Instructor Notes

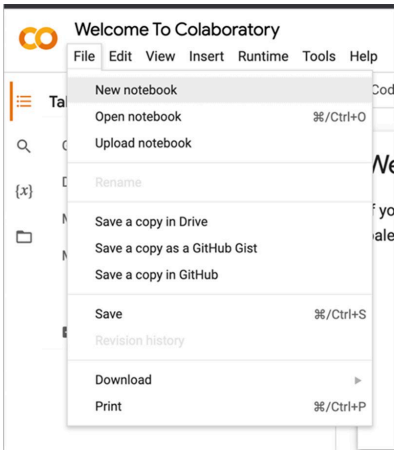
Hello everyone! My name is Tina and I am a Fondren Fellow working on a project to enhance introductory Python workshops with effective teaching strategies under the guidance of Dr. Catherine Barber. I have been working for the past few months on making this workshop as accessible as possible to all skill levels, so let's get into it! I will try my best to wrap up this workshop within 90 minutes, but there is a chance it might go over time. I understand that you all might have other commitments, so feel free to leave when you need to.

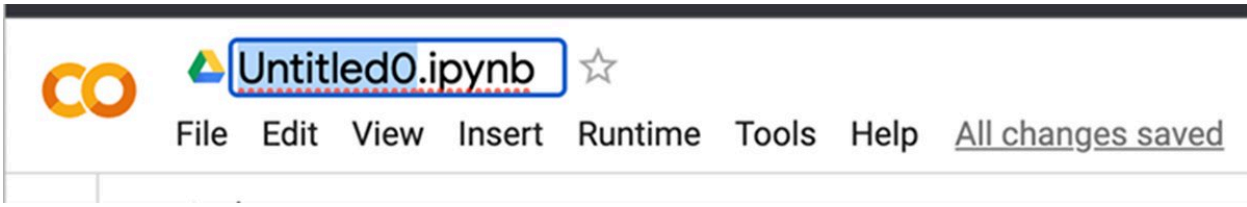
Why Python?

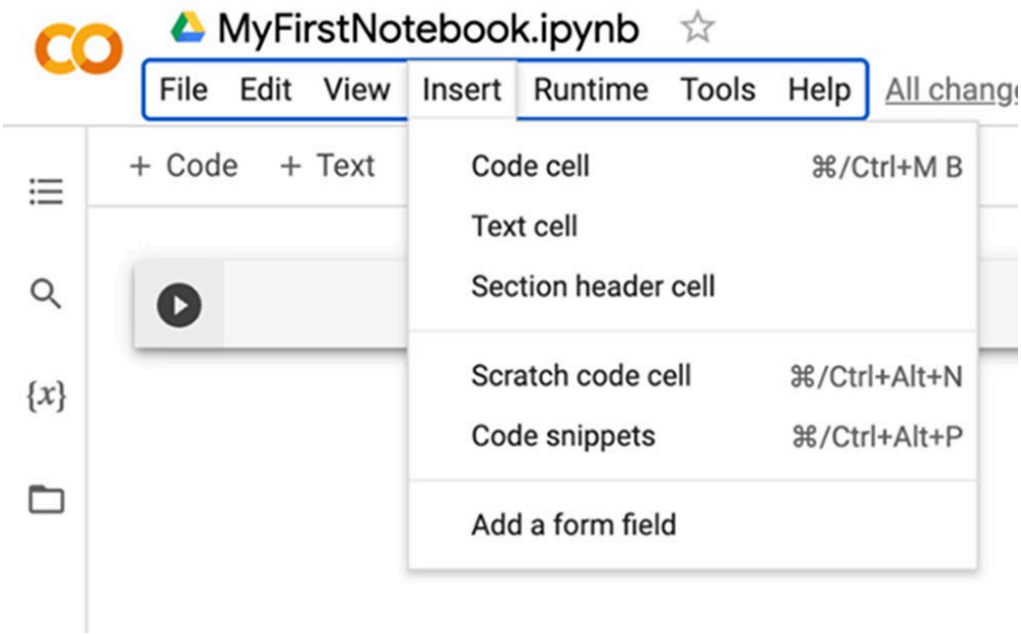
- Python is a major language in some of the most in-demand disciplines, such as machine learning, artificial intelligence, Big Data, and robotics, and cyber security.
- Python is free, open source, and cross-platform (no need to pay/have a license, you can freely use and distribute Python code, even for commercial purposes)
- Python is a high-level language
    - the syntax is composed of mostly normal words you use day to day (such as "print" when you want to print something)
- Large standard library
    - use packages to help our code with specific tasks, it reduces the work for the programmer because packages you need to use are already available

About Google Colab

- Google Colaboratory, or Colab, is a hosted Jupyter Notebook (a free software for interactive computing across all programming languages) service that enables you to write and execute Python code through your browser without any local software installation
- How to use:
    - To use Colaboratory, you must have a Google account.
    - On your first visit, you will see a "Welcome To Colaboratory" notebook with links to video introductions and basic information on how to use Colab.
    - From the File menu, click New Notebook to create a workbook.

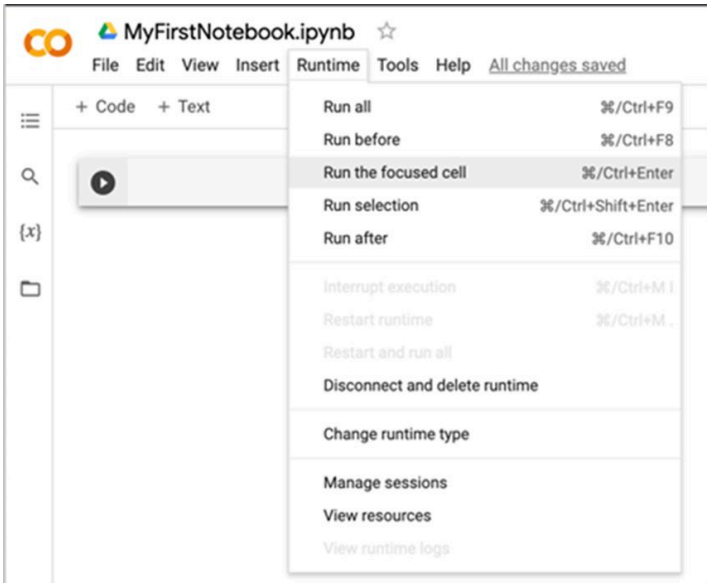
- 

- The notebook will by default have a generic name; click on the filename field to rename it. The file type, IPYNB, is short for "IPython notebook" because IPython was the forerunner of Jupyter Notebook.
- 

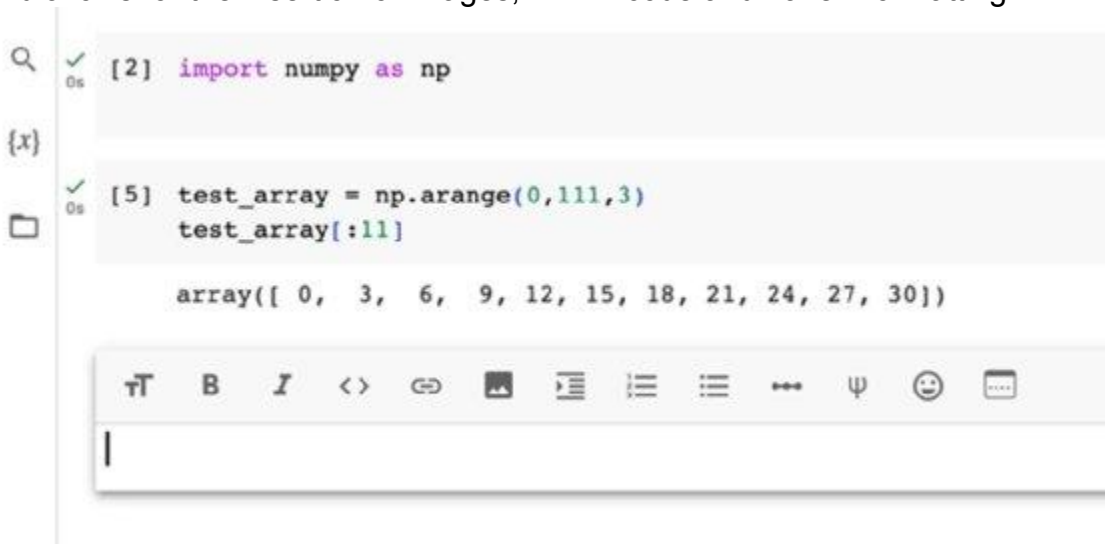
- The interface allows you to insert various kinds of cells, mainly text and code, which have their own shortcut buttons under the menu bar via the Insert menu.
- 

- You can insert Python code to execute in a code cell. The code can be entirely standalone or imported from various Python libraries.

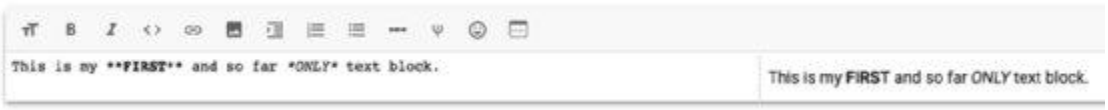
- A notebook can be treated as a rolling log of work, with earlier code snippets being no longer executed in favor of later ones, or treated as an evolving set of code blocks intended for ongoing execution. The Runtime menu offers execution options, such as Run all, Run before or Run the focused cell, to match either approach.

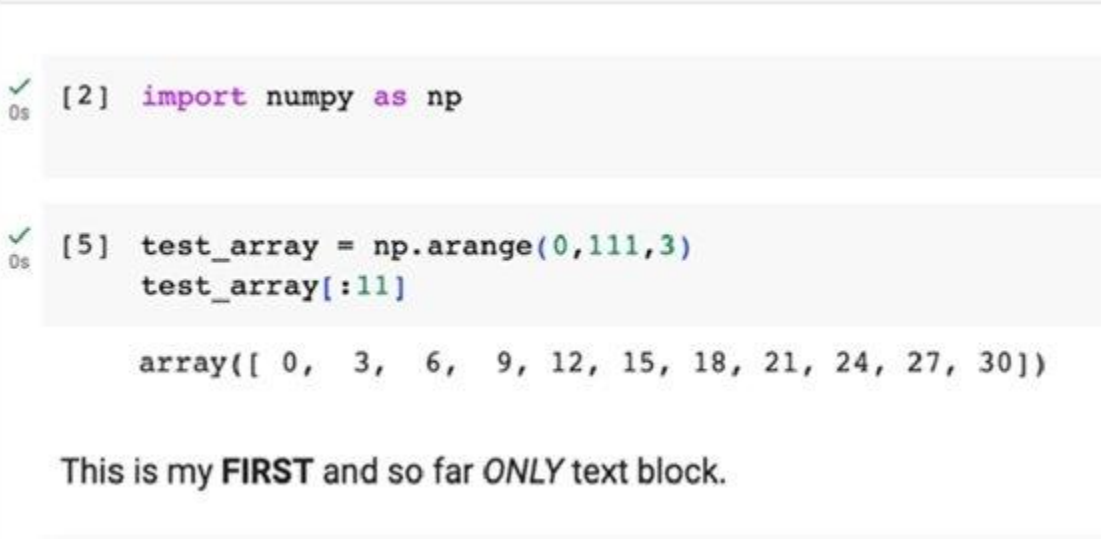


- Each code cell has a “play” icon on the left edge, as shown above. You can type code into a cell and hit that icon to execute it immediately.
- If the code generates an error, the error output will appear beneath the cell. Correcting the problem and hitting run again replaces the error info with program output.
- A text cell provides basic rich text using Markdown formatting by default and allows for the insertion of images, HTML code and LaTeX formatting.



- As you add text on the left side of the text cell, the formatted output appears on the right.

- 

- Once you stop editing a block, only the final formatted version shows.
- 

Pre-Test

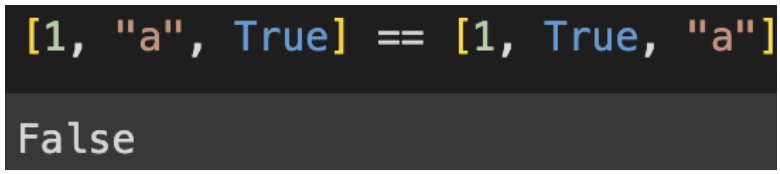
- https://forms.gle/3yjHY6NFHD3uNzZP9

The print() Function

- The print function takes in an input and outputs it in your terminal
- Let's print out the popular computer science string "Hello World"
    - "Fondren is awesome!"
    - You can also print out numbers
    - Or a combination of words and numbers print("Temperature Outside:", 55)
    - Why do letters have quotation marks and numbers don't? This has to do with the fact that they are different object types … (transition to next section)
    - To run only one line: Cmd + Shift + Enter

Fundamental Object Types

- You may see the term "object" used frequently in your programming journey. In Python, the term "object" is quite the catch-all; including numbers, strings of characters, lists, and functions - a Python object is essentially anything that you can assign to a variable. That being said, there are different types of objects: Python treats integers as a different type of object than a string, for instance.
- numbers (integers, floating-point numbers, and complex numbers)
    - integers are any whole number

    - floating-point numbers are numbers with a decimal
- booleans
    - There are two boolean-type objects: True and False; they belong to the built-in type bool
- the “null” type
    - NoneType that has exactly one object: None. None is used to represent “null”… nothing
- strings
    - The string type is used to store written characters. A string can be formed using single quotes, double quotes, or triple quotes
    - For R users: Python does not have a character data type, a single character is simply a string with a length of 1
- lists
    - A list is a type of Python object that allows us to store a sequence of other objects. One of its major utilities is that it provides us with a means for updating the contents of a list later on.
    - A list object is created using square brackets, and its contents are separated by commas: [item1, item2, ..., itemN]. Its contents need not be of the same type of object.
    - Like a string, the ordering of a list’s contents matters, meaning that a list is sequential in nature.
    - 

- There are times when you need to change from one object type to another
    - int()
    - float()
    - str()
    - For example, if you are trying to run a linear regression model to predict a certain variable y, but the y variable is a string, you would need to convert it to a numerical object before running your model
    - You can check the type of an object by using isinstance() or type()

- Practice 1 for using print() and different object types

Variables

- Variables allow you to give names to values in your programs and have several benefits:
    - They can add significant clarity to your program. It's going to mean a lot more when you see a word naming a value rather than say just a number
    - You can reuse values in your programs easily you can just use a variable name over and over again rather than having to recompute the value every time that you need it
- To assign a value to a variable, we use the = operator, also known as the assignment operator. This is different from the == operator, which is used to check equivalence
  (ex: x = 31)
    - The = may seem a little deceptive and imply that x is a location that you are putting the number 31 into. This is not correct and is not what is happening behind the scenes. All I am doing is giving a name, x, to the number 31
- Valid variable names need to start with an underscore or with a letter. I wouldn't recommend using underscores at this stage and would instead recommend starting your names with letters followed by letters, numbers, or underscores. Just because a variable name is valid doesn't make it good, here are some common Python conventions:
    - Variable names should be meaningful and descriptive
        - having the names ABC or A9_ don't really mean anything and don't add information to your program
    - Variable names should start with a lowercase letter and any space in the name should be replaced with an underscore

- example:

```
#Variables NAME values
age = 31
print(age)
bobs_age = 54
print(bobs_age)

#You can assign any expression to a variable
bakers_dozen = 12 + 1
junk = (32 - 62) / (3.2 ** 4)
print(junk)

#Variables allow you to reuse values
fahrenheit = 23
celsius = (5.0 / 9.0) * (fahrenheit - 32)
print(fahrenheit, celsius)
print("Celsius:", celsius)
```

```
31
54
-0.28610229492187494
23 -5.0
Celsius: -5.0
```

- **Practice 2 for Variables**

Arithmetic and Comparison Operators

- Arithmetic: + is addition, - is subtraction, * is multiplication, / is division, ** is exponentiation, // is integer division
- Comparison: == is equivalence (careful not to confuse = with ==), != is not equal, >, >=, <, <=

```
print(3 + 2)
print(17.6 - 143.7)
print(3 * 4)
print(9 / 2)

#The // is integer division, so instead of returning 4.5, it will truncate the
#decimal and just give the "integer" part of the result
print(9 // 2)

#If you put in a float, it will still return the "integer" part, but instead
#return a float
print(9.0 // 2)
print(3.0 * 4)
print(3.2 * 4.3)
```

```
5
-126.1
12
4.5
4
4.0
12.0
13.76
```

-

    - // is also known as floor division because the floor() function from the math module in Python returns the largest integer not greater than its input
    - Note: Diving two integers in Python 3 with / returns a float
- **Practice 3 for Variables and Arithmetic and Comparison Operators**

Functions

- Functions are a powerful programming construct. They allow you to create code that takes inputs, performs a computation, and produces an output. You can then use these functions over and over to compute results based upon different inputs.
- Let's consider an example of a function that returns the sum of squares of two inputs and breakdown the the elements of a function:

```
def sum_of_squares(num1, num2):
    """
    Compute the sum of the squares of the two inputs.

    Inputs:
       num1 - floating point number
       num2 - floating point number

    Returns the sum of squares of num1 and num2
    """
    sq1 = num1 * num1
    sq2 = num2 * num2
    sumsq = sq1 + sq2
    return sumsq
```

1. **def**: All function definitions must begin with the Python keyword def (short for "define"). Note that def is not indented at all. In general, this will be the case. As you learn more Python, you will find that there are exceptions to this, but at this point, your functions will always start at the beginning of a line.
2. **name**: After the keyword def, your function must have a name. In this case, it is sum_of_squares. Function names must begin with a letter. Function names can include letters, numbers, and underscore (_) characters. By convention in Python function names should not use uppercase letters and words should be separated by underscores. You should choose clear, understandable function names that indicate what the function does.
3. **parameters**: The parameters, or inputs, of the function appear directly after the name. They are surrounded by parentheses. A function in Python can have zero or more parameters. The parameters should be separated by commas. In the above example, there are two parameters, num1 and num2.
4. **colon**: After the parameters, there must be a colon. Python requires this colon to indicate that what follows will be the code for the function.

5. **indentation**: Note that the remainder of the function must be indented. In some languages, indentation is optional and is used simply to help the person reading the code to understand what is and what is not a part of the function. In Python, this indentation is required. By convention, the code of the function should all be indented exactly 4 spaces.
6. **docstring**: The line immediately following the colon should be a docstring (short for "documentation string"). The docstring for this function is on lines 2 through 10. This is a multiline string that describes, in English, what the function does. You should not be describing how the function performs the computation. Rather, this is a description of how to use the function. The docstring can be as long as necessary to explain the behavior of the function. However, it should be clear and concise so as not to confuse the reader.
7. **body**: After the docstring, is the body of the function. This is the actual code that will be executed when the function is called. The code for this function is on lines 11 through 14. A function can contain any valid Python code. This function includes arithmetic expressions and variable assignments. Note that you can even call other functions within a function.
8. **return**: Functions often return values to the caller. The return keyword is used to indicate what should be returned from the function. Whatever appears after the return keyword is what will be returned. This can be any expression, which will first be evaluated and then the result will be returned. In this case, the value of the variable sumsq will be returned. This is not strictly required however. The function may be performing actions (such as printing messages) that do not require it to return anything to the caller. If you omit the return statement from your function, Python will automatically return the special value None from the function.
9. **local variables**: There are two types of local variables that exist in this function: the parameters and variables that are defined within the function body. These local variables are newly created each time the function is called and then they only exist during the execution of the function. This is important to understand. You cannot, for instance, refer to the variable sq1 outside of this function. It does not exist! It is created during the execution of the function and is destroyed as soon as the function returns. Similarly, you cannot refer to the parameter num1 outside of the function. When the function is called num1 is assigned the value of the first argument that was passed to the function. You can then use this variable within the body of the function. But, again, it is destroyed as soon as the function is returned. It is a common mistake to try to refer to parameters and other local variables of a function outside of the body of the function. This will not work. Furthermore, if you create variables with the same names as those within the function, they are different variables that have nothing to do with the variables

you have created inside the function. When you call functions, you should only try to interact with functions by passing them arguments and using their return values. The caller of a function should not try to use any of the local variables of a function outside of that function.

- In order to call a function, you must use the function's name, pass it any arguments it needs, and then optionally do something with the return value. For example, you could call the sum_of_squares function as follows:

```
print(sum_of_squares(3, 5))

#OR

number = 3
value = sum_of_squares(number, 5)
print(value)
```

- More on the default return value of functions:
    - All functions in Python must return some value. However, it is not an error if you do not have a return statement in your function. Instead, Python effectively adds one for you that returns the special value None. In Python, None is used to mean the absence of a value. It can be used like other values: you can print it, you can compare to it, etc. The following two functions return the same thing:

```
#Default Return Value
def say_hello():
    """
    Prints hello.

    Inputs:
      none

    returns None
    """
    print("Hello!")

result = say_hello()
# The following line will print None
print(result)

def say_hello2():
    """
    Prints hello.

    Inputs:
      none

    returns None
    """
    print("Hello!")
    return None

result2 = say_hello2()
# The following line will print None
print(result2)
```

- **Practice 4 for Functions**

Conditionals - STOP HERE

The if Statement

- Conditionals are an important part of programming, as they allow the program to perform different actions depending on the situation that exists when the program is run. The actions of the program are dependent on some logical expression, that likely includes arithmetic comparisons and/or boolean logic.

```
if hungry:
    print("Have a snack!")
```

- The conditional statement starts with the keyword **if**. Then you have a condition that evaluates to either **True** or **False**. The condition can be any Python expression that will evaluate to **True** or **False**. Finally, the line ends with a colon. This colon is required and it indicates to Python that what follows is the body of the if statement. The body of the if statement can contain any Python code and can be as long as necessary. But, all lines in the body of the if statement must be indented. As with functions, the

convention is that the body should be indented by 4 spaces. Once the indentation stops, the body of the if statement is over and the subsequent code will execute regardless of the value of the condition. So, in this code, if **hungry** is a boolean, then this program will print a message if **hungry** is **True** and will do nothing if it is **False**. This allows you to write programs which will only print this message if the "user" is actually hungry.

The else Clause

- Sometimes, you would like to do one thing if the condition is true and something else if it is false. Consider the following program:

```
if hungry:
    print("Have a snack!")
else:
    print("Save your snack for later!")
```

- An **if** statement can also include optional clauses following the body of the if statement. You can optionally include an **else** clause which will execute if the condition of the if statement evaluates to **False**. The else clause starts with the keyword **else** followed by a colon. After that, the body of the else clause must be indented, again with 4 spaces by convention. As with the if statement body, the body of the else clause can contain any Python code and ends when the indentation stops.
- Note that *either* the body of the if or the body of the else will execute, but not both. If **hungry** is **True**, then the code on line 2 will execute. If it is **False**, then the code on line 4 will execute.

The elif Clause

- Sometimes you want to build up more complicated conditional expressions. Consider the following code:

```
if hungry and thirsty:
    print("Go have lunch!")
elif hungry:
    print("Have a snack!")
elif thirsty:
    print("Have a drink!")
else:
    print("Save your food and drinks for later!")
```

- You can also add **elif** clauses (short for "else if") to an if statement. Each elif clause includes a condition, exactly like the initial if statement. An elif

clause must directly follow either the initial if statement or another elif clause and has the following structure: the **elif** keyword, an expression that evaluates to **True** or **False**, and a colon. The body of the elif clause is then indented by 4 spaces and ends when the indentation ends.

- The way to understand the elif clause is that if none of the conditions above it have evaluated to **True**, then the clause will execute. If the condition of the elif clause is **True**, then the associated body will execute, and no other bodies in the
- **if/elif/else** construct will execute. If the condition of the elif clause is **False**, then the associated body will not execute and any following **if/elif/else** clauses will be checked.
- There can be as many elif clauses as you would like, but they must follow a single if statement. They can optionally be followed by a single else clause at the end. The body of the final else clause, if it exists, executes only if none of the previous conditions were **True**.

- **Practice 5 for Conditionals**

Arithmetic Comparisons in Conditionals

- Arithmetic comparisons are often used within the conditions of if statements (and associated elif clauses). Consider the following code:

```
if (value > -10) and (value < 10):
    # do something
elif (value <= -10):
    # do something different
elif (value >= 10):
    # do another different thing
```

- This program has three possible paths, depending on the value of the variable **value**. If it is between -10 and 10 (but not exactly -10 or 10), then the body on line 2 will be executed. If, instead, it is less than or equal to -10, the body on line 4 will be executed. Finally, if it is greater than or equal to 10, the body on line 6 will be executed. No else clause is necessary here, since all possibilities have been covered. Note, however, that the final elif clause could be replaced with an else clause, since if the code gets to that point, **value** must be greater than or equal to 10.

- **Practice 6 for Functions and Conditionals**

Post-Test

- Hand out post-test
- Hand out answer key with solutions and explanations