved in the current working directory (from which you started Jupyter).

In this lab, you will learn the basic Jupyter interactive session commands.

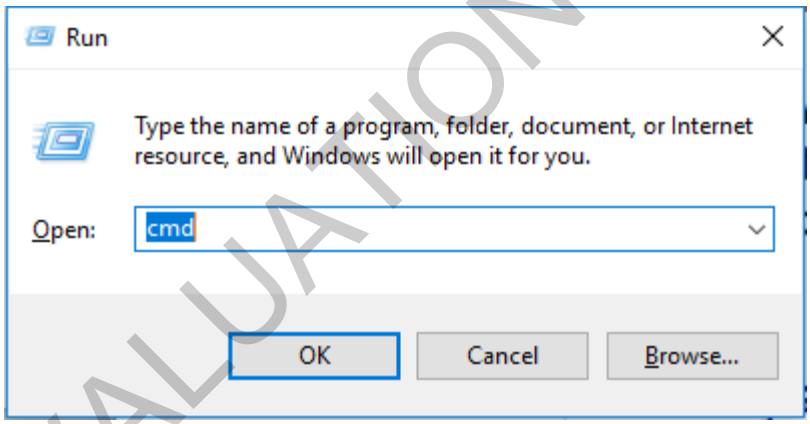
For this course we will be using Chrome.

> **Note**
>
> If you have a problem while working on a lab in the virtual class delivery environment, let the instructor know the following three pieces of information:
>
> 1. The lab number (or the lab title)
>
> 2. The lab part (printed in the lab as **Part # - <Part Name>**, e.g. **Part 3 - Read the Input File)**
>
> 3. The lab step number (lab step numbers are prefixed with two underscores, e.g. **_2**)

Part 1 - Set up the Environment

__1. Start the command prompt.

A fast way to start the command prompt in any modern version of Windows ® is to press the **Win** + **R** key combination and then type **cmd** in the open window and press **Enter**



__2. In the command prompt window that opens, create a new working directory **c:\Works** and then change directory to it using this commands:

```
mkdir c:\Works
cd c:\Works
```

> **Note**: All the subsequent labs, unless specified otherwise, will be done using this, **c:\Works,** directory.

Part 2 - Check the Versions of Key Modules

__1. Enter the following command in the command prompt window:

```
conda list
```

You should see a listing of installed packages.

> **Note:** Conda is an open source package management system for dependency and environment management for any language—Python, R, Ruby, Lua, Scala, Java, JavaScript, C/ C++, FORTRAN [https://conda.io/docs/]. Conda supports the following commands (without quotes):
>
> 'clean', 'config', 'create', 'help', 'info', 'install', 'list', 'package', 'remove', 'uninstall', 'search', 'update', 'upgrade'

__2. Scroll up to see the versions of the scikit-learn and numpy packages:

```
scikit-learn                0.19.1
numpy                       1.14.0
```

They should be equal or greater than the ones listed above.

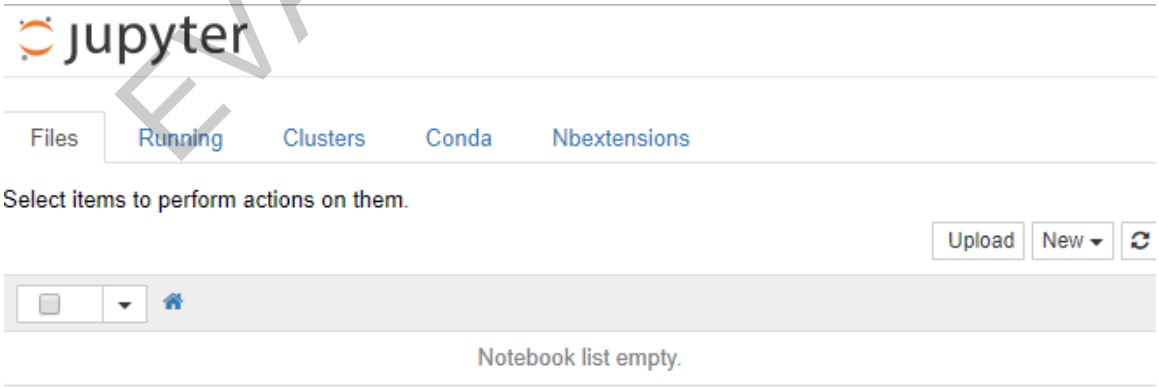
Part 3 - Start and Work in a New Jupyter Notebook

__1. In the command prompt, enter the following command:

```
jupyter notebook
```

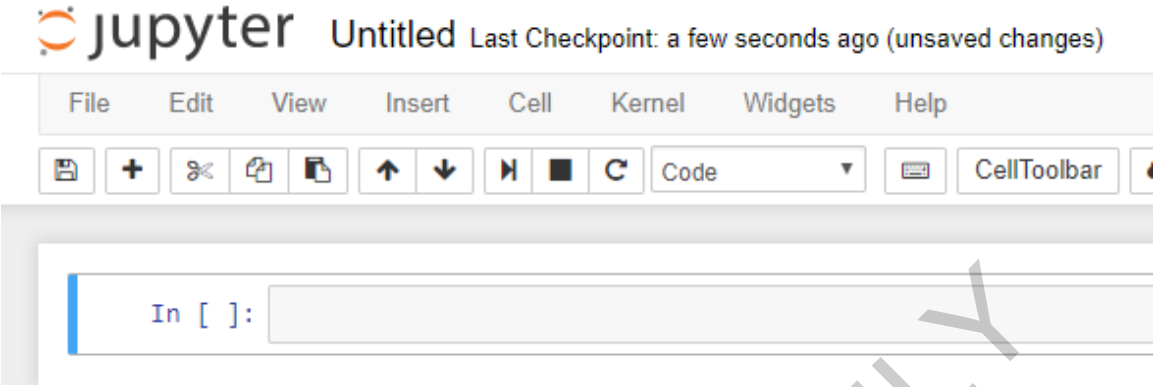
Wait for the back-end web server to start.

A new browser window should open displaying an empty Jupyter notebook.



__2. In the **New** drop-down, select **Python 3** to create a new notebook:

The new *Untitled* notebook should initialize.



Note: If you have multiple versions of Python installed, you can specify the one you want to use.

__3. In the current input cell with a blue border on the left, enter the following command:

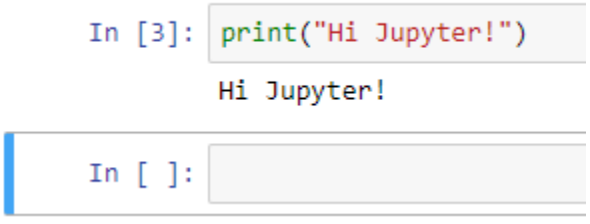
```
print("Hi Jupyter!")
```

Notice, that as you started entering the command, the color of the border changed to green.

The Jupyter Notebook has two different keyboard input modes: *Edit mode* and *command mode*. **Edit mode** allows you to type code/text into a cell and is indicated by a **green** cell border. **Command mode** binds the keyboard to notebook level actions and is indicated by a grey cell border with a **blue** left margin. You enable **command mode** by pressing **Esc**. Edit mode gets enabled when you press **Enter** or click inside the cell.

__4. Press **Shift+Enter**

You should get the *print's* argument, **Hi Jupyter!** printed just below the cell and a new command cell opened below.

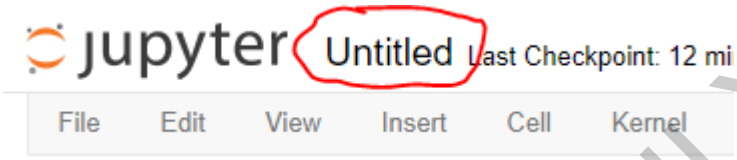


Note: When you read in subsequent lab instructions *enter* or *submit command(s)*, you should enter the provided command(s) in the new cell and use the **Shift+Enter** key combination to submit the command(s) for execution.

To get help on Jupyter short-cuts, select the **Help > Keyboard Shortcuts** in the menubar.

Let's rename the notebook.

__5. Click the **Untitled** label in the top left hand-side corner.



__6. In the **Rename Notebook** dialog that pops up, enter **Getting started with Jupyter**

__7. Press **Rename**.

Spend some time navigating the Jupyter browser-based development environment.

Let's see how you can save the current notebook as a Python file.

__8. In the menubar, select to the **File > Download as > Python (.py)**

You will be prompted to confirm the download and save command (the prompt depends on your browser).

__9. Confirm your intended operation (by clicking the **Keep**, or **OK** button, depending on the browser).

The file named **Getting+started+with+Jupyter.py** should be saved in the current user's *Downloads* directory.

If you open the file in your text editor, you should see the following content (yours may be slightly different):

```
# coding: utf-8

# In[3]:

print("Hi Jupyter!")


# In[ ]:
```

In our subsequent work, we will be using Python 3.6, let's quickly verify the version of the installed Python.

__10. In Jupyter, enter the following command:

```
import sys
sys.version
```

__11. Remember to press **Shift+Enter**

You should see the following output:

```
3.6.4  <other information is omitted ...>
```

Part 4 - Clean Up

In the menu bar, select **File > Close and Halt**

The command will shutdown the notebook's kernel (the sandboxed Python session) and close the interactive edit session.

Keep the browser window open as we are going to use it later.

This is the last step in this lab.

Part 5 - Review

In this lab, you learned the basics of the Jupyter development environment.

Lab 2 - Python with NumPy and pandas

This lab aims at helping you refresh your knowledge of Python and show how Python integrates with NumPy and pandas libraries.

> **Note**
>
> If you have a problem while working on a lab in the virtual class delivery environment, let the instructor know the following three pieces of information:
>
> 1. The lab number (or the lab title)
>
> 2. The lab part (printed in the lab as **Part # - <Part Name>**, e.g. **Part 3 - Read the Input File)**
>
> 3. The lab step number (lab step numbers are prefixed with two underscores, e.g. **_2**)

Part 1 - Set up the Environment

We are going to re-use the Jupyter working session from the previous lab.

If you have your environment shut-down, start the command prompt and change to the **c:\Works** directory (follow the steps in the **Part 1 - Set up the Environment** from the **Using Jupyter Notebook** lab) , then start a new Jupyter working session by running this command:

```
jupyter notebook
```

Wait for the Jupyter notebook page to open in the browser.

__1. In the **New** drop-down box in the top right-hand corner of the Jupyter Home page, select **Python 3** to create a new notebook.

A new notebook, called *Untitled,* should initialize.

__2. Rename the notebook as **Python with NumPy and pandas Lab**

Part 2 - Setting up Imports

__1. In the currently active input cell, enter the following commands one after another pressing **Enter** after each line and **Shift+Enter** to submit the command batch for execution:

```
import numpy as np
import pandas as pd
```

The import aliases **np** and **pd** are standard for the imported NumPy and pandas libraries, which you should adhere to as well.

Part 3 - Python Refresher

In this lab part, we will review parts of Python that will help you understand and complete subsequent labs. Mind you that this is not, by any means, an introduction to Python and the instructions below assume you already have some familiarity with the language … or that you are able to pick up another language as you go (Python, at its core, is a rather simple language).

So let's get started ...

Python naming convention for variable names is lowercase with words separated by underscores, e.g. *this_is_my_variable*. The *mixedCase* is also allowed in contexts where that's already the prevailing used style.

The None Type

In Python, the closest to null in other languages is **None** which is a singleton object equal to another **None**.

__1. Enter the following command:

```
type(None)
```

You should see the following output:

```
NoneType
```

__2. Enter the following command:

```
None == None
```

You should see the following output:

```
True
```

You would usually use the identity operator **is** to check if the object carries the *None* value:

```
n = None
n is None   # True
```

Version of packages is very important. Here is how you can quickly check a package version.

Printing a Package Version

__3. Enter the following command:

```
np.__version__
```

You should see this output (your version may be higher):

```
'1.14.0'
```

The Integer Division

There were some fundamental changes in operations between Python 2 and Python 3.

One of the biggest one is the handling of the integer division.

Python 2 will treat 7/8 as an integer division, returning 0 (performing truncation). Python 3 will convert the factors to float and return a float result (0.875). If you need a-la Python 2 type of integer division, use two slashes //, e.g. 7//8.

String Formatting

Python 2.6+ introduced simple string formatting operator:

```
s = 'Mr. {1} is {0} years old'.format (45,'Smith')
```

The **s** variable will be assigned:

```
'Mr. Smith is 45 years old'
```

If you omit indexes (1 and 0) in the placeholders'{}' like so:

```
s = 'Mr. {} is {} years old'.format (45,'Smith')
```

You will have this interpolated string:

```
'Mr. 45 is Smith years old'
```

You may need to swap the actual parameters (*45,'Smith'*) to get what you may really want.

String substringing

For string substringing, Python uses the array notation treating characters in a string as elements in an array.

__4. Enter the following commands:

```
i = '1234567'
i[:3]
```

You should see the following output:

```
'123'
```

Try to guess what output will be for the *i[3:]* command.

__5. Enter the following command to get the last element in the array:

```
i[-1]
```

You should see the following output:

```
'7'
```

Getting the last element is a frequently used operation in processing Machine Learning data sets as the last element of an array (or the last column in a matrix) is often reserved for the target variable that you want to predict with your model.

And the predictor variables, which come before that last element, can be accessed in an array like so (leaving our the last element):

```
i[:-1]
```

Loops

Here is the common for-loop cycling idiom in Python.

__6. Enter the following commands:

```
for k in range(5):
    print (k)
```

You should see the following output:

```
0
1
2
3
4
```

Note. In Python 3, *range()* took over from the *xrange()* function (which was more memory efficient for large ranges) and the latter was dropped.

The *range()* supports the start (with the default of 0), stop, and step (default is 1) parameters.

Easy Numbers

Python 3.6 introduced the user-friendly large number notation, e.g.

```
f = 1_222_333.99
```

which will be interpreted by Python as a floating-point number of *1222333.99*.

Note. Python 2 had the *int* and *long* types.

Python 3 has only *int*. Essentially, *long* was renamed to *int*. So Python 3 has only one built-in integral type, named *int* which behaves mostly like the old *long* type.

The zip Function

The *zip()* function allows you to iterate over lists passed to it as parameters (you may have two or more lists).

__7. Enter the following commands:

```
x = [1,2,3,4,5]
y = [10,20,30,40,50]
z = ['a','b','c','d','e']

[z + '->' + str(x) + ':' + str(y) for x, y, z in zip(x,y,z)]
```

You should see the following output (of type list):

```
['a->1:10', 'b->2:20', 'c->3:30', 'd->4:40', 'e->5:50']
```

List Comprehensions

Comprehensions are constructs that allow sequences to be built from other sequences. Python 2.0 introduced list comprehensions and Python 3.0 extended this functionality to work with dictionaries and sets.

Usually, you use list comprehensions to capture indexes of array elements that satisfy a predicate.

Here is how it works.

__8. Enter the following command:

```
[ i**2 for i in range (1, 5) if (i % 2 == 0) ]
```

You should see the following output (we start with 1, not the default 0):

```
[4, 16]
```

It works as follows:

First, the *for i in range (1, 5)* loop operation kicks in. Then, for each **i**, the *if (i % 2 == 0)* predicate is evaluated. If *i* satisfies the predicate condition (*i* is even, in our case), i is made available to the power operation (i ** 2).

Basically, the input sequence (*range(1, 5)*) is translated into the output list [4, 16].

Part 4 - The NumPy Library

At the core of NumPy is its "ndarray" data structure, which is a n-dimensional array.

__1. Enter the following commands:

```
python_list = [1,2,3,4,5]
np_array = np.array([python_list])
```

np_array is of type *numpy.ndarra.*

NumPy's *array()* function takes a Python list (a rather inefficient bloated data structure) as its input and converts it into a small-footprint nimble data structure using native C-language bridge.

NumPy has a function similar to Python's *range* called *arange()* which acts as a generator of an arithmetic progression sequence.

__2. Enter the following command:

```
np.arange(12)
```

You should see the following output:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```

The above array is a simple one-dimensional structure of type *numpy.ndarray*.

__3. Enter the following command:

```
np.arange(12).shape
```

The above command will return a tuple that confirms that we have a one-dimensional array:

```
(12,)
```

Let's build a 3x4 matrix (3 rows by 4 columns) from the above one-dimensional array.

__4. Enter the following command:

```
np.arange(12).reshape(3,4)
```

You should see the following output:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

The *shape* property called on it would show this value (packaged as a tuple):

```
(3,4)
```

NumPy supports a wide spectrum of linear algebra operations, like matrix multiplication (through the *.dot()* method), and the like.

NumPy makes it easy to filter elements in a *numpy.ndarray* subject to a predicate as well as generate boolean arrays.

__5. Enter the following command:

```
np_array % 2 == 0
```

You should see the following output (which is a boolean array):

```
array([[False,  True, False,  True, False,  True]])
```

__6. Enter the following command (which acts as a filter on the target array):

```
np_array [np_array <= 3]
```

You should see the following output (the result of a filtering operation):

```
array([1, 2, 3])
```

Part 5 - The pandas Library

pandas is an open source, BSD-licensed library that provides high-performance data structures and data analysis tools for the Python programming language. The center piece of *pandas* is the DataFrame object around which the whole *pandas*' functionality is built.

In this lab part, we will illustrate the main operations involving the DataFrame object. We will also demonstrate the common idioms related to generating data sets for proof-of-concept Data Science projects.

First, we will create a small 5x2 matrix that we will use as data for a DataFrame object.

Our DataFrame object will have two columns: *Sales* and *Location* that represent sales figures per location for some fictitious retail business.

__1. Enter the following commands to generate some sales numbers (expressed in, say, million dollars) to be used in the *Sales* column:

```
np.random.seed(10)
sales = np.array((100 * np.random.rand (5)).astype(int))
sales
```

You should see the following output:

```
array([77,  2, 63, 74, 49])
```

The *np.random.seed(10)* command is the common way to seed (set the initial value of) the random generator so that we can reproduce the results at a later time using the same seed value.

Here is how we can generate the *Location* column values.

__2. Enter the following commands:

```
location = []
for i in range(5):
    location.append('ABCDE'[i])

location
```

You should see the following output:

```
['A', 'B', 'C', 'D', 'E']
```

Now, let's combine both arrays into a matrix.

__3. Enter the following commands:

```
data = np.array([sales, location]).T
data
```

Notice how we transposed (swapped rows for columns) the data matrix using the **.T** operator.

You should see the following output:

```
array([['77', 'A'],
       ['2', 'B'],
       ['63', 'C'],
       ['74', 'D'],
       ['49', 'E']], dtype='<U11')
```

Also notice that the integer values in the first columns have been turned into the String type -- you can often observe this behavior when you read data from a file. We will restore the integer type of the first column shortly.

__4. Enter the following commands:

```
df = pd.DataFrame(data, columns = ["Sales", "Location"])
df
```

Notice how we add the column names to be used in the DataFrame object. Without this configuration, panda's DataFrame will assume the column names as ordinal values of 0 and 1.

You should see the following output:

	Sales	Location
0	77	A
1	2	B
2	63	C
3	74	D
4	49	E

In essence, a DataFame object is similar to a relational database table or an Excel spreadsheet.

Now let's fix the Sales' column type.

__5. Enter the following command:

```
df.dtypes
```

You should see the following output:

```
Sales       object
Location    object
dtype: object
```

String is disguised as an Object.

Here is the common idiom for changing a DataFrame's column type to the desired one.

__6. Enter the following command:

```
df.Sales = df.Sales.astype(int)
```

If you repeat the *df.dtypes* command now, you should be able to see the following output:

```
Sales       int32
Location    object
dtype: object
```

For numeric types, pandas offers a summary function called *describe()*.

__7. Enter the following command:

```
df.describe()
```

You should see the following output listing a number of useful descriptive statistics' metrics like the count of non-None values, the mean, and such like:

```
           Sales
count      5.000000
mean       53.000000
std        30.553232
min        2.000000
25%        49.000000
50%        63.000000
75%        74.000000
max        77.000000
```

To get on-line help on a function in Jupyter notebook (functionality delegated to the underlying IPython engine), place a question mark '?' in front of the command (there are some other variations where and when you can place the ? sign).

```
?df.describe
```

Here is how you can access values in a DataFrame column (similar to how you can access a column in the *SELECT col FROM Table* SQL statement).

__8. Enter the following command (name auto-completion on DataFrame columns is supported):

```
df.Location
```

You should see the values in the *Location* column.

__9. Enter the following command:

```
df.Sales
```

The above command is functionally equivalent to the command that uses the *loc* index accessor, namely:

```
df.loc[:, 'Sales']
```

Now, let's say you want to see the first three values in the *Location* column of your data frame, here is how you can do this:

__10. Enter the following command:

```
df.loc[:2, 'Location']
```

The index 2 is inclusive (the actual projected indexes are 0, 1, 2).

If you want to see the values in both columns in the rows 1, 2, 3, the command will be:

```
df.loc[1:3, :]
```

The last column (:) is kind of a wild card (*) operator applied to all the columns in the DataFrame object.

If you want to take a peek at the last row's values, use the *iloc* index accessor.

__11. Enter the following command using the *iloc* index accessor:

```
df.iloc[-1, :]
```

You should see the following output:

```
Sales       49
Location     E
Name: 4, dtype: object
```

To find the minimum in a numeric column, use this command:

```
<your_data_frame>.<column_name>.min()
```

Same command can be used for finding the maximum value; you can also use the *describe()* method to this end as well.

__12. Enter the following command to sort the values in the *Sales* column in descending order:

```
df.Sales.sort_values(ascending=False)
```

You should see the following output:

```
0    77
3    74
2    63
4    49
1     2
Name: Sales, dtype: int32
```

Notice how row ids (listed as the first column in the output) have been changed accordingly.

The *df.Sales.sort_values()* command will sort (the operation does not update data in-place – it is not mutating) the DataFrame object in ascending order.

Now let's demonstrate how we can add and drop a column in a DataFrame object.

__13. Enter the following command to create a new '*Month*' column:

```
df['Month'] = pd.DataFrame(['Jan', 'Feb', 'Mar', 'Apr', 'May'])
```

__14. Enter the following command:

```
df
```

The result of the above command will be this DataFrame object;

```
          Sales     Location  Month
0         77        A         Jan
1         2         B         Feb
2         63        C         Mar
3         74        D         Apr
4         49        E         May
```

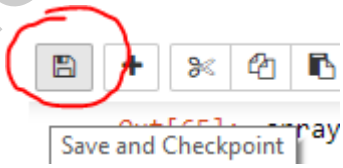
__15. Enter the following command to drop the column we just created:

```
del df['Month']
```

We are almost done in this lab.

Part 6 - Clean-Up

__1. Click the **Save** button in the toolbar.



__2. In the menu bar, select **File > Close and Halt**

The command will shutdown the notebook's kernel (the sandboxed Python session) and close the interactive edit session.

Keep the browser window open as we are going to use it later.

This is the last step in this lab.

Part 7 - Review

In this lab, you refreshed your memory (or learned) several things you need to know about Python; you also learned (or refreshed you memory) how to use NumPy and pandas libraries.