[Global 88] Introduction to Jupyter, Python, and Gender Violence in India

Professor: Karenjot Bhangoo Randhawa

Estimated Time: *50 minutes*
Notebook Created By: Carlos Calderon, Bella Chang, Emily Guo
Code Maintenance: Carlos Calderon

Welcome! This an introductory notebook in many topics. First, we introduce the Jupyter Notebook environment and how to navigate it. Then, we give a brief crash course on some elementary topics in Python. Lastly, we begin to start looking at table operations using datasets on Indian states and cities, and more specifically, containing data on violence against women.

Learning Outcomes

By the end of this notebook, students will be able to:

1. How to work with Jupyter Notebooks (adding cells, restarting kernel, etc.)
2. Understand basic concepts in Python and programming.
3. Perform basic table operations using the `datascience` library.

Table of Contents

1. Introduction to Jupyter

Welcome to the Jupyter Notebook, an interactive and web-based notebook that is used in data science to document and visualize data! The Jupyter notebook supports regular text as well as programming code.

1.1 Text Cells

In a notebook, each rectangle is called a *cell*. Each *cell* or chunk of text can either take on the form of code or regular text.

The way that we determine the type of the cell is by clicking "Cell," located at the top of your screen, and then clicking "Cell Type." From there you will see an arrow that offers the options of changing the current cell you are working in to **Markdown** (which is just regular text) or **Code**.

One of the ways to create a cell is to click the **+** button in the menu bar found at the top of the notebook. By default, a code cell is generated. To change the cell type from **Code** to **Markdown (Text)**, select the cell and click the dropdown menu (the second rightmost button in the top menu) and change the cell type.

Make sure to choose accordingly depending on your different tasks later down the road; cells can only support the type that they were assigned (for example, a **Markdown** type cell will not run code).

Each cell can be composed of text or code. Text cells are written in a formatting language called Markdown. This is how you will be answering a portion of the questions in future assignments.

You can click the ► button or hold down `shift` + `return` to confirm any changes.

Question 1.1: This paragraph is in its own cell. Create a new cell below this one. Make sure it is a Markdown (text) cell, then write up a small intro about yourself. It can be any sort of introduction you'd like.

1.2 Code Cells

Other cells can contain code. For this class, all of the code will be written in the Python 3.x programming language. Running a code cell will execute the code that it contains.

To run a code cell, click the cell. You will notice that it becomes highlighted with either a blue/green rectangle. Then, press the ►or hold `shift` + `return`. Basically, you can run any type of cell with either option.

Question 1.2: The cell below is a markdown cell. Change it to a code cell and run it to produce the code output. Do not worry about understanding the code for now.

print("Hello world")

1.3 Errors

Adapted from Lab 1: Expressions in Data 8

Python is a language, and like natural human languages, it has rules. It differs from natural language in two important ways:

1. The rules are simple. You can learn most of them in a few weeks and gain reasonable proficiency with the language in a semester.
2. The rules are *rigid*. If you're proficient in a natural language, you can understand a non-proficient speaker, glossing over small mistakes. A computer running Python code is not smart enough to do that.

Whenever you write code, you'll make mistakes. When you run a code cell that has errors, Python will sometimes produce error messages to tell you what you did wrong.

Errors are okay; even experienced programmers make many errors. When you make an error, you just have to find the source of the problem, fix it, and move on.

We have made an error in the next cell. Run it and see what happens.

```
In [ ]: print("This line is missing something."
```

Note: In the toolbar, there is the option to click `Cell > Run All`, which will run all the code cells in this notebook in order. However, the notebook stops running code cells if it hits an error, like the one in the cell above.

You should see something like this (minus our annotations):

Error

The last line of the error output attempts to tell you what went wrong. The syntax of a language is its structure, and this `SyntaxError` tells you that you have created an illegal structure. "`EOF`" means "end of file," so the message is saying Python expected you to write something more (in this case, a right parenthesis) before finishing the cell.

There's a lot of terminology in programming languages, but you don't need to know it all in order to program effectively. If you see a cryptic message like this, you can often get by without deciphering it. (Of course, if you're frustrated, ask a neighbor or a staff member for help.)

Try to fix the code above so that you can run the cell and see the intended message instead of an error.

1.4 The Kernel

Adapted from Lab 1: Expressions in Data 8

The kernel is a program that executes the code inside your notebook and outputs the results. In the top right of your window, you can see a circle that indicates the status of your kernel. If the circle is empty (○), the kernel is idle and ready to execute code. If the circle is filled in (●), the kernel is busy running some code.

Next to every code cell, you'll see some text that says In [...]. Before you run the cell, you'll see [ ]. When the cell is running, you'll see In [*]. If you see an asterisk (*) next to a cell that doesn't go away, it's likely that the code inside the cell is taking too long to run, and it might be a good time to interrupt the kernel (discussed below). When a cell is finished running, you'll see a number inside the brackets, like so: In [1]. The number corresponds to the order in which you run the cells; so, the first cell you run will show a 1 when it's finished running, the second will show a 2, and so on.

You may run into problems where your kernel is stuck for an excessive amount of time, your notebook is very slow and unresponsive, or your kernel loses its connection. If this happens, try the following steps:

1. At the top of your screen, click Kernel, then Interrupt.
2. If that doesn't help, click Kernel, then Restart. If you do this, you will have to run your code cells from the start of your notebook up until where you paused your work.
3. If that doesn't help, restart your server. First, save your work by clicking File at the top left of your screen, then Save and Checkpoint. Next, click Control Panel at the top right. Choose Stop My Server to shut it down, then Start My Server to start it back up. Then, navigate back to the notebook you were working on. You'll still have to run your code cells again.

1.5 Importing Packages

In data science and programming as a whole, we often make use of packages (often references as library or modules) that contain powerful functions not found in the default Python library. To use them, we need to import them into our notebook using the `import` Python statement. For most of this course, we will be using the `numpy` and `datascience` libraries, alongside other visualization tools. The cell below is an example of how these statements look. You don't have to worry much about understanding the code or how it works. **Just make sure to first run the cell containing all of the import statements before running any code cells. Not doing so may result in pesky errors in the code.**

```
In [30]: # Don't change this cell; just run it.
import numpy as np
from datascience import *
```

2. Python Basics

Python is a programming language, thus, it allows us to communicate with the computer. Python has a specific syntax and a set of common practices, just like any other language. Before we dive into our data, let's go over some of these basic functionalities in Python that you will be using in this lesson.

2.1 Print Statements and Comments

For the computer to physically print out words, we are able to use the print statement. Using the command **`print()`**, we can print anything we want the computer to say. It's also important that we write the words we want to print in between quotes. This can be in the form of `"..."` or `'...'`, however, make sure to *always open and close with either double quotes or single quotes*. Opening with a single quote and closing with a double quote will result in errors. See below for an example:

```
In [ ]: print("I am a good print statement.")
```

Question 2.1: Create a new code cell below this one, and print out the statement "Hello, world! My name is _____," filling out the blank with your name.

A comment in Python is used to annotate portions of code, and are not executed by Python. Instead, they are not evaluated and thus can be treated as text within a code cell. To make a comment, we use the `#` notation, as shown below.

```
In [11]: # This cell is the same as the cell above.

print("I am a good print statement.")

I am a good print statement.
```

2.2 Math Expressions and Names (Variables)

In addition to printing text, Python can be used to compute basic arithmetic expressions. Given that quantative information is found everywhere in data science, it is important to get familiarized with basic math operations in python. As in regular math, we can encapsulate operations in parantheses to instruct Python to compute these first. The following cells will walk through a series of small examples showcasing some of the math operations we can do.

```
In [2]: # Here is a simple example. Notice how Python functions like regular math, performing operations in the parantheses first.
(2 * 4) + 3

Out[2]: 11
```

```
In [3]: # Notice how the values changed. Why do you think that is?
2 * (4 + 3)

Out[3]: 14
```

Note: You may have noticed that we did not have to print these values to see them in our output. This is because a Jupyter notebook implicitly prints out the value of the last line in a code cell.

The following is adapted from Lab 1: Expressions in Data 8

In natural language, we have terminology that lets us quickly reference very complicated concepts. We don't say, "That's a large mammal with brown fur and sharp teeth!" Instead, we just say, "Bear!" In Python, we do this with ***assignment statements***. An assignment statement has a name on the left side of an `=` sign and an expression to be evaluated on the right. This expression is now *bound* to the name we assigned it (the value on the **left** side of the `=`). The following cell computes the expression `(2*4)+3` on the right hand side, and assigns that value to the name `total`. An alternate nomenclature for "name" is the word "variable". For the rest of the notebook and the rest of this class, we will use the "variable" terminology to reference a Python name.

```
In [5]: total = (2 * 4) + 3
```

Now that we have run that cell, we can now reference that variable (or name) by "calling" it as shown in the cell below. The output will be the expression that was assigned to the variable name.

```
In [6]: total

Out[6]: 11
```

A common pattern in Jupyter notebooks is to assign a value to a name and then immediately evaluate the name in the last line in the cell so that the value is displayed as output.

```
In [7]: num_seconds_in_min = 60
num_seconds_in_min

Out[7]: 60
```

Another common pattern is that a series of lines in a single cell will build up a complex computation in stages, naming the intermediate results.

```
In [9]: num_seconds_in_min = 60
num_minutes_in_hour = 60
num_seconds_in_hour = num_seconds_in_min * num_minutes_in_hour
num_hours_in_day = 24
num_seconds_in_day = num_seconds_in_hour * num_hours_in_day
num_seconds_in_day

Out[9]: 86400
```

Question 2.2: In the code cell below, assign `num_days_in_week` to the number of days in a week. Then, use `num_days_in_week` and the variable `num_seconds_in_day` to assign the variable `num_seconds_in_week` to the total number of seconds in a week.

```
In [ ]: # Fill in your code by replacing the "..."
num_days_in_week = ...
num_seconds_in_week = ... * ...
num_seconds_in_week
```

Throughout your work, it is important to know that in Python, variable names:

- Can have letters (upper- and lower-case letters are both okay and count as different letters),
- Can have underscores,
- Can have numbers **BUT,** the first character can't be a number (otherwise a name might look like a number).
- Cannot contain spaces, since spaces are used to separate pieces of code from each other. Instead, use the underscore `_` to separate words in a variable name.

Other than those rules, what you name something doesn't matter to Python. For example, this cell does the same thing as the cell we saw above, except everything has a different name. Notice how this code is not as easy to understand. Keep this in mind when coming up with variable names, as they should roughly describe what they are assigned to (i.e.: `num_seconds_in_min` used to reference the number of seconds in a minute.)

```
In [10]: a = 60
b = 60
c = a * b
d = 24
e = d * c
e

Out[10]: 86400
```

2.3 Functions and Calling Functions

Adapted from Lab 1: Expressions in Data 8

The most common way to combine or manipulate values in Python is by calling functions. Python comes with many built-in functions that perform common operations. Functions are also bound, or assigned, to variable names. To use these functions, we need to "call" that function through the parentheses syntax: `function_name()`. Adding the parantheses to this function allows us to call it and use its output. Notice that the parantheses must be attached to the function name. That is, the syntax `function_name ()` will result in an error since there is a space between the function name and the parantheses. The values that go inside the parantheses are known as the *function parameters*, on which the function operates on.

For a more concrete example, take the `abs` function. This function takes in a single number as its argument and returns the absolute value of that number. Run the next two cells and see if you understand the output

```
In [14]: abs(5)

Out[14]: 5
```

```
In [15]: abs(-5)

Out[15]: 5
```

Functions can have zero or more parameters. The sky is truly the limit! The `print` statement we showed above is a also a function, in which the parameters can be anything from a string (text) to a number. Python is also very powerful in that it allows for nesting of functions. Run the cell below, which calls the `print` function on the output of the `abs` value function and prints it out.

```
In [18]: print("The absolute value of -5 is: ", abs(-5))

The absolute value of -5 is:  5
```

We can also implement our own functions and call them in the same manner that we do with built-in functions. Do not worry if you don't understand the fuction code, the overall structure will become more intuitive as we move through the class. For now, the main takeaway is that we can "call" built-in functions and user-made functions in the same way.

```
In [19]: def square(x):
    # Squares the value of x and outputs it
    return x * x

print("The square of 5 is: ", square(5))

The square of 5 is:  25
```

Question 2.3: Using the `square` function defined above, assign `three_squared` to the squared value of 3 in the cell below.

```
In [ ]: three_squared = ...
three_squared
```

3. Data

Now that we have gotten a bit familiarized with the Jupyter Notebook environment and seen some of the elementary concepts in Python, we can start delving a bit deeper into data science. As mentioned above, a lot of work in data science is achieved through the use of libraries. Here, we will give a small introduction to the `datascience` library and the table manipulations it allows us to perform. More than that, we will see how we can start using basic functions in the `datascience` package to derive insights about our data.

3.1 Background to Data & Data Importing

Our data comes from the National Crime Records Bureau of India which was set up in 1986 to "function as a repository of information on crime and criminals so as to assist the investigators in linking crime to the perpetrators." For the purpose of this analysis, we deal with two datasets: one pertaining to crime in cities in India and another pertaining to crime across Indian states/union territories.

Both of these datasets look at crimes in India between the years of 2017-2019 as well as at specific factors (population, total crime against women) that will be useful in trying to make inferences about gender-based violence in certain areas in India.

*Note: In the Indian numbering system, a **lakh** is equal to 100,000. Moreover, politically, a* union territory *is a small administrative unit like a state, but while states are self-governed, union territories are directly ruled over by the central or union government. Feel to read more here*

Here is our **state & union territory based dataset**. For now, don't worry too much about the specifics of the code. All you need to know is that we are loading in our datasets and assigning these to the names `states` and `cities`. There is no output because the cells contain only assignment statements, which don't produce an output.

```
In [3]: # When we reference this table later, notice that we can just call the variable name states!
states_url = "https://raw.githubusercontent.com/cxrlsc/data/master/global-fa21/State%20Data.csv"
states = Table.read_table(states_url)

# Again, we can reference this city table later using just the variable cities.
cities_url = "https://raw.githubusercontent.com/cxrlsc/data/master/global-fa21/Table%203B.1_3.csv"
cities = Table.read_table(cities_url)
```

3.2 Table Operations

For the rest of this section, we will look at some of the functions available to us from the `datascience` library. Throughout, we will be presenting the name of a table function and what operation that function executes. Then, we'll ask you to perform the operation. You can use this reference for more in depth explanations of each method if you want.

Note: The terms "method" and "function" are technical *not* the same thing. However, for the purposes of this course, we will use both terms to refer to the same thing.

`show`

The method `show` displays the first N rows in a dataset. You can specify what the value of N is by passing that argument onto the `show` method, as shown below. It is always good practice to display a few amount of rows to make sure your dataset was loaded correctly.

```
In [4]: states.show(3)
```

State/UT	2017 Crimes	2018 Crimes	2019 Crimes	Percentage State Share to Total (2019)	Mid-Year Projected Female Population (in lakhs) (2019)	Rate of Total Crime Against Women (2019)	2018-2019 GDP Per Capita ($)	2018-2019 Unemployment Rate (%)	2021 Literacy Rate (%)
Andhra Pradesh	17909	16438	17746	4.4	261.4	67.9	2480	7.3	67.02
Arunachal Pradesh	337	368	317	0.1	7.3	43.3	2253	11.1	65.38
Assam	23082	27687	30025	7.4	168.9	177.8	1365	10.7	72.19

... (33 rows omitted)

Question 3.1: Using the `show` method, display the **first five rows** of our Indian cities dataset.

Hint: Use the `cities` variable we defined two cells above.

```
In [ ]: ...
```

`num_columns` & `num_rows`

Before we introduce more table operations, it is important to differentiate between the terms *row* and *column*. A table is always composed of a certain amount of rows and columns. These values don't necessarily need to be the same.

Rows denote the horizontal components of a table, and usually represent the individual units in our data. On the other hand, columns denote the vertical components. Where rows represent the invidual units in our data, columns usually represent the features or characteristics of that unit. This will become more clear in the next few cells.

The table property `num_rows` outputs the total number of rows in a dataset. A property is different than a method/function in that it doesn't need to be called. Don't worry if these nuances are not fully clear. All you need to know is that a property does not require parantheses to produce an output.

```
In [9]: num_states_rows = states.num_rows
num_states_rows

Out[9]: 36
```

This shows us that we have a total of 36 rows in our states dataset. What this truly means is that each row represents a state, and thus the the total number of states in our dataset is 36. Let's verify this by displaying our dataset once again.

```
In [7]: states.show(1)
```

State/UT	2017 Crimes	2018 Crimes	2019 Crimes	Percentage State Share to Total (2019)	Mid-Year Projected Female Population (in lakhs) (2019)	Rate of Total Crime Against Women (2019)	2018-2019 GDP Per Capita ($)	2018-2019 Unemployment Rate (%)	2021 Literacy Rate (%)
Andhra Pradesh	17909	16438	17746	4.4	261.4	67.9	2480	7.3	67.02

... (35 rows omitted)

Now, let's see how many features we have for each state. We can do this by checking the number of rows, using the `num_columns` property.

```
In [8]: num_states_features = states.num_columns
num_states_features

Out[8]: 10
```

This tells us that our dataset contains 10 features for each of our 36 states.

Question 3.2: Using the `num_rows` property, find and assign `num_cities` to the **total number of cities** in our Indian cities dataset.

```
In [10]: num_cities = ...
num_cities

Out[10]: Ellipsis
```

Question 3.3: Using the `num_columns` property, find and assign `num_city_features` to the **total number of features for each city** in our Indian cities dataset.

```
In [ ]: num_city_features = ...
num_city_features
```

`labels` & `select`

Now that we have found what the number of rows and columns is for each of our datasets, it would be interesting to find what the name of those features were. We can do so with the table property `labels` as shown below:

```
In [19]: states.labels

Out[19]: ('State/UT',
 '2017 Crimes',
 '2018 Crimes',
 '2019 Crimes',
 'Percentage State Share to Total (2019)',
 'Mid-Year Projected Female Population (in lakhs) (2019)',
 'Rate of Total Crime Against Women (2019)',
 '2018-2019 GDP Per Capita ($)',
 '2018-2019 Unemployment Rate (%)',
 '2021 Literacy Rate (%)')
```

Here, we can our `states` dataset, we have information about the number of crimes against women in each state during 2017, 2018, and 2019, alongside other information. Now, say we were interest **only** on these features. Using the `select` table method, we can specify which columns (features) are the only ones we'd like to see. Notice how we wrap the column names with quotation marks `".."` and separate the name of each column with a comma. Notice how capitalization is also the same in the output of `states.labels` above and the name of that column in the `select` method. These are all small, but crucial components that need to be addressed early on or else the code will error.

```
In [22]: states_crime_2017_to_2019 = states.select("2017 Crimes", "2018 Crimes", "2019 Crimes")
states_crime_2017_to_2019.show(3)
```

2017 Crimes	2018 Crimes	2019 Crimes
17909	16438	17746
337	368	317
23082	27687	30025

... (33 rows omitted)

Now we have a table that contains only the columns we were interested on. However, notice how we do not have the state name data anymore. No worries, the `select` method can select as many columns as you need.

Question 3.4: Using the `select` method, find and assign `states_crimes_2017_to_2019` to a table containing the state name, and that state's respective number of crimes against women during 2017, 2018, 2019.

Hint: What column name would you add in the `select` method example shown above?

Note: Make sure that your column names have the correct capitalization. Adding an extra space when there isn't one may also result in errors.

```
In [ ]: states_crime_2017_to_2019 = states.select("...", "...", "...", "...")
states_crime_2017_to_2019.show(3)
```

`sort`

Now, suppose we were interested in the states with the highest rates of crime against women. We can do this with the `sort` method! We can sort by whatever column we'd like. One important measure when we have data that deals with time is to observe any changes in our data throughout time. We could do this with either our `cities` or `states` dataset using the columns that contain information on crimes against women in 2017, 2018, and 2019. First, let's see which states had the highest amount of crimes against women in 2017.

```
In [26]: states.sort("2017 Crimes").show(5)
```

State/UT	2017 Crimes	2018 Crimes	2019 Crimes	Percentage State Share to Total (2019)	Mid-Year Projected Female Population (in lakhs) (2019)	Rate of Total Crime Against Women (2019)	2018-2019 GDP Per Capita ($)	2018-2019 Unemployment Rate (%)	2021 Literacy Rate (%)
Lakshadweep (UT)	6	11	38	0	0.3	115.2	931	28.6	91.85
D&N Haveli (UT)	20	38	49	0	2.3	21.6	635	1.8	76.24
Daman & Diu (UT)	26	16	33	0	1.3	25.2	988	0	87.1
Nagaland	79	75	43	0	10.4	4.1	1918	21.1	79.55
A & N Islands (UT)	132	147	135	0	1.9	72.2	3321	12	86.63

... (31 rows omitted)

This doesn't look correct, however. This is because the `sort` method sorts values from smallest to greatest by default. No worries, we just have to add an extra argument to switch this:

```
In [27]: states.sort("2017 Crimes", descending=True).show(5)
```

State/UT	2017 Crimes	2018 Crimes	2019 Crimes	Percentage State Share to Total (2019)	Mid-Year Projected Female Population (in lakhs) (2019)	Rate of Total Crime Against Women (2019)	2018-2019 GDP Per Capita ($)	2018-2019 Unemployment Rate (%)	2021 Literacy Rate (%)
Uttar Pradesh	56011	59445	59853	14.7	1081.4	55.4	1043	10.6	67.68
Maharashtra	31979	35497	37144	9.2	588.5	63.1	3193	6.4	82.34
West Bengal	30992	30394	30394	7.5	474.9	64	1634	4.9	76.26
Madhya Pradesh	29788	28942	27560	6.8	399.6	69	1461	7.4	69.32
Rajasthan	25993	27866	41550	10.2	376.4	110.4	1820	9.5	66.11

... (31 rows omitted)

Question 3.5: Using the `sort` method, generate a table sorted by the number of crimes against women in 2018 from greatest to smallest.

```
In [29]: states.sort("2019 Crimes", descending=True).show(5)
```

State/UT	2017 Crimes	2018 Crimes	2019 Crimes	Percentage State Share to Total (2019)	Mid-Year Projected Female Population (in lakhs) (2019)	Rate of Total Crime Against Women (2019)	2018-2019 GDP Per Capita ($)	2018-2019 Unemployment Rate (%)	2021 Literacy Rate (%)
Uttar Pradesh	56011	59445	59853	14.7	1081.4	55.4	1043	10.6	67.68
Rajasthan	25993	27866	41550	10.2	376.4	110.4	1820	9.5	66.11
Maharashtra	31979	35497	37144	9.2	588.5	63.1	3193	6.4	82.34
West Bengal	30992	30394	30394	7.5	474.9	64	1634	4.9	76.26
Assam	23082	27687	30025	7.4	168.9	177.8	1365	10.7	72.19

... (31 rows omitted)

Question 3.6: Compare and contrast between the five states with the highest amount of crimes against women in 2017 and 2019. Is it the same set of states for each? If not, what states are present in one and missing in the other? Write your answer below.

Replace this text with your answer

Conclusion

Congratulations! You've reached the end of the assignment. Although we only touched upon some basics of data science, next week we will expand on these ideas and build up from there. Next week we will look at more complicated table operations that will allow us to develop more nuanced insights about our data.

Submitting Your Work

```
In [ ]: # This may take a few seconds
from IPython.display import display, HTML
!pip install -U notebook-as-pdf -q
!jupyter-nbconvert --to PDFviaHTML notebook1.ipynb
display(HTML("Save this notebook, then click <a href='notebook1.pdf' download>here</a> to open the pd
f."))
```

4. Data Science Resources at UC Berkeley

If any assistance is required for this notebook, you can open up a thread on Piazza or email any of the staff. We also have our peer advisors available to help! Their drop-in hours are here. You can also email them to book an appointment if there are any time conflicts.

If you are interested in data science as a whole, a great course to start with is Data 8, which is designed for students with no programming or stats experience. For the full list of courses and degree programs, click here.

Feedback:

Please let us know of your thoughts on this notebook! Fill out the following survey here.