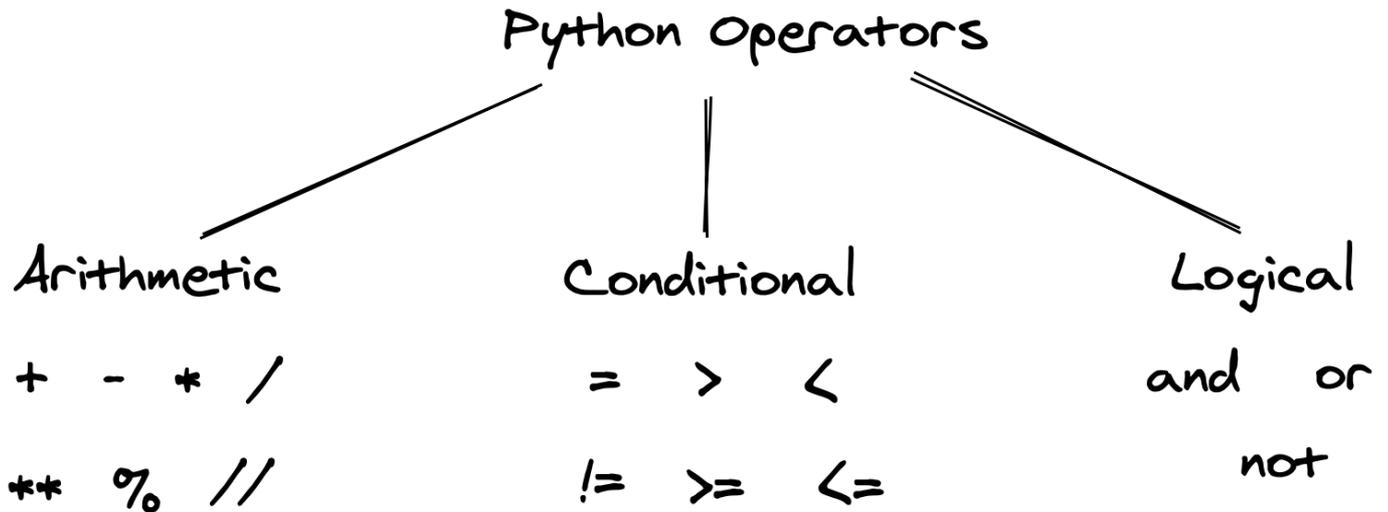


First Steps with Python and Jupyter



This tutorial covers the following topics:

- Performing arithmetic operations using Python
- Solving multi-step problems using variables
- Evaluating conditions using Python
- Combining conditions with logical operators
- Adding text styles using Markdown

How to run the code

This tutorial is an executable [Jupyter notebook](#) hosted on [Jovian](#) (don't worry if these terms seem unfamiliar; we'll learn more about them soon). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Jupyter Notebooks: This tutorial is a [Jupyter notebook](#) - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc. instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things -

you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Performing Arithmetic Operations using Python

Let's begin by using Python as a calculator. You can write and execute Python using a code cell within Jupyter.

Working with cells: To create a new cell within Jupyter, you can select "Insert > Insert Cell Below" from the menu bar or just press the "+" button on the toolbar. You can also use the keyboard shortcut `Esc+B` to create a new cell. Once a cell is created, click on it to select it. You can then change the cell type to code or markdown (text) using the "Cell > Cell Type" menu option. You can also use the keyboard shortcuts `Esc+Y` and `Esc+M`. Double-click a cell to edit the content within the cell. To apply your changes and run a cell, use the "Cell > Run Cells" menu option or click the "Run" button on the toolbar or just use the keyboard shortcut `Shift+Enter`. You can see a full list of keyboard shortcuts using the "Help > Keyboard Shortcuts" menu option.

Run the code cells below to perform calculations and view their result. Try changing the numbers and run the modified cells again to see updated results. Can you guess what the `//`, `%`, and `**` operators are used for?

```
2 + 3 + 9
```

14

```
99 - 73
```

26

```
23.54 * -1432
```

-33709.28

```
100 / 7
```

14.285714285714286

```
100 // 7
```

14

```
100 % 7
```

2

```
5 ** 3
```

125

As you might expect, operators like `/` and `*` take precedence over other operators like `+` and `-` as per mathematical conventions. You can use parentheses, i.e. `()`, to specify the order in which operations are performed.

```
((2 + 5) * (17 - 3)) / (4 ** 3)
```

1.53125

Python supports the following arithmetic operators:

Operator	Purpose	Example	Result
<code>+</code>	Addition	<code>2 + 3</code>	5
<code>-</code>	Subtraction	<code>3 - 2</code>	1
<code>*</code>	Multiplication	<code>8 * 12</code>	96
<code>/</code>	Division	<code>100 / 7</code>	14.28..
<code>//</code>	Floor Division	<code>100 // 7</code>	14
<code>%</code>	Modulus/Remainder	<code>100 % 7</code>	2
<code>**</code>	Exponent	<code>5 ** 3</code>	125

Try solving some simple problems from this page: <https://www.math-only-math.com/worksheet-on-word-problems-on-four-operations.html>.

You can use the empty cells below and add more cells if required.

Solving multi-step problems using variables

Let's try solving the following word problem using Python:

A grocery store sells a bag of ice for \$1.25 and makes a 20% profit. If it sells 500 bags of ice, how much total profit does it make?

We can list out the information provided and gradually convert the word problem into a mathematical expression that can be evaluated using Python.

Cost of ice bag (\$) = 1.25

Profit margin = 20% = .2

Profit per bag (\$) = profit margin * cost of ice bag = .2 * 1.25

No. of bags = 500

Total profit = no. of bags * profit per bag = 500 * (.2 * 1.25)

```
500 * (.2 * 1.25)
```

125.0

Thus, the grocery store makes a total profit of \$125. While this is a reasonable way to solve a problem, it's not entirely clear by looking at the code cell what the numbers represent. We can give names to each of the numbers by creating Python *variables*.

Variables: While working with a programming language such as Python, information is stored in *variables*. You can think of variables as containers for storing data. The data stored within a variable is called its *value*.

```
cost_of_ice_bag = 1.25
```

```
profit_margin = .2
```

```
number_of_bags = 500
```

The variables `cost_of_ice_bag`, `profit_margin`, and `number_of_bags` now contain the information provided in the word problem. We can check the value of a variable by typing its name into a cell. We can combine variables using arithmetic operations to create other variables.

Code completion: While typing the name of an existing variable in a code cell within Jupyter, just type the first few characters and press the `Tab` key to autocomplete the variable's name. Try typing `pro` in a code cell below and press `Tab` to autocomplete to `profit_margin`.

```
profit_margin
```

0.2

```
profit_per_bag = cost_of_ice_bag * profit_margin
```

```
profit_per_bag
```

0.25

```
total_profit = number_of_bags * profit_per_bag
```

```
total_profit
```

125.0

If you try to view the value of a variable that has not been *defined*, i.e., given a value using the assignment statement `variable_name = value`, Python shows an error.

```
net_profit = 100
```

Storing and manipulating data using appropriately named variables is a great way to explain what your code does.

Let's display the result of the word problem using a friendly message. We can do this using the `print` function.

Functions: A function is a reusable set of instructions. It takes one or more inputs, performs certain operations, and often returns an output. Python provides many in-built functions like `print` and also allows us to define our own functions.

```
print("The grocery store makes a total profit of $", total_profit)
```

The grocery store makes a total profit of \$ 125.0

print : The `print` function is used to display information. It takes one or more inputs, which can be text (within quotes, e.g., "this is some text"), numbers, variables, mathematical expressions, etc. We'll learn more about variables & functions in the next tutorial.

Creating a code cell for each variable or mathematical operation can get tedious. Fortunately, Jupyter allows you to write multiple lines of code within a single code cell. The result of the last line of code within the cell is displayed as the output.

Let's rewrite the solution to our word problem within a single cell.

```
# Store input data in variables
cost_of_ice_bag = 1.25
profit_margin = .2
number_of_bags = 500

# Perform the required calculations
profit_per_bag = cost_of_ice_bag * profit_margin
total_profit = number_of_bags * profit_per_bag

# Display the result
print("The grocery store makes a total profit of $", total_profit)
```

The grocery store makes a total profit of \$ 125.0

Note that we're using the `#` character to add *comments* within our code.

Comments: Comments and blank lines are ignored during execution, but they are useful for providing information to humans (including yourself) about what the code does. Comments can be inline (at the end of some code), on a separate line, or even span multiple lines.

Inline and single-line comments start with `#`, whereas multi-line comments begin and end with three quotes, i.e. `"""`. Here are some examples of code comments:

```
my_favorite_number = 1 # an inline comment
```

```
# This comment gets its own line
my_least_favorite_number = 3
```

```
"""This is a multi-line comment.
Write as little or as much as you'd like.

Comments are really helpful for people reading
your code, but try to keep them short & to-the-point.

Also, if you use good variable names, then your code is
often self explanatory, and you may not even need comments!
"""
a_neutral_number = 5
```

EXERCISE: A travel company wants to fly a plane to the Bahamas. Flying the plane costs 5000 dollars. So far, 29 people have signed up for the trip. If the company charges 200 dollars per ticket, what is the profit made by the company? Create variables for each numeric quantity and use appropriate arithmetic operations.

```
flight_cost = 5000
total_passengers = 29
ticket_price = 200
profit = (total_passengers * ticket_price) - flight_cost
print("Profit = ", profit)
```

Profit = 800

My goofings around:

```
bears = 50
chickens = 300
pigs = 45
turkeys = 33

total_animals = bears + chickens + pigs + turkeys
print("Total animals = ", total_animals)
```

Total animals = 428

Evaluating conditions using Python

Apart from arithmetic operations, Python also provides several operations for comparing numbers & variables.

Operator	Description
<code>==</code>	Check if operands are equal

Operator	Description
<code>!=</code>	Check if operands are not equal
<code>></code>	Check if left operand is greater than right operand
<code><</code>	Check if left operand is less than right operand
<code>>=</code>	Check if left operand is greater than or equal to right operand
<code><=</code>	Check if left operand is less than or equal to right operand

The result of a comparison operation is either `True` or `False` (note the uppercase `T` and `F`). These are special keywords in Python. Let's try out some experiments with comparison operators.

```
my_favorite_number = 1
my_least_favorite_number = 5
a_neutral_number = 3
```

```
# Equality check - True
my_favorite_number == 1
```

`True`

```
# Equality check - False
my_favorite_number == my_least_favorite_number
```

`False`

```
# Not equal check - True
my_favorite_number != a_neutral_number
```

`True`

```
# Not equal check - False
a_neutral_number != 3
```

`False`

```
# Greater than check - True
my_least_favorite_number > a_neutral_number
```

`True`

```
# Greater than check - False
my_favorite_number > my_least_favorite_number
```

`False`

```
# Less than check - True
my_favorite_number < 10
```

`True`

```
# Less than check - False
my_least_favorite_number < my_favorite_number
```

False

```
# Greater than or equal check - True
my_favorite_number >= 1
```

True

```
# Greater than or equal check - False
my_favorite_number >= 3
```

False

```
# Less than or equal check - True
3 + 6 <= 9
```

True

```
# Less than or equal check - False
my_favorite_number + a_neutral_number <= 3
```

False

Just like arithmetic operations, the result of a comparison operation can also be stored in a variable.

```
cost_of_ice_bag = 1.25
is_ice_bag_expensive = cost_of_ice_bag >= 10
print("Is the ice bag expensive?", is_ice_bag_expensive)
```

Is the ice bag expensive? False

Combining conditions with logical operators

The logical operators `and`, `or` and `not` operate upon conditions and `True` & `False` values (also known as *booleans*). `and` and `or` operate on two conditions, whereas `not` operates on a single condition.

The `and` operator returns `True` when both the conditions evaluate to `True`. Otherwise, it returns `False`.

a	b	a and b
True	True	True
True	False	False
False	True	False
False	False	False

```
my_favorite_number
```

1

```
my_favorite_number > 0 and my_favorite_number <= 3
```

True

```
my_favorite_number < 0 and my_favorite_number <= 3
```

False

```
my_favorite_number > 0 and my_favorite_number >= 3
```

False

```
True and False
```

False

```
True and True
```

True

The `or` operator returns `True` if at least one of the conditions evaluates to `True`. It returns `False` only if both conditions are `False`.

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	False

```
a_neutral_number = 3
```

```
a_neutral_number == 3 or my_favorite_number < 0
```

True

```
a_neutral_number != 3 or my_favorite_number < 0
```

False

```
my_favorite_number < 0 or True
```

True

```
False or False
```

False

The `not` operator returns `False` if a condition is `True` and `True` if the condition is `False` .

```
not a_neutral_number == 3
```

False

```
not my_favorite_number < 0
```

True

```
not False
```

True

```
not True
```

False

Logical operators can be combined to form complex conditions. Use round brackets or parentheses (`and`) to indicate the order in which logical operators should be applied.

```
(2 > 3 and 4 <= 5) or not (my_favorite_number < 0 and True)
```

True

```
not (True and 0 < 1) or (False and True)
```

False

If parentheses are not used, logical operators are applied from left to right.

```
not True and 0 < 1 or False and True
```

False

Experiment with arithmetic, conditional and logical operators in Python using the interactive nature of Jupyter notebooks. We will learn more about variables and functions in future tutorials.

Adding text styles using Markdown

Adding explanations using text cells (like this one) is a great way to make your notebook informative for other readers. It is also useful if you need to refer back to it in the future. You can double click on a text cell within Jupyter to edit it. In the edit mode, you'll notice that the text looks slightly different (for instance, the heading has a `##` prefix. This text is written using Markdown, a simple way to add styles to your text. Execute this cell to see the output without the special characters. You can switch back and forth between the source and the output to apply a specific style.

For instance, you can use one or more `#` characters at the start of a line to create headers of different sizes:

Header 1

Header 2

Header 3

Header 4

To create a bulleted or numbered list, simply start a line with `*` or `1 . .`

A bulleted list:

- Item 1
- Item 2
- Item 3

A numbered list:

1. Apple
2. Banana
3. Pineapple

You can make some text bold using `**`, e.g., **some bold text**, or make it italic using `*`, e.g., *some italic text*. You can also create links, e.g., [a link](#). Images are easily embedded too:



Another really nice feature of Markdown is the ability to include blocks of code. Note that code blocks inside Markdown cells cannot be executed.

```
# Perform the required calculations
profit_per_bag = cost_of_ice_bag * profit_margin
total_profit = number_of_bags * profit_per_bag

# Display the result
print("The grocery store makes a total profit of $", total_profit)
```

You can learn the full syntax of Markdown here: <https://learnxinyminutes.com/docs/markdown/>

Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let

them execute your code. [Jovian](#) offers an easy way of saving and sharing your Jupyter notebooks online.

First, you need to install the Jovian python library if it isn't already installed.

```
!pip install jovian --upgrade --quiet
```

Next, the library needs to be imported.

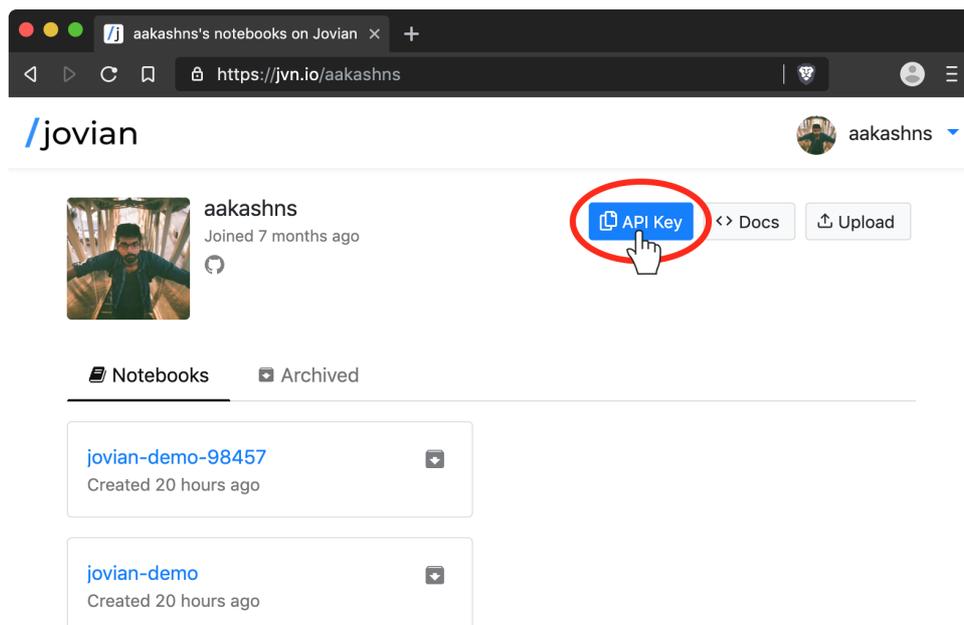
```
import jovian
```

Finally, you can run `jovian.commit` to capture and upload a snapshot of the notebook.

```
jovian.commit(project='first-steps-with-python')
```

```
[jovian] Updating notebook "evanmarie/first-steps-with-python" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/first-steps-with-python
'https://jovian.ai/evanmarie/first-steps-with-python'
```

The first time you run `jovian.commit`, you'll be asked to provide an *API Key* to securely upload the notebook to your Jovian account. You can get the API key from your [Jovian profile page](#) after logging in / signing up.



`jovian.commit` uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work. Jovian also includes a powerful commenting interface, so you can discuss & comment on specific parts of your notebook:

danb / keras-mnist-jovian
Updated 38 minutes ago

keras python

Notebook Files Records Collaborators

Wide Resnet22 (3 wee...)

View Diff Compare Present Share

Handwritten Digit Recognition

Objective: Classify handwritten digits from the MNIST dataset by training a convolutional neural network (CNN) using the [Keras](#) deep learning library.

```
In [3]: import matplotlib.pyplot as plt
f, axarr = plt.subplots(grid_size, grid_size)
for i in range(grid_size):
    for j in range(grid_size):
        ax = axarr[i, j]
        ax.imshow(train_images[i * grid_size + j], cmap='gray')
```

You can comment on a cell if you have any questions!

You can do a lot more with the `jovian` Python library. Visit the documentation site to learn more: <https://jovian.ai/docs/index.html>

Further Reading and References

Following are some resources where you can learn about more arithmetic, conditional and logical operations in Python:

- Python Tutorial at W3Schools: <https://www.w3schools.com/python/>
- Practical Python Programming: <https://dabeaz-course.github.io/practical-python/Notes/Contents.html>
- Python official documentation: <https://docs.python.org/3/tutorial/index.html>

Now that you have taken your first steps with Python, you are ready to move on to the next tutorial: "[A Quick Tour of Variables and Data Types in Python](#)".

Questions for Revision

Try answering the following questions to test your understanding of the topics covered in this notebook:

1. What is a Jupyter notebook?
2. How do you add a new code cell below an existing cell?
3. How do you add a new Markdown cell below an existing cell?
4. How do you convert a code cell to a Markdown cell or vice versa?
5. How do you execute a code cell within Jupyter?
6. What are the different arithmetic operations supported in Python?
7. How do you perform arithmetic operations using Python?
8. What is the difference between the `/` and the `//` operators?

9. What is the difference between the * and the ** operators?
10. What is the order of precedence for arithmetic operators in Python?
11. How do you specify the order in which arithmetic operations are performed in an expression involving multiple operators?
12. How do you solve a multi-step arithmetic word problem using Python?
13. What are variables? Why are they useful?
14. How do you create a variable in Python?
15. What is the assignment operator in Python?
16. What are the rules for naming a variable in Python?
17. How do you view the value of a variable?
18. How do you store the result of an arithmetic expression in a variable?
19. What happens if you try to access a variable that has not been defined?
20. How do you display messages in Python?
21. What type of inputs can the print function accept?
22. What are code comments? How are they useful?
23. What are the different ways of creating comments in Python code?
24. What are the different comparison operations supported in Python?
25. What is the result of a comparison operation?
26. What is the difference between = and == in Python?
27. What are the logical operators supported in Python?
28. What is the difference between the and and or operators?
29. Can you use comparison and logical operators in the same expression?
30. What is the purpose of using parentheses in arithmetic or logical expressions?
31. What is Markdown? Why is it useful?
32. How do you create headings of different sizes using Markdown?
33. How do you create bulleted and numbered lists using Markdown?
34. How do you create bold or italic text using Markdown?
35. How do you include links & images within Markdown cells?
36. How do you include code blocks within Markdown cells?
37. Is it possible to execute the code blocks within Markdown cells?
38. How do you upload and share your Jupyter notebook online using Jovian?
39. What is the purpose of the API key requested by jovian.commit ? Where can you find the API key?
40. Where can you learn about arithmetic, conditional and logical operations in Python?

Solution for Exercise

EXERCISE: A travel company wants to fly a plane to the Bahamas. Flying the plane costs 5000 dollars. So far, 29 people have signed up for the trip. If the company charges 200 dollars per ticket, what is the profit made by the company? Create variables for each numeric quantity and use appropriate arithmetic operations.

```
plane_cost=5000  
total_people_signed_up=29  
ticket_cost=200
```

```
amount_received_by_company=total_people_signed_up*ticket_cost
```

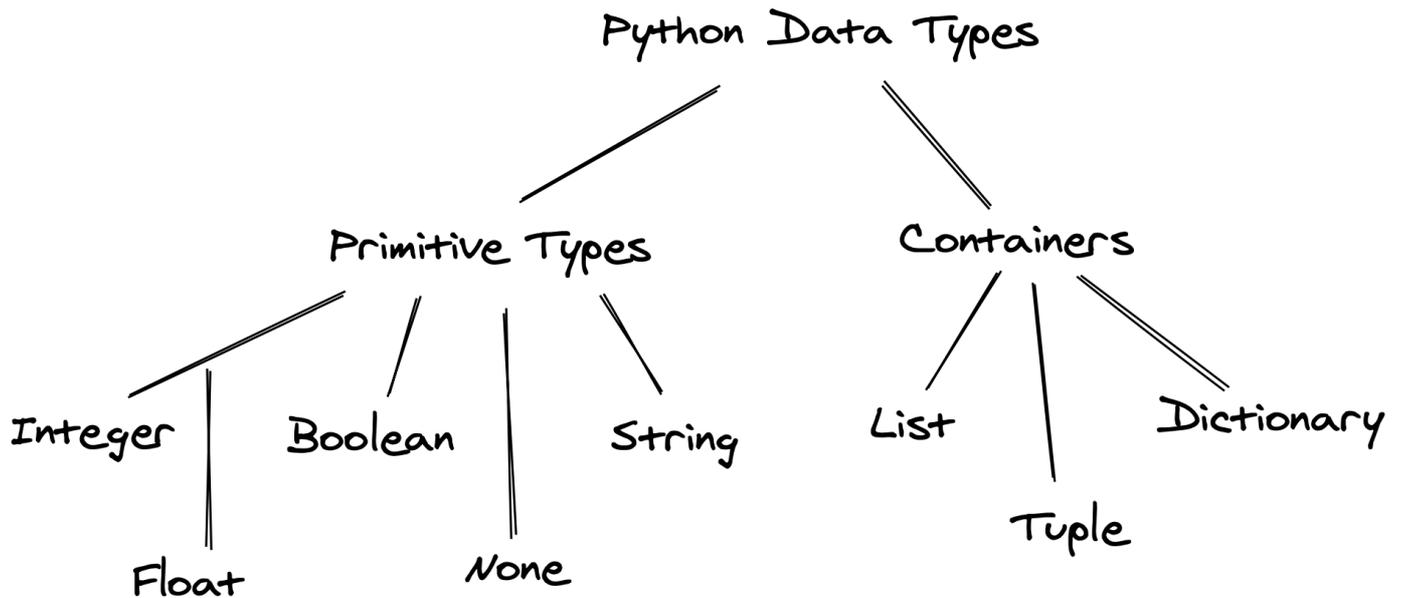
```
total_profit=amount_received_by_company-plane_cost
```

```
total_profit
```

800

The travel company is in a profit of \$800!

A Quick Tour of Variables and Data Types in Python



These tutorials take a practical and coding-focused approach. The best way to learn the material is to execute the code and experiment with it yourself.

This tutorial covers the following topics:

- Storing information using variables
- Primitive data types in Python: Integer, Float, Boolean, None and String
- Built-in data structures in Python: List, Tuple and Dictionary
- Methods and operators supported by built-in data types

How to run the code

This tutorial is an executable [Jupyter notebook](#) hosted on [Jovian](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Jupyter Notebooks: This tutorial is a [Jupyter notebook](#) - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc. instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things -

you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Storing information using variables

Computers are useful for two purposes: storing information (also known as data) and performing operations on stored data. While working with a programming language such as Python, data is stored in variables. You can think of variables as containers for storing data. The data stored within a variable is called its value. Creating variables in Python is pretty easy, as we've already seen in the [previous tutorial](#).

```
my_favorite_color = "blue"
```

```
my_favorite_color
```

```
'blue'
```

A variable is created using an assignment statement. It begins with the variable's name, followed by the assignment operator `=` followed by the value to be stored within the variable. Note that the assignment operator `=` is different from the equality comparison operator `==`.

You can also assign values to multiple variables in a single statement by separating the variable names and values with commas.

```
color1, color2, color3 = "red", "green", "blue"
```

```
color1
```

```
'red'
```

```
color2
```

```
'green'
```

```
color3
```

```
'blue'
```

You can assign the same value to multiple variables by chaining multiple assignment operations within a single statement.

```
color4 = color5 = color6 = "magenta"
```

```
color4
```

```
'magenta'
```

```
color5
```

```
'magenta'
```

```
color6
```

```
'magenta'
```

You can change the value stored within a variable by assigning a new value to it using another assignment statement. Be careful while reassigning variables: when you assign a new value to the variable, the old value is lost and no longer accessible.

```
my_favorite_color = "red"
```

```
my_favorite_color
```

```
'red'
```

While reassigning a variable, you can also use the variable's previous value to compute the new value.

```
counter = 10
```

```
counter = counter + 1
```

```
counter
```

```
11
```

The pattern `var = var op something` (where `op` is an arithmetic operator like `+`, `-`, `*`, `/`) is very common, so Python provides a *shorthand* syntax for it.

```
counter = 10
```

```
# Same as `counter = counter + 4`  
counter += 4
```

```
counter
```

```
14
```

Variable names can be short (`a`, `x`, `y`, etc.) or descriptive (`my_favorite_color`, `profit_margin`, `the_3_musketeers`, etc.). However, you must follow these rules while naming Python variables:

- A variable's name must start with a letter or the underscore character `_`. It cannot begin with a number.
- A variable name can only contain lowercase (small) or uppercase (capital) letters, digits, or underscores (`a-z`, `A-Z`, `0-9`, and `_`).
- Variable names are case-sensitive, i.e., `a_variable`, `A_Variable`, and `A_VARIABLE` are all different variables.

Here are some valid variable names:

```
a_variable = 23
is_today_Saturday = False
my_favorite_car = "Delorean"
the_3_musketeers = ["Athos", "Porthos", "Aramis"]
```

Let's try creating some variables with invalid names. Python prints a syntax error if your variable's name is invalid.

Syntax: The syntax of a programming language refers to the rules that govern the structure of a valid instruction or *statement*. If a statement does not follow these rules, Python stops execution and informs you that there is a *syntax error*. You can think of syntax as the rules of grammar for a programming language.

```
a variable = 23
```

```
File "/tmp/ipykernel_38/605469086.py", line 1
```

```
a variable = 23
```

```
^
```

SyntaxError: invalid syntax

```
is_today_Saturday = False
```

```
File "/tmp/ipykernel_38/3433388187.py", line 1
```

```
is_today_Saturday = False
```

```
^
```

SyntaxError: invalid syntax

```
my-favorite-car = "Delorean"
```

```
File "/tmp/ipykernel_38/1843242419.py", line 1
```

```
my-favorite-car = "Delorean"
```

```
^
```

SyntaxError: cannot assign to operator

```
3_musketeers = ["Athos", "Porthos", "Aramis"]
```

```
File "/tmp/ipykernel_38/3494872227.py", line 1
```

```
3_musketeers = ["Athos", "Porthos", "Aramis"]
```

```
^
```

SyntaxError: invalid decimal literal

Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. [Jovian](#) offers an easy way of saving and sharing your Jupyter notebooks online.

```
# Install the jovian library
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit(project='python-variables-and-data-types')
```

```
[jovian] Updating notebook "evanmarie/python-variables-and-data-types" on  
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-variables-and-data-types
```

```
'https://jovian.ai/evanmarie/python-variables-and-data-types'
```

The first time you run `jovian.commit`, you'll be asked to provide an API Key to securely upload the notebook to your Jovian account. You can get the API key from your [Jovian profile page](#) after logging in / signing up.

`jovian.commit` uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work.

Built-in data types in Python

Any data or information stored within a Python variable has a *type*. You can view the type of data stored within a variable using the `type` function.

```
a_variable
```

```
23
```

```
type(a_variable)
```

```
int
```

```
is_today_Saturday
```

```
False
```

```
type(is_today_Saturday)
```

```
bool
```

```
my_favorite_car
```

```
'Delorean'
```

```
type(my_favorite_car)
```

```
str
```

```
the_3_musketeers
```

```
['Athos', 'Porthos', 'Aramis']
```

```
type(the_3_musketeers)
```

```
list
```

Python has several built-in data types for storing different kinds of information in variables. Following are some commonly used data types:

1. Integer
2. Float
3. Boolean
4. None
5. String
6. List
7. Tuple
8. Dictionary

Integer, float, boolean, None, and string are *primitive data types* because they represent a single value. Other data types like list, tuple, and dictionary are often called *data structures* or *containers* because they hold multiple pieces of data together.

Integer

Integers represent positive or negative whole numbers, from negative infinity to infinity. Note that integers should not include decimal points. Integers have the type `int`.

```
current_year = 2020
```

```
current_year
```

```
2020
```

```
type(current_year)
```

```
int
```

Unlike some other programming languages, integers in Python can be arbitrarily large (or small). There's no lowest or highest value for integers, and there's just one `int` type (as opposed to `short`, `int`, `long`, `long long`, `unsigned int`, etc. in C/C++/Java).

```
a_large_negative_number = -23374038374832934334234317348343
```

```
a_large_negative_number
```

```
-23374038374832934334234317348343
```

```
type(a_large_negative_number)
```

int

Float

Floats (or floating-point numbers) are numbers with a decimal point. There are no limits on the value or the number of digits before or after the decimal point. Floating-point numbers have the type `float`.

```
pi = 3.141592653589793238
```

```
pi
```

3.141592653589793

```
type(pi)
```

float

Note that a whole number is treated as a float if written with a decimal point, even though the decimal portion of the number is zero.

```
a_number = 3.0
```

```
a_number
```

3.0

```
type(a_number)
```

float

```
another_number = 4.
```

```
another_number
```

4.0

```
type(another_number)
```

float

Floating point numbers can also be written using the scientific notation with an "e" to indicate the power of 10.

```
one_hundredth = 1e-2
```

```
one_hundredth
```

0.01

```
type(one_hundredth)
```

float

```
avogadro_number = 6.02214076e23
```

```
avogadro_number
```

6.02214076e+23

```
type(avogadro_number)
```

float

You can convert floats into integers and vice versa using the `float` and `int` functions. The operation of converting one type of value into another is called casting.

```
float(current_year)
```

2020.0

```
float(a_large_negative_number)
```

-2.3374038374832935e+31

```
int(pi)
```

3

```
int(avogadro_number)
```

602214075999999987023872

While performing arithmetic operations, integers are automatically converted to `float`s if any of the operands is a `float`. Also, the division operator `/` always returns a `float`, even if both operands are integers. Use the `//` operator if you want the result of the division to be an `int`.

```
type(45 * 3.0)
```

float

```
type(45 * 3)
```

int

```
type(10/3)
```

float

```
type(10/2)
```

float

```
type(10//2)
```

int

Boolean

Booleans represent one of 2 values: `True` and `False`. Booleans have the type `bool`.

```
is_today_Sunday = True
```

```
is_today_Sunday
```

True

```
type(is_today_Saturday)
```

bool

Booleans are generally the result of a comparison operation, e.g., `==`, `>=`, etc.

```
cost_of_ice_bag = 1.25  
is_ice_bag_expensive = cost_of_ice_bag >= 10
```

```
is_ice_bag_expensive
```

False

```
type(is_ice_bag_expensive)
```

bool

Booleans are automatically converted to `int`s when used in arithmetic operations. `True` is converted to `1` and `False` is converted to `0`.

```
5 + False
```

5

```
3. + True
```

4.0

Any value in Python can be converted to a Boolean using the `bool` function.

Only the following values evaluate to `False` (they are often called *falsy* values):

1. The value `False` itself
2. The integer `0`
3. The float `0.0`
4. The empty value `None`
5. The empty text `""`
6. The empty list `[]`
7. The empty tuple `()`
8. The empty dictionary `{}`
9. The empty set `set()`
10. The empty range `range(0)`

Everything else evaluates to `True` (a value that evaluates to `True` is often called a *truthy* value).

```
bool(False)
```

False

```
bool(0)
```

False

```
bool(0.0)
```

False

```
bool(None)
```

False

```
bool("")
```

False

```
bool([])
```

False

```
bool(())
```

False

```
bool({})
```

False

```
bool(set())
```

False

```
bool(range(0))
```

False

```
bool(True), bool(1), bool(2.0), bool("hello"), bool([1,2]), bool((2,3)), bool(range(10))
```

(True, True, True, True, True, True, True)

None

The None type includes a single value `None`, used to indicate the absence of a value. `None` has the type `NoneType`. It is often used to declare a variable whose value may be assigned later.

```
nothing = None
```

```
type(nothing)
```

NoneType

String

A string is used to represent text (*a string of characters*) in Python. Strings must be surrounded using quotations (either the single quote `'` or the double quote `"`). Strings have the type `string`.

```
today = "Saturday"
```

```
today
```

'Saturday'

```
type(today)
```

str

You can use single quotes inside a string written with double quotes, and vice versa.

```
my_favorite_movie = "One Flew over the Cuckoo's Nest"
```

```
my_favorite_movie
```

"One Flew over the Cuckoo's Nest"

```
my_favorite_pun = 'Thanks for explaining the word "many" to me, it means a lot.'
```

```
my_favorite_pun
```

```
'Thanks for explaining the word "many" to me, it means a lot.'
```

To use a double quote within a string written with double quotes, *escape* the inner quotes by prefixing them with the `\` character.

```
another_pun = "The first time I got a universal remote control, I thought to myself \"T
```

```
another_pun
```

```
'The first time I got a universal remote control, I thought to myself "This changes everything".'
```

Strings created using single or double quotes must begin and end on the same line. To create multiline strings, use three single quotes `'''` or three double quotes `"""` to begin and end the string. Line breaks are represented using the newline character `\n`.

```
yet_another_pun = '''Son: "Dad, can you tell me what a solar eclipse is?"  
Dad: "No sun."'''
```

```
yet_another_pun
```

```
'Son: "Dad, can you tell me what a solar eclipse is?" \nDad: "No sun."'
```

Multiline strings are best displayed using the `print` function.

```
print(yet_another_pun)
```

```
Son: "Dad, can you tell me what a solar eclipse is?"
```

```
Dad: "No sun."
```

```
a_music_pun = """  
Two windmills are standing in a field and one asks the other,  
"What kind of music do you like?"  
  
The other says,  
"I'm a big metal fan."  
"""
```

```
print(a_music_pun)
```

```
Two windmills are standing in a field and one asks the other,  
"What kind of music do you like?"
```

```
The other says,
```

```
"I'm a big metal fan."
```

You can check the length of a string using the `len` function.

```
len(my_favorite_movie)
```

```
31
```

Note that special characters like `\n` and escaped characters like `\"` count as a single character, even though they are written and sometimes printed as two characters.

```
multiline_string = """a
b"""
multiline_string
```

```
'a\nb'
```

```
len(multiline_string)
```

```
3
```

A string can be converted into a list of characters using `list` function.

```
list(multiline_string)
```

```
['a', '\n', 'b']
```

Strings also support several list operations, which are discussed in the next section. We'll look at a couple of examples here.

You can access individual characters within a string using the `[]` indexing notation. Note the character indices go from `0` to `n-1`, where `n` is the length of the string.

```
today = "Saturday"
```

```
today[0]
```

```
'S'
```

```
today[3]
```

```
'u'
```

```
today[7]
```

```
'y'
```

You can access a part of a string using by providing a `start:end` range instead of a single index in `[]`.

```
today[5:8]
```

```
'day'
```

You can also check whether a string contains a some text using the `in` operator.

```
'day' in today
```

```
True
```

```
'Sun' in today
```

```
False
```

Two or more strings can be joined or *concatenated* using the `+` operator. Be careful while concatenating strings, sometimes you may need to add a space character `" "` between words.

```
full_name = "Derek O'Brien"
```

```
greeting = "Hello"
```

```
greeting + full_name
```

```
"HelloDerek O'Brien"
```

```
greeting + " " + full_name + "!" # additional space
```

```
"Hello Derek O'Brien!"
```

Strings in Python have many built-in *methods* that are used to manipulate them. Let's try out some common string methods.

Methods: Methods are functions associated with data types and are accessed using the `.` notation e.g. `variable_name.method()` or `"a string".method()`. Methods are a powerful technique for associating common operations with values of specific data types.

The `.lower()`, `.upper()` and `.capitalize()` methods are used to change the case of the characters.

```
today.lower()
```

```
'saturday'
```

```
"saturday".upper()
```

```
'SATURDAY'
```

```
"monday".capitalize() # changes first character to uppercase
```

```
'Monday'
```

The `.replace` method replaces a part of the string with another string. It takes the portion to be replaced and the replacement text as *inputs* or *arguments*.

```
another_day = today.replace("Satur", "Wednes")
```

```
another_day
```

```
'Wednesday'
```

Note that `replace` returns a new string, and the original string is not modified.

```
today
```

```
'Saturday'
```

The `.split` method splits a string into a list of strings at every occurrence of provided character(s).

```
"Sun,Mon,Tue,Wed,Thu,Fri,Sat".split(",")
```

```
['Sun', 'Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat']
```

The `.strip` method removes whitespace characters from the beginning and end of a string.

```
a_long_line = "      This is a long line with some space before, after,      and some s
```

```
a_long_line_stripped = a_long_line.strip()
```

```
a_long_line_stripped
```

```
'This is a long line with some space before, after,      and some space in the middle..'
```

The `.format` method combines values of other data types, e.g., integers, floats, booleans, lists, etc. with strings. You can use `format` to construct output messages for display.

```
# Input variables
cost_of_ice_bag = 1.25
profit_margin = .2
number_of_bags = 500

# Template for output message
output_template = """If a grocery store sells ice bags at $ {} per bag, with a profit margin of {}%,
then the total profit it makes by selling {} ice bags is $ {}."""

print(output_template)
```

If a grocery store sells ice bags at \$ {} per bag, with a profit margin of {} %, then the total profit it makes by selling {} ice bags is \$ {}.

```
# Inserting values into the string
total_profit = cost_of_ice_bag * profit_margin * number_of_bags
output_message = output_template.format(cost_of_ice_bag, profit_margin*100, number_of_bags)

print(output_message)
```

If a grocery store sells ice bags at \$ 1.25 per bag, with a profit margin of 20.0 %, then the total profit it makes by selling 500 ice bags is \$ 125.0.

Notice how the placeholders {} in the output_template string are replaced with the arguments provided to the .format method.

It is also possible to use the string concatenation operator + to combine strings with other values. However, those values must first be converted to strings using the str function.

```
"If a grocery store sells ice bags at $ " + cost_of_ice_bag + ", with a profit margin of " + profit_margin
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-127-78b7958ec7cc> in <module>
----> 1 "If a grocery store sells ice bags at $ " + cost_of_ice_bag + ", with a profit
margin of " + profit_margin
```

TypeError: can only concatenate str (not "float") to str

```
"If a grocery store sells ice bags at $ " + str(cost_of_ice_bag) + ", with a profit margin of " + profit_margin
```

```
'If a grocery store sells ice bags at $ 1.25, with a profit margin of 0.2'
```

You can str to convert a value of any data type into a string.

```
str(23)
```

```
'23'
```

```
str(23.432)
```

```
'23.432'
```

```
str(True)
```

```
'True'
```

```
the_3_musketeers = ["Athos", "Porthos", "Aramis"]
str(the_3_musketeers)
```

```
"['Athos', 'Porthos', 'Aramis']"
```

Note that all string methods return new values and DO NOT change the existing string. You can find a full list of string methods here: https://www.w3schools.com/python/python_ref_string.asp.

Strings also support the comparison operators `==` and `!=` for checking whether two strings are equal.

```
first_name = "John"
```

```
first_name == "Doe"
```

False

```
first_name == "John"
```

True

```
first_name != "Jane"
```

True

We've looked at the primitive data types in Python. We're now ready to explore non-primitive data structures, also known as containers.

Before continuing, let us run `jovian.commit` once again to record another snapshot of our notebook.

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

```
[jovian] Updating notebook "samanvitha/python-variables-and-data-types" on  
https://jovian.ai
```

```
[jovian] Uploading notebook..
```

```
[jovian] Uploading additional files...
```

```
[jovian] Committed successfully! https://jovian.ai/samanvitha/python-variables-and-data-types
```

```
'https://jovian.ai/samanvitha/python-variables-and-data-types'
```

Running `jovian.commit` multiple times within a notebook records new versions automatically. You will continue to have access to all the previous versions of your notebook, using the versions dropdown on the notebook's Jovian.

List

A list in Python is an ordered collection of values. Lists can hold values of different data types and support operations to add, remove, and change values. Lists have the type `list`.

To create a list, enclose a sequence of values within square brackets `[` and `]`, separated by commas.

```
fruits = ['apple', 'banana', 'cherry']
```

```
fruits
```

```
['apple', 'banana', 'cherry']
```

```
type(fruits)
```

```
list
```

Let's try creating a list containing values of different data types, including another list.

```
a_list = [23, 'hello', None, 3.14, fruits, 3 <= 5]
```

```
a_list
```

```
[23, 'hello', None, 3.14, ['apple', 'banana', 'cherry'], True]
```

```
empty_list = []
```

```
empty_list
```

```
[]
```

To determine the number of values in a list, use the `len` function. You can use `len` to determine the number of values in several other data types.

```
len(fruits)
```

```
3
```

```
print("Number of fruits:", len(fruits))
```

```
Number of fruits: 3
```

```
len(a_list)
```

```
6
```

```
len(empty_list)
```

```
0
```

You can access an element from the list using its *index*, e.g., `fruits[2]` returns the element at index 2 within the list `fruits`. The starting index of a list is 0.

```
fruits[0]
```

```
'apple'
```

```
fruits[1]
```

```
'banana'
```

```
fruits[2]
```

```
'cherry'
```

If you try to access an index equal to or higher than the length of the list, Python returns an `IndexError` .

```
fruits[3]
```

```
-----  
IndexError                                Traceback (most recent call last)  
<ipython-input-152-7ceeafd384d7> in <module>  
----> 1 fruits[3]
```

`IndexError`: list index out of range

```
fruits[4]
```

```
-----  
IndexError                                Traceback (most recent call last)  
<ipython-input-153-b8c91da6ba3a> in <module>  
----> 1 fruits[4]
```

`IndexError`: list index out of range

You can use negative indices to access elements from the end of a list, e.g., `fruits[-1]` returns the last element, `fruits[-2]` returns the second last element, and so on.

```
fruits[-1]
```

```
'cherry'
```

```
fruits[-2]
```

```
'banana'
```

```
fruits[-3]
```

```
'apple'
```

```
fruits[-4]
```

```
-----  
IndexError                                Traceback (most recent call last)  
<ipython-input-157-1cb2d66442ee> in <module>  
----> 1 fruits[-4]
```

`IndexError`: list index out of range

You can also access a range of values from the list. The result is itself a list. Let us look at some examples.

```
a_list = [23, 'hello', None, 3.14, fruits, 3 <= 5]
```

```
a_list
```

```
[23, 'hello', None, 3.14, ['apple', 'banana', 'cherry'], True]
```

```
len(a_list)
```

```
6
```

```
a_list[2:5]
```

```
[None, 3.14, ['apple', 'banana', 'cherry']]
```

Note that the range `2:5` includes the element at the start index `2` but does not include the element at the end index `5`. So, the result has 3 values (index `2`, `3`, and `4`).

Here are some experiments you should try out (use the empty cells below):

- Try setting one or both indices of the range are larger than the size of the list, e.g., `a_list[2:10]`
- Try setting the start index of the range to be larger than the end index, e.g., `a_list[12:10]`
- Try leaving out the start or end index of a range, e.g., `a_list[2:]` or `a_list[:5]`
- Try using negative indices for the range, e.g., `a_list[-2:-5]` or `a_list[-5:-2]` (can you explain the results?)

The flexible and interactive nature of Jupyter notebooks makes them an excellent tool for learning and experimentation. If you are new to Python, you can resolve most questions as soon as they arise simply by typing the code into a cell and executing it. Let your curiosity run wild, discover what Python is capable of and what it isn't!

You can also change the value at a specific index within a list using the assignment operation.

```
fruits
```

```
['apple', 'banana', 'cherry']
```

```
fruits[1] = 'blueberry'
```

```
fruits
```

```
['apple', 'blueberry', 'cherry']
```

A new value can be added to the end of a list using the `append` method.

```
fruits.append('dates')
```

```
fruits
```

```
['apple', 'blueberry', 'cherry', 'dates']
```

A new value can also be inserted at a specific index using the `insert` method.

```
fruits.insert(1, 'banana')
```

```
fruits
```

```
['apple', 'banana', 'blueberry', 'cherry', 'dates']
```

You can remove a value from a list using the `remove` method.

```
fruits.remove('blueberry')
```

```
fruits
```

```
['apple', 'banana', 'cherry', 'dates']
```

What happens if a list has multiple instances of the value passed to `.remove` ? Try it out.

To remove an element from a specific index, use the `pop` method. The method also returns the removed element.

```
fruits
```

```
['apple', 'banana', 'cherry', 'dates']
```

```
fruits.pop(1)
```

```
'banana'
```

```
fruits
```

```
['apple', 'cherry', 'dates']
```

If no index is provided, the `pop` method removes the last element of the list.

```
fruits.pop()
```

```
'dates'
```

```
fruits
```

```
['apple', 'cherry']
```

You can test whether a list contains a value using the `in` operator.

```
'pineapple' in fruits
```

```
False
```

```
'cherry' in fruits
```

```
True
```

To combine two or more lists, use the `+` operator. This operation is also called *concatenation*.

```
fruits
```

```
['apple', 'cherry']
```

```
more_fruits = fruits + ['pineapple', 'tomato', 'guava'] + ['dates', 'banana']
```

```
more_fruits
```

```
['apple', 'cherry', 'pineapple', 'tomato', 'guava', 'dates', 'banana']
```

To create a copy of a list, use the `copy` method. Modifying the copied list does not affect the original.

```
more_fruits_copy = more_fruits.copy()
```

```
more_fruits_copy
```

```
['apple', 'cherry', 'pineapple', 'tomato', 'guava', 'dates', 'banana']
```

```
# Modify the copy  
more_fruits_copy.remove('pineapple')  
more_fruits_copy.pop()  
more_fruits_copy
```

```
['apple', 'cherry', 'tomato', 'guava', 'dates']
```

```
# Original list remains unchanged
```

```
more_fruits
```

```
['apple', 'cherry', 'pineapple', 'tomato', 'guava', 'dates', 'banana']
```

Note that you cannot create a copy of a list by simply creating a new variable using the assignment operator = . The new variable will point to the same list, and any modifications performed using either variable will affect the other.

```
more_fruits
```

```
['apple', 'cherry', 'pineapple', 'tomato', 'guava', 'dates', 'banana']
```

```
more_fruits_not_a_copy = more_fruits
```

```
more_fruits_not_a_copy.remove('pineapple')
```

```
more_fruits_not_a_copy.pop()
```

```
'banana'
```

```
more_fruits_not_a_copy
```

```
['apple', 'cherry', 'tomato', 'guava', 'dates']
```

```
more_fruits
```

```
['apple', 'cherry', 'tomato', 'guava', 'dates']
```

Just like strings, there are several in-built methods to manipulate a list. However, unlike strings, most list methods modify the original list rather than returning a new one. Check out some common list operations here:

https://www.w3schools.com/python/python_ref_list.asp .

Following are some exercises you can try out with list methods (use the blank code cells below):

- Reverse the order of elements in a list
- Add the elements of one list at the end of another list
- Sort a list of strings in alphabetical order
- Sort a list of numbers in decreasing order

Tuple

A tuple is an ordered collection of values, similar to a list. However, it is not possible to add, remove, or modify values in a tuple. A tuple is created by enclosing values within parentheses (and), separated by commas.

Any data structure that cannot be modified after creation is called *immutable*. You can think of tuples as immutable lists.

Let's try some experiments with tuples.

```
fruits = ('apple', 'cherry', 'dates')
```

```
# check no. of elements  
len(fruits)
```

3

```
# get an element (positive index)  
fruits[0]
```

'apple'

```
# get an element (negative index)  
fruits[-2]
```

'cherry'

```
# check if it contains an element  
'dates' in fruits
```

True

```
# try to change an element  
fruits[0] = 'avocado'
```

```
TypeError                                 Traceback (most recent call last)  
<ipython-input-195-eea1d48cc8b5> in <module>  
      1 # try to change an element  
----> 2 fruits[0] = 'avocado'
```

TypeError: 'tuple' object does not support item assignment

```
# try to append an element  
fruits.append('blueberry')
```

```
AttributeError                             Traceback (most recent call last)  
<ipython-input-196-e5ea20adaaf8> in <module>
```

```
1 # try to append an element
----> 2 fruits.append('blueberry')
```

AttributeError: 'tuple' object has no attribute 'append'

```
# try to remove an element
fruits.remove('apple')
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-197-37543d45b6b4> in <module>
    1 # try to remove an element
----> 2 fruits.remove('apple')
```

AttributeError: 'tuple' object has no attribute 'remove'

You can also skip the parantheses (and) while creating a tuple. Python automatically converts comma-separated values into a tuple.

```
the_3_musketeers = 'Athos', 'Porthos', 'Aramis'
```

```
the_3_musketeers
```

```
('Athos', 'Porthos', 'Aramis')
```

You can also create a tuple with just one element by typing a comma after it. Just wrapping it with parantheses (and) won't make it a tuple.

```
single_element_tuple = 4,
```

```
single_element_tuple
```

```
(4,)
```

```
another_single_element_tuple = (4,)
```

```
another_single_element_tuple
```

```
(4,)
```

```
not_a_tuple = (4)
```

```
not_a_tuple
```

```
4
```

Tuples are often used to create multiple variables with a single statement.

```
point = (3, 4)
```

```
point_x, point_y = point
```

```
point_x
```

3

```
point_y
```

4

You can convert a list into a tuple using the `tuple` function, and vice versa using the `list` function

```
tuple(['one', 'two', 'three'])
```

```
('one', 'two', 'three')
```

```
list(('Athos', 'Porthos', 'Aramis'))
```

```
['Athos', 'Porthos', 'Aramis']
```

Tuples have just two built-in methods: `count` and `index`. Can you figure out what they do? While you look could look for documentation and examples online, there's an easier way to check a method's documentation, using the `help` function.

```
a_tuple = 23, "hello", False, None, 23, 37, "hello"
```

```
help(a_tuple.count)
```

Help on built-in function count:

```
count(value, /) method of builtins.tuple instance  
    Return number of occurrences of value.
```

Within a Jupyter notebook, you can also start a code cell with `?` and type the name of a function or method. When you execute this cell, you will see the function/method's documentation in a pop-up window.

```
?a_tuple.index
```

Try using `count` and `index` with `a_tuple` in the code cells below.

Dictionary

A dictionary is an unordered collection of items. Each item stored in a dictionary has a key and value. You can use a key to retrieve the corresponding value from the dictionary. Dictionaries have the type `dict`.

Dictionaries are often used to store many pieces of information e.g. details about a person, in a single variable. Dictionaries are created by enclosing key-value pairs within braces or curly brackets `{` and `}`.

```
person1 = {  
    'name': 'John Doe',  
    'sex': 'Male',  
    'age': 32,  
    'married': True  
}
```

```
person1
```

```
{'name': 'John Doe', 'sex': 'Male', 'age': 32, 'married': True}
```

Dictionaries can also be created using the `dict` function.

```
person2 = dict(name='Jane Judy', sex='Female', age=28, married=False)
```

```
person2
```

```
{'name': 'Jane Judy', 'sex': 'Female', 'age': 28, 'married': False}
```

```
type(person1)
```

```
dict
```

Keys can be used to access values using square brackets `[` and `]`.

```
person1['name']
```

```
'John Doe'
```

```
person1['married']
```

```
True
```

```
person2['name']
```

```
'Jane Judy'
```

If a key isn't present in the dictionary, then a `KeyError` is *thrown*.

```
person1['address']
```

```
KeyError                                Traceback (most recent call last)
<ipython-input-223-f2b8bd2476d4> in <module>
----> 1 person1['address']
```

KeyError: 'address'

You can also use the `get` method to access the value associated with a key.

```
person2.get("name")
```

'Jane Judy'

The `get` method also accepts a default value, returned if the key is not present in the dictionary.

```
person2.get("address", "Unknown")
```

'Unknown'

You can check whether a key is present in a dictionary using the `in` operator.

```
'name' in person1
```

True

```
'address' in person1
```

False

You can change the value associated with a key using the assignment operator.

```
person2['married']
```

False

```
person2['married'] = True
```

```
person2['married']
```

True

The assignment operator can also be used to add new key-value pairs to the dictionary.

```
person1
```

```
{'name': 'John Doe', 'sex': 'Male', 'age': 32, 'married': True}
```

```
person1['address'] = '1, Penny Lane'
```

```
person1
```

```
{'name': 'John Doe',  
 'sex': 'Male',  
 'age': 32,  
 'married': True,  
 'address': '1, Penny Lane'}
```

To remove a key and the associated value from a dictionary, use the `pop` method.

```
person1.pop('address')
```

```
'1, Penny Lane'
```

```
person1
```

```
{'name': 'John Doe', 'sex': 'Male', 'age': 32, 'married': True}
```

Dictionaries also provide methods to view the list of keys, values, or key-value pairs inside it.

```
person1.keys()
```

```
dict_keys(['name', 'sex', 'age', 'married'])
```

```
person1.values()
```

```
dict_values(['John Doe', 'Male', 32, True])
```

```
person1.items()
```

```
dict_items([('name', 'John Doe'), ('sex', 'Male'), ('age', 32), ('married', True)])
```

```
person1.items()[1]
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-239-b9fb361f2a9e> in <module>  
----> 1 person1.items()[1]
```

TypeError: 'dict_items' object is not subscriptable

The results of `keys`, `values`, and `items` look like lists. However, they don't support the indexing operator `[]` for retrieving elements.

Can you figure out how to access an element at a specific index from these results? Try it below. *Hint: Use the `list` function*

Dictionaries provide many other methods. You can learn more about them here:

https://www.w3schools.com/python/python_ref_dictionary.asp.

Here are some experiments you can try out with dictionaries (use the empty cells below):

- What happens if you use the same key multiple times while creating a dictionary?
- How can you create a copy of a dictionary (modifying the copy should not change the original)?
- Can the value associated with a key itself be a dictionary?
- How can you add the key-value pairs from one dictionary into another dictionary? Hint: See the update method.
- Can the dictionary's keys be something other than a string, e.g., a number, boolean, list, etc.?

Further Reading

We've now completed our exploration of variables and common data types in Python. Following are some resources to learn more about data types in Python:

- Python official documentation: <https://docs.python.org/3/tutorial/index.html>
- Python Tutorial at W3Schools: <https://www.w3schools.com/python/>
- Practical Python Programming: <https://dabeaz-course.github.io/practical-python/Notes/Contents.html>

You are now ready to move on to the next tutorial: [Branching using conditional statements and loops in Python](#)

Let's save a snapshot of our notebook one final time using `jovian.commit`.

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
[jovian] Updating notebook "samanvitha/python-variables-and-data-types" on
https://jovian.ai
[jovian] Uploading notebook..
[jovian] Uploading additional files...
```

[jovian] Committed successfully! <https://jovian.ai/samanvitha/python-variables-and-data-types>

'<https://jovian.ai/samanvitha/python-variables-and-data-types>'

Questions for Revision

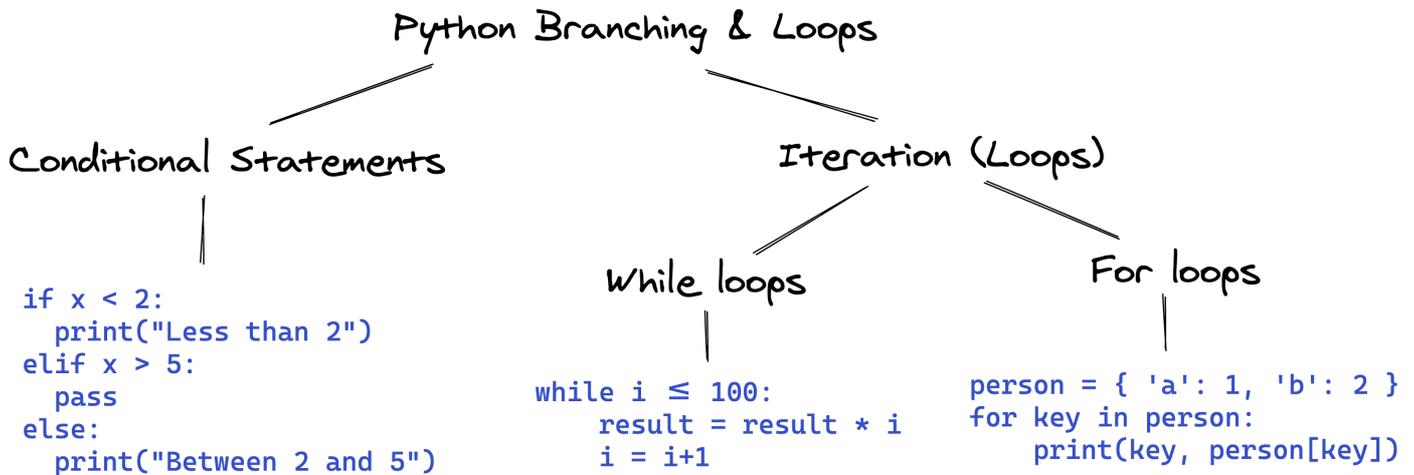
Try answering the following questions to test your understanding of the topics covered in this notebook:

1. What is a variable in Python?
2. How do you create a variable?
3. How do you check the value within a variable?
4. How do you create multiple variables in a single statement?
5. How do you create multiple variables with the same value?
6. How do you change the value of a variable?
7. How do you reassign a variable by modifying the previous value?
8. What does the statement `counter += 4` do?
9. What are the rules for naming a variable?
10. Are variable names case-sensitive? Do `a_variable`, `A_Variable`, and `A_VARIABLE` represent the same variable or different ones?
11. What is Syntax? Why is it important?
12. What happens if you execute a statement with invalid syntax?
13. How do you check the data type of a variable?
14. What are the built-in data types in Python?
15. What is a primitive data type?
16. What are the primitive data types available in Python?
17. What is a data structure or container data type?
18. What are the container types available in Python?
19. What kind of data does the Integer data type represent?
20. What are the numerical limits of the integer data type?
21. What kind of data does the float data type represent?
22. How does Python decide if a given number is a float or an integer?
23. How can you create a variable which stores a whole number, e.g., 4 but has the float data type?
24. How do you create floats representing very large (e.g., 6.023×10^{23}) or very small numbers (0.000000123)?
25. What does the expression $23e-12$ represent?
26. Can floats be used to store numbers with unlimited precision?
27. What are the differences between integers and floats?
28. How do you convert an integer to a float?
29. How do you convert a float to an integer?
30. What is the result obtained when you convert 1.99 to an integer?

31. What are the data types of the results of the division operators / and //?
32. What kind of data does the Boolean data type represent?
33. Which types of Python operators return booleans as a result?
34. What happens if you try to use a boolean in arithmetic operation?
35. How can any value in Python be covered to a boolean?
36. What are truthy and falsy values?
37. What are the values in Python that evaluate to False?
38. Give some examples of values that evaluate to True.
39. What kind of data does the None data type represent?
40. What is the purpose of None?
41. What kind of data does the String data type represent?
42. What are the different ways of creating strings in Python?
43. What is the difference between strings creating using single quotes, i.e. ' and ' vs. those created using double quotes, i.e. " and "?
44. How do you create multi-line strings in Python?
45. What is the newline character, \n?
46. What are escaped characters? How are they useful?
47. How do you check the length of a string?
48. How do you convert a string into a list of characters?
49. How do you access a specific character from a string?
50. How do you access a range of characters from a string?
51. How do you check if a specific character occurs in a string?
52. How do you check if a smaller string occurs within a bigger string?
53. How do you join two or more strings?
54. What are "methods" in Python? How are they different from functions?
55. What do the .lower, .upper and .capitalize methods on strings do?
56. How do you replace a specific part of a string with something else?
57. How do you split the string "Sun,Mon,Tue,Wed,Thu,Fri,Sat" into a list of days?
58. How do you remove whitespace from the beginning and end of a string?
59. What is the string .format method used for? Can you give an example?
60. What are the benefits of using the .format method instead of string concatenation?
61. How do you convert a value of another type to a string?
62. How do you check if two strings have the same value?
63. Where can you find the list of all the methods supported by strings?
64. What is a list in Python?
65. How do you create a list?
66. Can a Python list contain values of different data types?

67. Can a list contain another list as an element within it?
68. Can you create a list without any values?
69. How do you check the length of a list in Python?
70. How do you retrieve a value from a list?
71. What is the smallest and largest index you can use to access elements from a list containing five elements?
72. What happens if you try to access an index equal to or larger than the size of a list?
73. What happens if you try to access a negative index within a list?
74. How do you access a range of elements from a list?
75. How many elements does the list returned by the expression `a_list[2:5]` contain?
76. What do the ranges `a_list[:2]` and `a_list[2:]` represent?
77. How do you change the item stored at a specific index within a list?
78. How do you insert a new item at the beginning, middle, or end of a list?
79. How do you remove an item from a list?
80. How do you remove the item at a given index from a list?
81. How do you check if a list contains a value?
82. How do you combine two or more lists to create a larger list?
83. How do you create a copy of a list?
84. Does the expression `a_new_list = a_list` create a copy of the list `a_list`?
85. Where can you find the list of all the methods supported by lists?
86. What is a Tuple in Python?
87. How is a tuple different from a list?
88. Can you add or remove elements in a tuple?
89. How do you create a tuple with just one element?
90. How do you convert a tuple to a list and vice versa?
91. What are the `count` and `index` methods of a Tuple used for?
92. What is a dictionary in Python?
93. How do you create a dictionary?
94. What are keys and values?
95. How do you access the value associated with a specific key in a dictionary?
96. What happens if you try to access the value for a key that doesn't exist in a dictionary?
97. What is the `.get` method of a dictionary used for?
98. How do you change the value associated with a key in a dictionary?
99. How do you add or remove a key-value pair in a dictionary?
100. How do you access the keys, values, and key-value pairs within a dictionary?

Branching using Conditional Statements and Loops in Python



This tutorial covers the following topics:

- Branching with `if`, `else` and `elif`
- Nested conditions and `if` expressions
- Iteration with `while` loops
- Iterating over containers with `for` loops
- Nested loops, `break` and `continue` statements

How to run the code

This tutorial is an executable [Jupyter notebook](#) hosted on [Jovian](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Jupyter Notebooks: This tutorial is a [Jupyter notebook](#) - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc., instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Branching with `if`, `else` and `elif`

One of the most powerful features of programming languages is *branching*: the ability to make decisions and execute a different set of statements based on whether one or more conditions are true.

The `if` statement

In Python, branching is implemented using the `if` statement, which is written as follows:

```
if condition:
    statement1
    statement2
```

The `condition` can be a value, variable or expression. If the condition evaluates to `True`, then the statements within the *if block* are executed. Notice the four spaces before `statement1`, `statement2`, etc. The spaces inform Python that these statements are associated with the `if` statement above. This technique of structuring code by adding spaces is called *indentation*.

Indentation: Python relies heavily on *indentation* (white space before a statement) to define code structure. This makes Python code easy to read and understand. You can run into problems if you don't use indentation properly. Indent your code by placing the cursor at the start of the line and pressing the `Tab` key once to add 4 spaces. Pressing `Tab` again will indent the code further by 4 more spaces, and press `Shift+Tab` will reduce the indentation by 4 spaces.

For example, let's write some code to check and print a message if a given number is even.

```
a_number = 34
```

```
if a_number % 2 == 0:
    print("We're inside an if block")
    print('The given number {} is even.'.format(a_number))
```

We're inside an if block

The given number 34 is even.

We use the modulus operator `%` to calculate the remainder from the division of `a_number` by `2`. Then, we use the comparison operator `==` check if the remainder is `0`, which tells us whether the number is even, i.e., divisible by 2.

Since `34` is divisible by `2`, the expression `a_number % 2 == 0` evaluates to `True`, so the `print` statement under the `if` statement is executed. Also, note that we are using the string `format` method to include the number within the message.

Let's try the above again with an odd number.

```
another_number = 33
```

```
if another_number % 2 == 0:
    print('The given number {} is even.'.format(another_number))
```

As expected, since the condition `another_number % 2 == 0` evaluates to `False`, no message is printed.

The `else` statement

We may want to print a different message if the number is not even in the above example. This can be done by adding the `else` statement. It is written as follows:

```
if condition:
    statement1
    statement2
else:
    statement4
    statement5
```

If `condition` evaluates to `True`, the statements in the `if` block are executed. If it evaluates to `False`, the statements in the `else` block are executed.

```
a_number = 34
```

```
if a_number % 2 == 0:
    print('The given number {} is even.'.format(a_number))
else:
    print('The given number {} is odd.'.format(a_number))
```

The given number 34 is even.

```
another_number = 33
```

```
if another_number % 2 == 0:
    print('The given number {} is even.'.format(another_number))
else:
    print('The given number {} is odd.'.format(another_number))
```

The given number 33 is odd.

Here's another example, which uses the `in` operator to check membership within a tuple.

```
the_3_musketeers = ('Athos', 'Porthos', 'Aramis')
```

```
a_candidate = "D'Artagnan"
```

```
if a_candidate in the_3_musketeers:
    print("{} is a musketeer".format(a_candidate))
```

```
else:  
    print("{} is not a musketeer".format(a_candidate))
```

D'Artagnan is not a musketeer

```
if a_candidate in the_3_musketeers:  
    print(f"{a_candidate} is a musketeer.")  
else:  
    print(f"{a_candidate} is not a musketeer.")
```

D'Artagnan is not a musketeer.

The elif statement

Python also provides an `elif` statement (short for "else if") to chain a series of conditional blocks. The conditions are evaluated one by one. For the first condition that evaluates to `True`, the block of statements below it is executed. The remaining conditions and statements are not evaluated. So, in an `if`, `elif`, `elif ...` chain, at most one block of statements is executed, the one corresponding to the first condition that evaluates to `True`.

```
today = 'Wednesday'
```

```
if today == 'Sunday':  
    print("Today is the day of the sun.")  
elif today == 'Monday':  
    print("Today is the day of the moon.")  
elif today == 'Tuesday':  
    print("Today is the day of Tyr, the god of war.")  
elif today == 'Wednesday':  
    print("Today is the day of Odin, the supreme diety.")  
elif today == 'Thursday':  
    print("Today is the day of Thor, the god of thunder.")  
elif today == 'Friday':  
    print("Today is the day of Frigga, the goddess of beauty.")  
elif today == 'Saturday':  
    print("Today is the day of Saturn, the god of fun and feasting.")
```

Today is the day of Odin, the supreme diety.

In the above example, the first 3 conditions evaluate to `False`, so none of the first 3 messages are printed. The fourth condition evaluates to `True`, so the corresponding message is printed. The remaining conditions are skipped. Try changing the value of `today` above and re-executing the cells to print all the different messages.

To verify that the remaining conditions are skipped, let us try another example.

```
a_number = 15
```

```
if a_number % 2 == 0:  
    print('{} is divisible by 2'.format(a_number))
```

```
elif a_number % 3 == 0:
    print('{} is divisible by 3'.format(a_number))
elif a_number % 5 == 0:
    print('{} is divisible by 5'.format(a_number))
elif a_number % 7 == 0:
    print('{} is divisible by 7'.format(a_number))
```

15 is divisible by 3

Note that the message 15 is divisible by 5 is not printed because the condition `a_number % 5 == 0` isn't evaluated, since the previous condition `a_number % 3 == 0` evaluates to `True`. This is the key difference between using a chain of `if`, `elif`, `elif ...` statements vs. a chain of `if` statements, where each condition is evaluated independently.

```
if a_number % 2 == 0:
    print('{} is divisible by 2'.format(a_number))
if a_number % 3 == 0:
    print('{} is divisible by 3'.format(a_number))
if a_number % 5 == 0:
    print('{} is divisible by 5'.format(a_number))
if a_number % 7 == 0:
    print('{} is divisible by 7'.format(a_number))
```

15 is divisible by 3

15 is divisible by 5

Using `if`, `elif`, and `else` together

You can also include an `else` statement at the end of a chain of `if`, `elif ...` statements. This code within the `else` block is evaluated when none of the conditions hold true.

```
a_number = 49
```

```
if a_number % 2 == 0:
    print('{} is divisible by 2'.format(a_number))
elif a_number % 3 == 0:
    print('{} is divisible by 3'.format(a_number))
elif a_number % 5 == 0:
    print('{} is divisible by 5'.format(a_number))
else:
    print('All checks failed!')
    print('{} is not divisible by 2, 3 or 5'.format(a_number))
```

All checks failed!

49 is not divisible by 2, 3 or 5

Conditions can also be combined using the logical operators `and`, `or` and `not`. Logical operators are explained in detail in the [first tutorial](#).

```
a_number = 12
```

```
if a_number % 3 == 0 and a_number % 5 == 0:
    print("The number {} is divisible by 3 and 5".format(a_number))
elif not a_number % 5 == 0:
    print("The number {} is not divisible by 5".format(a_number))
```

The number 12 is not divisible by 5

Non-Boolean Conditions

Note that conditions do not necessarily have to be booleans. In fact, a condition can be any value. The value is converted into a boolean automatically using the `bool` operator. This means that falsy values like `0`, `''`, `{}`, `[]`, etc. evaluate to `False` and all other values evaluate to `True`.

```
if '':
    print('The condition evaluated to True')
else:
    print('The condition evaluated to False')
```

The condition evaluated to False

```
if 'Hello':
    print('The condition evaluated to True')
else:
    print('The condition evaluated to False')
```

The condition evaluated to True

```
if { 'a': 34 }:
    print('The condition evaluated to True')
else:
    print('The condition evaluated to False')
```

The condition evaluated to True

```
if None:
    print('The condition evaluated to True')
else:
    print('The condition evaluated to False')
```

The condition evaluated to False

Nested conditional statements

The code inside an `if` block can also include an `if` statement inside it. This pattern is called `nesting` and is used to check for another condition after a particular condition holds true.

```
a_number = 15
```

```

if a_number % 2 == 0:
    print("{} is even".format(a_number))
    if a_number % 3 == 0:
        print("{} is also divisible by 3".format(a_number))
    else:
        print("{} is not divisible by 3".format(a_number))
else:
    print("{} is odd".format(a_number))
    if a_number % 5 == 0:
        print("{} is also divisible by 5".format(a_number))
    else:
        print("{} is not divisible by 5".format(a_number))

```

15 is odd

15 is also divisible by 5

Notice how the `print` statements are indented by 8 spaces to indicate that they are part of the inner `if / else` blocks.

Nested `if`, `else` statements are often confusing to read and prone to human error. It's good to avoid nesting whenever possible, or limit the nesting to 1 or 2 levels.

Shorthand `if` conditional expression

A frequent use case of the `if` statement involves testing a condition and setting a variable's value based on the condition.

```

a_number = 13

if a_number % 2 == 0:
    parity = 'even'
else:
    parity = 'odd'

print('The number {} is {}'.format(a_number, parity))

```

The number 13 is odd.

Python provides a shorter syntax, which allows writing such conditions in a single line of code. It is known as a *conditional expression*, sometimes also referred to as a *ternary operator*. It has the following syntax:

```
x = true_value if condition else false_value
```

It has the same behavior as the following `if - else` block:

```

if condition:
    x = true_value
else:
    x = false_value

```

Let's try it out for the example above.

```
parity = 'even' if a_number % 2 == 0 else 'odd'
```

```
print('The number {} is {}'.format(a_number, parity))
```

The number 49 is odd.

Statements and Expressions

The conditional expression highlights an essential distinction between *statements* and *expressions* in Python.

Statements: A statement is an instruction that can be executed. Every line of code we have written so far is a statement e.g. assigning a variable, calling a function, conditional statements using `if`, `else`, and `elif`, loops using `for` and `while` etc.

Expressions: An expression is some code that evaluates to a value. Examples include values of different data types, arithmetic expressions, conditions, variables, function calls, conditional expressions, etc.

Most expressions can be executed as statements, but not all statements are expressions. For example, the regular `if` statement is not an expression since it does not evaluate to a value. It merely performs some branching in the code. Similarly, loops and function definitions are not expressions (we'll learn more about these in later sections).

As a rule of thumb, an expression is anything that can appear on the right side of the assignment operator `=`. You can use this as a test for checking whether something is an expression or not. You'll get a syntax error if you try to assign something that is not an expression.

```
# if statement
result = if a_number % 2 == 0:
    'even'
else:
    'odd'
```

File `"/tmp/ipykernel_38/26250527.py"`, line 2

```
result = if a_number % 2 == 0:
        ^
```

SyntaxError: invalid syntax

```
# if expression
result = 'even' if a_number % 2 == 0 else 'odd'
```

The pass statement

`if` statements cannot be empty, there must be at least one statement in every `if` and `elif` block. You can use the `pass` statement to do nothing and avoid getting an error.

```
a_number = 9
```

```
if a_number % 2 == 0:
elif a_number % 3 == 0:
    print('{} is divisible by 3 but not divisible by 2')
```

File "/tmp/ipykernel_38/4157426157.py", line 2

```
elif a_number % 3 == 0:
```

^

IndentationError: expected an indented block

```
if a_number % 2 == 0:
    pass
elif a_number % 3 == 0:
    print('{} is divisible by 3 but not divisible by 2'.format(a_number))
```

9 is divisible by 3 but not divisible by 2

Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. [Jovian](#) offers an easy way of saving and sharing your Jupyter notebooks online.

```
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit(project='python-branching-and-loops')
```

```
[jovian] Updating notebook "evanmarie/python-branching-and-loops" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-branching-and-loops
'https://jovian.ai/evanmarie/python-branching-and-loops'
```

The first time you run `jovian.commit`, you may be asked to provide an API Key to securely upload the notebook to your Jovian account. You can get the API key from your [Jovian profile page](#) after logging in / signing up.

`jovian.commit` uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work.

Iteration with while loops

Another powerful feature of programming languages, closely related to branching, is running one or more statements multiple times. This feature is often referred to as *iteration* or *looping*, and there are two ways to do this in Python: using `while` loops and `for` loops.

`while` loops have the following syntax:

```
while condition:
    statement(s)
```

Statements in the code block under `while` are executed repeatedly as long as the `condition` evaluates to `True`. Generally, one of the statements under `while` makes some change to a variable that causes the condition to evaluate to `False` after a certain number of iterations.

Let's try to calculate the factorial of 100 using a `while` loop. The factorial of a number `n` is the product (multiplication) of all the numbers from 1 to `n`, i.e., $1*2*3*\dots*(n-2)*(n-1)*n$.

```
result = 1
i = 1

while i <= 100:
    result = result * i
    i = i+1

print('The factorial of 100 is: {}'.format(result))
```

The factorial of 100 is:

9332621544394415268169923885626670049071596826438162146859296389521759999322991560894146

Here's how the above code works:

- We initialize two variables, `result` and, `i`. `result` will contain the final outcome. And `i` is used to keep track of the next number to be multiplied with `result`. Both are initialized to 1 (can you explain why?)
- The condition `i <= 100` holds true (since `i` is initially 1), so the `while` block is executed.
- The `result` is updated to `result * i`, `i` is increased by 1 and it now has the value 2.
- At this point, the condition `i <= 100` is evaluated again. Since it continues to hold true, `result` is again updated to `result * i`, and `i` is increased to 3.
- This process is repeated till the condition becomes false, which happens when `i` holds the value 101. Once the condition evaluates to `False`, the execution of the loop ends, and the `print` statement below it is executed.

Can you see why `result` contains the value of the factorial of 100 at the end? If not, try adding `print` statements inside the `while` block to print `result` and `i` in each iteration.

Iteration is a powerful technique because it gives computers a massive advantage over human beings in performing thousands or even millions of repetitive operations really fast. With just 4-5 lines of code, we were able to multiply 100 numbers almost instantly. The same code can be used to multiply a thousand numbers (just change the condition to `i <= 1000`) in a few seconds.

You can check how long a cell takes to execute by adding the *magic* command `%%time` at the top of a cell. Try checking how long it takes to compute the factorial of 100, 1000, 10000, 100000, etc.

```
%%time

result = 1
i = 1

while i <= 1000:
```



```
****
*****
*****
*****
****
***
**
*
```

Here's another one, putting the two together:

```
 *
 ***
*****
*****
*****
*****
*****
*****
*****
***
 *
```

Infinite Loops

Suppose the condition in a `while` loop always holds true. In that case, Python repeatedly executes the code within the loop forever, and the execution of the code never completes. This situation is called an infinite loop. It generally indicates that you've made a mistake in your code. For example, you may have provided the wrong condition or forgotten to update a variable within the loop, eventually falsifying the condition.

If your code is *stuck* in an infinite loop during execution, just press the "Stop" button on the toolbar (next to "Run") or select "Kernel > Interrupt" from the menu bar. This will *interrupt* the execution of the code. The following two cells both lead to infinite loops and need to be interrupted.

```
# INFINITE LOOP - INTERRUPT THIS CELL

result = 1
i = 1

while i <= 100:
    result = result * i
    # forgot to increment i
```

KeyboardInterrupt

Traceback (most recent call last)

```
<ipython-input-41-5234d8c241fc> in <module>
      5
      6 while i <= 100:
----> 7     result = result * i
      8     # forgot to increment i
```

KeyboardInterrupt:

```
# INFINITE LOOP - INTERRUPT THIS CELL

result = 1
i = 1

while i > 0 : # wrong condition
    result *= i
    i += 1
```

KeyboardInterrupt Traceback (most recent call last)

```
<python-input-42-c4abf72fce4d> in <module>
      5
      6 while i > 0 : # wrong condition
----> 7     result *= i
      8     i += 1
```

KeyboardInterrupt:

break and continue statements

You can use the `break` statement within the loop's body to immediately stop the execution and *break* out of the loop (even if the condition provided to `while` still holds true).

```
i = 1
result = 1

while i <= 100:
    result *= i
    if i == 42:
        print('Magic number 42 reached! Stopping execution..')
        break
    i += 1

print('i:', i)
print('result:', result)
```

Magic number 42 reached! Stopping execution..

i: 42

result: 140500611775287989854314260624451156993638400000000

As you can see above, the value of `i` at the end of execution is 42. This example also shows how you can use an `if` statement within a `while` loop.

Sometimes you may not want to end the loop entirely, but simply skip the remaining statements in the loop and *continue* to the next loop. You can do this using the `continue` statement.

```
i = 1
result = 1

while i < 20:
    i += 1
    if i % 2 == 0:
        print('Skipping {}'.format(i))
        continue
    print('Multiplying with {}'.format(i))
    result = result * i

print('i:', i)
print('result:', result)
```

```
Skipping 2
Multiplying with 3
Skipping 4
Multiplying with 5
Skipping 6
Multiplying with 7
Skipping 8
Multiplying with 9
Skipping 10
Multiplying with 11
Skipping 12
Multiplying with 13
Skipping 14
Multiplying with 15
Skipping 16
Multiplying with 17
Skipping 18
Multiplying with 19
Skipping 20
i: 20
result: 654729075
```

In the example above, the statement `result = result * i` inside the loop is skipped when `i` is even, as indicated by the messages printed during execution.

Logging: The process of adding `print` statements at different points in the code (often within loops and conditional statements) for inspecting the values of variables at various stages of execution is called logging. As our programs get larger, they naturally become prone to human errors. Logging can help in verifying the program is working as expected. In many cases, `print` statements are added while writing & testing some code and are removed later.

Let us record a snapshot of our work before continuing using `jovian.commit` .

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

```
[jovian] Updating notebook "aakashns/python-branching-and-loops" on https://jovian.ai/
```

```
[jovian] Uploading notebook..
```

```
[jovian] Capturing environment..
```

```
[jovian] Committed successfully! https://jovian.ai/aakashns/python-branching-and-loops
```

```
'https://jovian.ai/aakashns/python-branching-and-loops'
```

Iteration with for loops

A `for` loop is used for iterating or looping over sequences, i.e., lists, tuples, dictionaries, strings, and *ranges*. For loops have the following syntax:

```
for value in sequence:  
    statement(s)
```

The statements within the loop are executed once for each element in `sequence`. Here's an example that prints all the element of a list.

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']  
  
for day in days:  
    print(day)
```

Monday

Tuesday

Wednesday

Thursday

Friday

Let's try using `for` loops with some other data types.

```
# Looping over a string  
for char in 'Monday':  
    print(char)
```

M

o

n

d

a

y

```
# Looping over a tuple  
for fruit in ('Apple', 'Banana', 'Guava'):  
    print("Here's a fruit:", fruit)
```

Here's a fruit: Apple

Here's a fruit: Banana

Here's a fruit: Guava

```
# Looping over a dictionary
person = {
    'name': 'John Doe',
    'sex': 'Male',
    'age': 32,
    'married': True
}

for key in person:
    print("Key:", key, ", ", "Value:", person[key])
```

Key: name , Value: John Doe

Key: sex , Value: Male

Key: age , Value: 32

Key: married , Value: True

Note that while using a dictionary with a `for` loop, the iteration happens over the dictionary's keys. The key can be used within the loop to access the value. You can also iterate directly over the values using the `.values` method or over key-value pairs using the `.items` method.

```
for value in person.values():
    print(value)
```

John Doe

Male

32

True

```
for key_value_pair in person.items():
    print(key_value_pair)
```

('name', 'John Doe')

('sex', 'Male')

('age', 32)

('married', True)

Since a key-value pair is a tuple, we can also extract the key & value into separate variables.

```
for key, value in person.items():
    print("Key:", key, ", ", "Value:", value)
```

Key: name , Value: John Doe

Key: sex , Value: Male

Key: age , Value: 32

Key: married , Value: True

Iterating using range and enumerate

The `range` function is used to create a sequence of numbers that can be iterated over using a `for` loop. It can be used in 3 ways:

- `range(n)` - Creates a sequence of numbers from 0 to n-1
- `range(a, b)` - Creates a sequence of numbers from a to b-1
- `range(a, b, step)` - Creates a sequence of numbers from a to b-1 with increments of step

Let's try it out.

```
for i in range(7):  
    print(i)
```

0
1
2
3
4
5
6

```
for i in range(3, 10):  
    print(i)
```

3
4
5
6
7
8
9

```
for i in range(3, 14, 4):  
    print(i)
```

3
7
11

Ranges are used for iterating over lists when you need to track the index of elements while iterating.

```
a_list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

```
for i in range(len(a_list)):
    print('The value at position {} is {}'.format(i, a_list[i]))
```

The value at position 0 is Monday.

The value at position 1 is Tuesday.

The value at position 2 is Wednesday.

The value at position 3 is Thursday.

The value at position 4 is Friday.

Another way to achieve the same result is by using the `enumerate` function with `a_list` as an input, which returns a tuple containing the index and the corresponding element.

```
for i, val in enumerate(a_list):
    print('The value at position {} is {}'.format(i, val))
```

The value at position 0 is Monday.

The value at position 1 is Tuesday.

The value at position 2 is Wednesday.

The value at position 3 is Thursday.

The value at position 4 is Friday.

break, continue and pass statements

Similar to while loops, for loops also support the `break` and `continue` statements. `break` is used for breaking out of the loop and `continue` is used for skipping ahead to the next iteration.

```
weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

```
for day in weekdays:
    print('Today is {}'.format(day))
    if (day == 'Wednesday'):
        print("I don't work beyond Wednesday!")
        break
```

Today is Monday

Today is Tuesday

Today is Wednesday

I don't work beyond Wednesday!

```
for day in weekdays:
    if (day == 'Wednesday'):
        print("I don't work on Wednesday!")
        continue
    print('Today is {}'.format(day))
```

Today is Monday

Today is Tuesday

I don't work on Wednesday!

Today is Thursday

Today is Friday

Like if statements, for loops cannot be empty, so you can use a `pass` statement if you don't want to execute any statements inside the loop.

```
for day in weekdays:  
    pass
```

Nested for and while loops

Similar to conditional statements, loops can be nested inside other loops. This is useful for looping lists of lists, dictionaries etc.

```
persons = [{'name': 'John', 'sex': 'Male'}, {'name': 'Jane', 'sex': 'Female'}]  
  
for person in persons:  
    for key in person:  
        print(key, ":", person[key])  
    print(" ")
```

name : John

sex : Male

name : Jane

sex : Female

```
days = ['Monday', 'Tuesday', 'Wednesday']  
fruits = ['apple', 'banana', 'guava']  
  
for day in days:  
    for fruit in fruits:  
        print(day, fruit)
```

Monday apple

Monday banana

Monday guava

Tuesday apple

Tuesday banana

Tuesday guava

Wednesday apple

Wednesday banana

Wednesday guava

With this, we conclude our discussion of branching and loops in Python.

Further Reading and References

We've covered a lot of ground in just 3 tutorials.

Following are some resources to learn about more about conditional statements and loops in Python:

- Python Tutorial at W3Schools: <https://www.w3schools.com/python/>
- Practical Python Programming: <https://dabeaz-course.github.io/practical-python/Notes/Contents.html>
- Python official documentation: <https://docs.python.org/3/tutorial/index.html>

You are now ready to move on to the next tutorial: [Writing Reusable Code Using Functions in Python](#)

Let's save a snapshot of our notebook one final time using `jovian.commit` .

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

Questions for Revision

Try answering the following questions to test your understanding of the topics covered in this notebook:

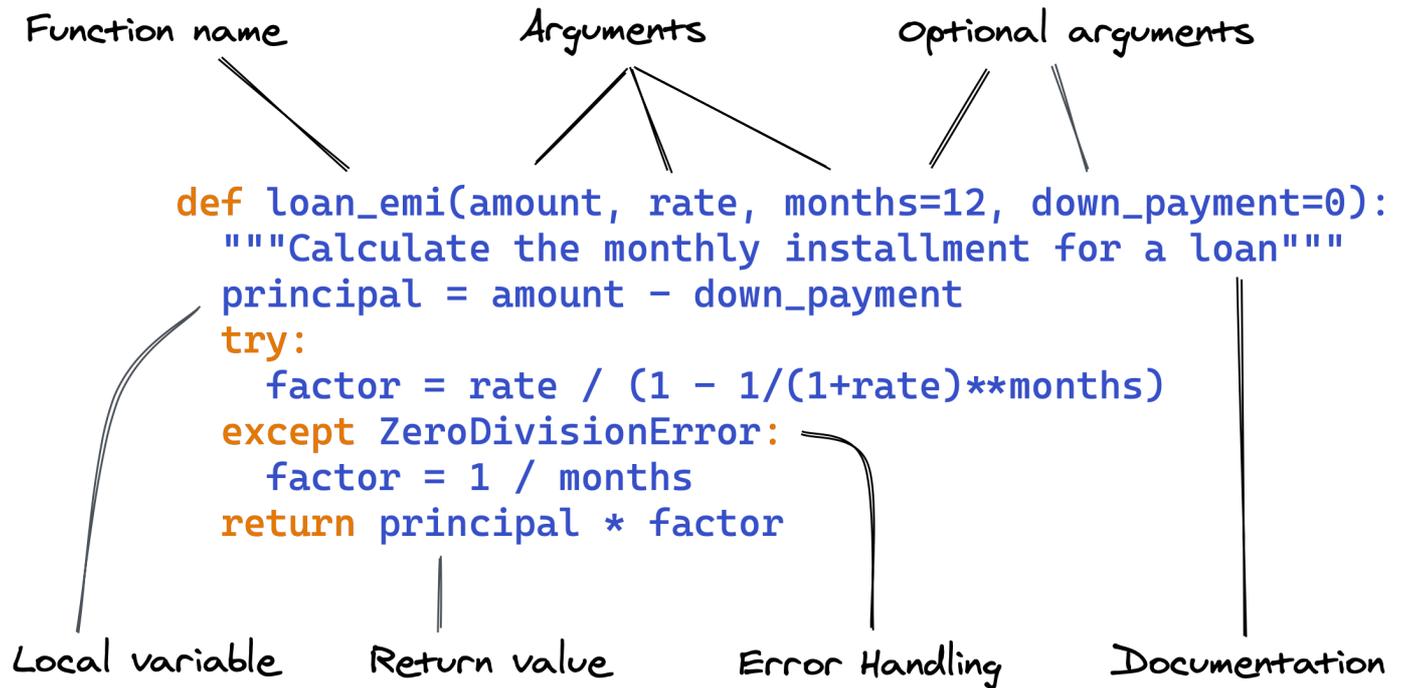
1. What is branching in programming languages?
2. What is the purpose of the `if` statement in Python?
3. What is the syntax of the `if` statement? Give an example.
4. What is indentation? Why is it used?
5. What is an indented block of statements?
6. How do you perform indentation in Python?
7. What happens if some code is not indented correctly?
8. What happens when the condition within the `if` statement evaluates to `True`? What happens if the condition evaluates for `false`?
9. How do you check if a number is even?
10. What is the purpose of the `else` statement in Python?
11. What is the syntax of the `else` statement? Give an example.
12. Write a program that prints different messages based on whether a number is positive or negative.
13. Can the `else` statement be used without an `if` statement?
14. What is the purpose of the `elif` statement in Python?
15. What is the syntax of the `elif` statement? Give an example.
16. Write a program that prints different messages for different months of the year.
17. Write a program that uses `if`, `elif`, and `else` statements together.
18. Can the `elif` statement be used without an `if` statement?
19. Can the `elif` statement be used without an `else` statement?

20. What is the difference between a chain of `if`, `elif`, `elif...` statements and a chain of `if`, `if`, `if...` statements? Give an example.
21. Can non-boolean conditions be used with `if` statements? Give some examples.
22. What are nested conditional statements? How are they useful?
23. Give an example of nested conditional statements.
24. Why is it advisable to avoid nested conditional statements?
25. What is the shorthand `if` conditional expression?
26. What is the syntax of the shorthand `if` conditional expression? Give an example.
27. What is the difference between the shorthand `if` expression and the regular `if` statement?
28. What is a statement in Python?
29. What is an expression in Python?
30. What is the difference between statements and expressions?
31. Is every statement an expression? Give an example or counterexample.
32. Is every expression a statement? Give an example or counterexample.
33. What is the purpose of the `pass` statement in `if` blocks?
34. What is iteration or looping in programming languages? Why is it useful?
35. What are the two ways for performing iteration in Python?
36. What is the purpose of the `while` statement in Python?
37. What is the syntax of the `while` statement in Python? Give an example.
38. Write a program to compute the sum of the numbers 1 to 100 using a `while` loop.
39. Repeat the above program for numbers up to 1000, 10000, and 100000. How long does it take each loop to complete?
40. What is an infinite loop?
41. What causes a program to enter an infinite loop?
42. How do you interrupt an infinite loop within Jupyter?
43. What is the purpose of the `break` statement in Python?
44. Give an example of using a `break` statement within a `while` loop.
45. What is the purpose of the `continue` statement in Python?
46. Give an example of using the `continue` statement within a `while` loop.
47. What is logging? How is it useful?
48. What is the purpose of the `for` statement in Python?
49. What is the syntax of `for` loops? Give an example.
50. How are `for` loops and `while` loops different?
51. How do you loop over a string? Give an example.
52. How do you loop over a list? Give an example.
53. How do you loop over a tuple? Give an example.
54. How do you loop over a dictionary? Give an example.

55. What is the purpose of the range statement? Give an example.
56. What is the purpose of the enumerate statement? Give an example.
57. How are the break, continue, and pass statements used in for loops? Give examples.
58. Can loops be nested within other loops? How is nesting useful?
59. Give an example of a for loop nested within another for loop.
60. Give an example of a while loop nested within another while loop.
61. Give an example of a for loop nested within a while loop.
62. Give an example of a while loop nested within a for loop.

Writing Reusable Code using Functions in Python

This tutorial is a part of [Data Analysis with Python: Zero to Pandas](#) and [Zero to Data Analyst Science Bootcamp](#).



These tutorials take a practical and coding-focused approach. The best way to learn the material is to execute the code and experiment with it yourself.

This tutorial covers the following topics:

- Creating and using functions in Python
- Local variables, return values, and optional arguments
- Reusing functions and using Python library functions
- Exception handling using try-except blocks
- Documenting functions using docstrings

How to run the code

This tutorial is an executable [Jupyter notebook](#) hosted on [Jovian](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Jupyter Notebooks: This tutorial is a [Jupyter notebook](#) - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc., instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Creating and using functions

A function is a reusable set of instructions that takes one or more inputs, performs some operations, and often returns an output. Python contains many in-built functions like `print`, `len`, etc., and provides the ability to define new ones.

```
today = "Saturday"
print("Today is", today)
```

Today is Saturday

You can define a new function using the `def` keyword.

```
def say_hello():
    print('Hello there!')
    print('How are you?')
```

Note the round brackets or parentheses `()` and colon `:` after the function's name. Both are essential parts of the syntax. The function's *body* contains an indented block of statements. The statements inside a function's body are not executed when the function is defined. To execute the statements, we need to *call* or *invoke* the function.

```
say_hello()
```

Hello there!

How are you?

```
def dance_time_warp():
    print('👉 Jump to the left!👈')
    print('👉 Step to the riiiiiiiiight! 👈')
    print('👉 Put your hands on your hips! 👈')
    print('👉 Pull your knees in tiiiiiiiiight! 👈')
```

```
dance_time_warp()
```

👉 Jump to the left!👈

👉 Step to the riiiiiiiiight! 👈

👉 Put your hands on your hips! 👈

👉 Pull your knees in tiiiiiiiiight! 👈

Function arguments

Functions can accept zero or more values as *inputs* (also known as *arguments* or *parameters*). Arguments help us write flexible functions that can perform the same operations on different values. Further, functions can return a result that can be stored in a variable or used in other expressions.

Here's a function that filters out the even numbers from a list and returns a new list using the `return` keyword.

```
def filter_even(number_list):
    result_list = []
    for number in number_list:
        if number % 2 == 0:
            result_list.append(number)
    return result_list
```

Can you understand what the function does by looking at the code? If not, try executing each line of the function's body separately within a code cell with an actual list of numbers in place of `number_list`.

```
even_list = filter_even([1, 2, 3, 4, 5, 6, 7])
```

```
even_list
```

```
[2, 4, 6]
```

```
even_list_02 = filter_even([1, 23, 5, 6, 23, 38, 578, 3294, 174, 847, 9947, 9479])
even_list_02
```

```
[6, 38, 578, 3294, 174]
```

Writing great functions in Python

As a programmer, you will spend most of your time writing and using functions. Python offers many features to make your functions powerful and flexible. Let's explore some of these by solving a problem:

Radha is planning to buy a house that costs \$1,260,000. She is considering two options to finance her purchase:

- Option 1: Make an immediate down payment of \$300,000, and take an 8-year loan with an interest rate of 10% (compounded monthly) for the remaining amount.
- Option 2: Take a 10-year loan with an interest rate of 8% (compounded monthly) for the entire amount.

Both these loans have to be paid back in equal monthly installments (EMIs). Which loan has a lower EMI among the two?

Since we need to compare the EMIs for two loan options, defining a function to calculate the EMI for a loan would be a great idea. The inputs to the function would be cost of the house, the down payment, duration of the loan, rate of interest etc. We'll build this function step by step.

First, let's write a simple function that calculates the EMI on the entire cost of the house, assuming that the loan must be paid back in one year, and there is no interest or down payment.

```
def loan_emi(amount):  
    emi = amount / 12  
    print('The EMI is ${}'.format(emi))
```

```
loan_emi(1260000)
```

The EMI is \$105000.0

Local variables and scope

Let's add a second argument to account for the duration of the loan in months.

```
def loan_emi(amount, duration):  
    emi = amount / duration  
    print('The EMI is ${}'.format(emi))
```

Note that the variable `emi` defined inside the function is not accessible outside. The same is true for the parameters `amount` and `duration`. These are all *local variables* that lie within the *scope* of the function.

Scope: Scope refers to the region within the code where a particular variable is visible. Every function (or class definition) defines a scope within Python. Variables defined in this scope are called *local variables*. Variables that are available everywhere are called *global variables*. Scope rules allow you to use the same variable names in different functions without sharing values from one to the other.

```
emi
```

```
-----  
NameError                                Traceback (most recent call last)  
/tmp/ipykernel_39/1358411353.py in <module>  
----> 1 emi
```

NameError: name 'emi' is not defined

```
amount
```

```
-----  
NameError                                Traceback (most recent call last)  
/tmp/ipykernel_39/628052274.py in <module>  
----> 1 amount
```

NameError: name 'amount' is not defined

```
duration
```

```
-----  
NameError                                Traceback (most recent call last)  
/tmp/ipykernel_39/3792379043.py in <module>  
----> 1 duration
```

NameError: name 'duration' is not defined

We can now compare a 6-year loan vs. a 10-year loan (assuming no down payment or interest).

```
loan_emi(1260000, 8*12)
```

The EMI is \$13125.0

```
loan_emi(1260000, 10*12)
```

The EMI is \$10500.0

Return values

As you might expect, the EMI for the 6-year loan is higher compared to the 10-year loan. Right now, we're printing out the result. It would be better to return it and store the results in variables for easier comparison. We can do this using the `return` statement

```
def loan_emi(amount, duration):  
    emi = amount / duration  
    return emi
```

```
emi1 = loan_emi(1260000, 8*12)
```

```
emi2 = loan_emi(1260000, 10*12)
```

```
emi1
```

13125.0

```
emi2
```

10500.0

```
emi1-emi2
```

2625.0

Optional arguments

Next, let's add another argument to account for the immediate down payment. We'll make this an *optional argument* with a default value of 0.

```
def loan_emi(amount, duration, down_payment=0):  
    loan_amount = amount - down_payment  
    emi = loan_amount / duration  
    return emi
```

```
emi1 = loan_emi(1260000, 8*12, 3e5)
```

```
emi1
```

```
10000.0
```

```
emi2 = loan_emi(1260000, 10*12)
```

```
emi2
```

```
10500.0
```

Next, let's add the interest calculation into the function. Here's the formula used to calculate the EMI for a loan:

$$EMI = \frac{P \times r \times (1+r)^n}{(1+r)^n - 1}$$

where:

- P is the loan amount (principal)
- n is the no. of months
- r is the rate of interest per month

The derivation of this formula is beyond the scope of this tutorial. See this video for an explanation:

<https://youtu.be/Coxza9ugW4E>.

```
def loan_emi(amount, duration, rate, down_payment=0):  
    loan_amount = amount - down_payment  
    emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)  
    return emi
```

Note that while defining the function, required arguments like `cost`, `duration` and `rate` must appear before optional arguments like `down_payment`.

Let's calculate the EMI for Option 1

```
loan_emi(1260000, 8*12, 0.1/12, 3e5)
```

```
14567.19753389219
```

While calculating the EMI for Option 2, we need not include the `down_payment` argument.

```
loan_emi(1260000, 10*12, 0.08/12)
```

```
15287.276888775077
```

```
print("8 years: ", loan_emi(1260000, 8 * 12, 0.1 / 12, 3e5))

print('10 years: ', loan_emi(1260000, 10 * 12, 0.08 / 12))
```

```
8 years: 14567.19753389219
10 years: 15287.276888775077
```

Named arguments

Invoking a function with many arguments can often get confusing and is prone to human errors. Python provides the option of invoking functions with *named* arguments for better clarity. You can also split function invocation into multiple lines.

```
emi1 = loan_emi(
    amount=1260000,
    duration=8*12,
    rate=0.1/12,
    down_payment=3e5
)
```

```
emi1
```

```
14567.19753389219
```

```
emi2 = loan_emi(amount=1260000, duration=10*12, rate=0.08/12)
```

```
emi2
```

```
15287.276888775077
```

```
print('EMI_1: ', emi1)
print('EMI_2: ', emi2)
```

```
EMI_1: 14567.19753389219
EMI_2: 15287.276888775077
```

```
eight_year = loan_emi(1260000, 8 * 12, 0.1 / 12, 3e5)
print(f'Eight year: ${eight_year:,.2f}')

ten_year = loan_emi(1260000, 10 * 12, 0.08 / 12)
print(f'Ten year: ${ten_year:,.2f}')

print('EMI_1: ', emi1)
print('EMI_2: ', emi2)
```

```
print(f'EMI_1: ${emi1:,.2f}')
print(f'EMI_2: ${emi2:,.2f}')
```

Eight year: \$14,567.20

Ten year: \$15,287.28

EMI_1: 14567.19753389219

EMI_2: 15287.276888775077

EMI_1: \$14,567.20

EMI_2: \$15,287.28

```
import math

print(math.ceil(1.2))

print('EMI_1: $', math.ceil(emi1))
print('EMI_2: $', math.ceil(emi2))

emi1 = math.ceil(emi1)
emi2 = math.ceil(emi2)

if emi1 < emi2:
    print('Option 1 is the lower EMI: ${:,}'.format(emi1))
else:
    print('Option 2 is the lower EMI: ${:,}'.format(emi2))
```

2

EMI_1: \$ 14568

EMI_2: \$ 15288

Option 1 is the lower EMI: \$14,568

Modules and library functions

We can already see that the EMI for Option 1 is lower than the EMI for Option 2. However, it would be nice to round up the amount to full dollars, rather than showing digits after the decimal. To achieve this, we might want to write a function that can take a number and round it up to the next integer (e.g., 1.2 is rounded up to 2). That would be a great exercise to try out!

However, since rounding numbers is a fairly common operation, Python provides a function for it (along with thousands of other functions) as part of the [Python Standard Library](#). Functions are organized into *modules* that need to be imported to use the functions they contain.

Modules: Modules are files containing Python code (variables, functions, classes, etc.). They provide a way of organizing the code for large Python projects into files and folders. The key benefit of using modules is *namespaces*: you must import the module to use its functions within a Python script or notebook. Namespaces provide encapsulation and avoid naming conflicts between your code and a module or across modules.

We can use the `ceil` function (short for *ceiling*) from the `math` module to round up numbers. Let's import the module and use it to round up the number `1.2`.

```
import math
```

```
help(math.ceil)
```

Help on built-in function ceil in module math:

```
ceil(x, /)
```

Return the ceiling of x as an Integral.

This is the smallest integer $\geq x$.

```
math.ceil(1.2)
```

2

Let's now use the `math.ceil` function within the `home_loan_emi` function to round up the EMI amount.

Using functions to build other functions is a great way to reuse code and implement complex business logic while still keeping the code small, understandable, and manageable. Ideally, a function should do one thing and one thing only. If you find yourself writing a function that does too many things, consider splitting it into multiple smaller, independent functions. As a rule of thumb, try to limit your functions to 10 lines of code or less. Good programmers always write short, simple, and readable functions.

```
def loan_emi(amount, duration, rate, down_payment=0):  
    loan_amount = amount - down_payment  
    emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)  
    emi = math.ceil(emi)  
    return emi
```

```
emi1 = loan_emi(  
    amount=1260000,  
    duration=8*12,  
    rate=0.1/12,  
    down_payment=3e5  
)
```

```
emi1
```

14568

```
emi2 = loan_emi(amount=1260000, duration=10*12, rate=0.08/12)
```

```
emi2
```

Let's compare the EMIs and display a message for the option with the lower EMI.

```
if emi1 < emi2:
    print("Option 1 has the lower EMI: {}".format(emi1))
else:
    print("Option 2 has the lower EMI: {}".format(emi2))
```

Option 1 has the lower EMI: \$14568

```
print("EMI for HOUSE: ${:,}".format(math.ceil(emi_house)))
```

```
print("EMI for CAR: ${:,}".format(math.ceil(emi_car)))
```

```
emi_with_interest = loan_emi(amount=100000, duration=10 * 12, rate=0.09 / 12)
print("EMI for $100,000 loan with interest: ${:,}".format(math.ceil(emi_with_interest)))

try:
    emi_without_interest = loan_emi(amount=100000, duration=10 * 12, rate=0. / 12)
except ZeroDivisionError:
    print('Division by zero is not allowed.')
```

EMI for \$100,000 loan with interest: \$1,267

Division by zero is not allowed.

Reusing and improving functions

Now we know for sure that "Option 1" has the lower EMI among the two options. But what's even better is that we now have a handy function `loan_emi` that we can use to solve many other similar problems with just a few lines of code. Let's try it with a couple more questions.

Q: Shaun is currently paying back a home loan for a house he bought a few years ago. The cost of the house was \$800,000. Shaun made a down payment of 25% of the price. He financed the remaining amount using a 6-year loan with an interest rate of 7% per annum (compounded monthly). Shaun is now buying a car worth \$60,000, which he is planning to finance using a 1-year loan with an interest rate of 12% per annum. Both loans are paid back in EMIs. What is the total monthly payment Shaun makes towards loan repayment?

This question is now straightforward to solve, using the `loan_emi` function we've already defined.

```
cost_of_house = 800000
home_loan_duration = 6*12 # months
home_loan_rate = 0.07/12 # monthly
home_down_payment = .25 * 800000

emi_house = loan_emi(amount=cost_of_house,
```

```
duration=home_loan_duration,  
rate=home_loan_rate,  
down_payment=home_down_payment)
```

```
emi_house
```

10230

```
cost_of_car = 60000  
car_loan_duration = 1*12 # months  
car_loan_rate = .12/12 # monthly  
  
emi_car = loan_emi(amount=cost_of_car,  
                  duration=car_loan_duration,  
                  rate=car_loan_rate)
```

```
emi_car
```

5331

```
print("Shaun makes a total monthly payment of ${} towards loan repayments.".format(emi_
```

Shaun makes a total monthly payment of \$15561 towards loan repayments.

Exceptions and try-except

Q: If you borrow \$100,000 using a 10-year loan with an interest rate of 9% per annum, what is the total amount you end up paying as interest?

One way to solve this problem is to compare the EMIs for two loans: one with the given rate of interest and another with a 0% rate of interest. The total interest paid is then simply the sum of monthly differences over the duration of the loan.

```
emi_with_interest = loan_emi(amount=100000, duration=10*12, rate=0.09/12)  
emi_with_interest
```

1267

```
emi_without_interest = loan_emi(amount=100000, duration=10*12, rate=0./12)  
emi_without_interest
```

```
-----  
ZeroDivisionError                                Traceback (most recent call last)  
<ipython-input-45-b684ffbee02d> in <module>  
----> 1 emi_without_interest = loan_emi(amount=100000, duration=10*12, rate=0./12)  
      2 emi_without_interest  
  
<ipython-input-35-ad16168becb0> in loan_emi(amount, duration, rate, down_payment)  
      1 def loan_emi(amount, duration, rate, down_payment=0):  
      2     loan_amount = amount - down_payment
```

```

----> 3     emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
        4     emi = math.ceil(emi)
        5     return emi

```

ZeroDivisionError: float division by zero

Something seems to have gone wrong! If you look at the error message above carefully, Python tells us precisely what is wrong. Python *throws* a `ZeroDivisionError` with a message indicating that we're trying to divide a number by zero. `ZeroDivisionError` is an *exception* that stops further execution of the program.

Exception: Even if a statement or expression is syntactically correct, it may cause an error when the Python interpreter tries to execute it. Errors detected during execution are called exceptions. Exceptions typically stop further execution of the program unless handled within the program using `try - except` statements.

Python provides many built-in exceptions *thrown* when built-in operators, functions, or methods are used incorrectly: <https://docs.python.org/3/library/exceptions.html#built-in-exceptions>. You can also define your custom exception by extending the `Exception` class (more on that later).

You can use the `try` and `except` statements to *handle* an exception. Here's an example:

```

try:
    print("Now computing the result..")
    result = 5 / 0
    print("Computation was completed successfully")
except ZeroDivisionError:
    print("Failed to compute result because you were trying to divide by zero")
    result = None

print(result)

```

Now computing the result..

Failed to compute result because you were trying to divide by zero

None

```

def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    try:
        emi_100k = loan_amount * rate * ((1 + rate) ** duration) / (((1 + rate) ** dura
    except ZeroDivisionError:
        emi_100k = loan_amount / duration # Do not divide by the rate, because it went
    emi_100k = math.ceil(emi_100k)

    return emi_100k

print("EMI for $100,000 loan with interest (try-exception) ${:,} ".format
      (math.ceil(loan_emi(100000, 10 * 12, 0.09 / 12))))

print("EMI for $100,000 loan without interest (try-exception) ${:,} ".format

```

```

    (math.ceil(loan_emi(100000, 10 * 12, rate=0))))

total_interest = (loan_emi(100000, 10 * 12, 0.09 / 12) - (loan_emi(100000, 10 * 12, rate=0)))

print("Total interest paid: ${:,}".format(math.ceil(total_interest)))

```

EMI for \$100,000 loan with interest (try-except) \$1,267

EMI for \$100,000 loan without interest (try-except) \$834

Total interest paid: \$51,960

When an exception occurs inside a `try` block, the block's remaining statements are skipped. The `except` block is executed if the type of exception thrown matches that of the exception being handled. After executing the `except` block, the program execution returns to the normal flow.

You can also handle more than one type of exception using multiple `except` statements. Learn more about exceptions here: https://www.w3schools.com/python/python_try_except.asp.

Let's enhance the `loan_emi` function to use `try - except` to handle the scenario where the interest rate is 0%. It's common practice to make changes/enhancements to functions over time as new scenarios and use cases come up. It makes functions more robust & versatile.

```

def loan_emi(amount, duration, rate, down_payment=0):
    loan_amount = amount - down_payment
    try:
        emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    except ZeroDivisionError:
        emi = loan_amount / duration
    emi = math.ceil(emi)
    return emi

```

We can use the updated `loan_emi` function to solve our problem.

Q: If you borrow \$100,000 using a 10-year loan with an interest rate of 9% per annum, what is the total amount you end up paying as interest?

```

emi_with_interest = loan_emi(amount=100000, duration=10*12, rate=0.09/12)
emi_with_interest

```

1267

```

emi_without_interest = loan_emi(amount=100000, duration=10*12, rate=0)
emi_without_interest

```

834

```

total_interest = (emi_with_interest - emi_without_interest) * 10*12

```

```
print("The total interest paid is ${}.".format(total_interest))
```

The total interest paid is \$51960.

Documenting functions using Docstrings

We can add some documentation within our function using a *docstring*. A docstring is simply a string that appears as the first statement within the function body, and is used by the `help` function. A good docstring describes what the function does, and provides some explanation about the arguments.

```
def loan_emi(amount, duration, rate, down_payment=0):
    """Calculates the equal montly installment (EMI) for a loan.

    Arguments:
        amount - Total amount to be spent (loan + down payment)
        duration - Duration of the loan (in months)
        rate - Rate of interest (monthly)
        down_payment (optional) - Optional intial payment (deducted from amount)
    """
    loan_amount = amount - down_payment
    try:
        emi = loan_amount * rate * ((1+rate)**duration) / (((1+rate)**duration)-1)
    except ZeroDivisionError:
        emi = loan_amount / duration
    emi = math.ceil(emi)
    return emi
```

In the docstring above, we've provided some additional information that the `duration` and `rate` are measured in months. You might even consider naming the arguments `duration_months` and `rate_monthly`, to avoid any confusion whatsoever. Can you think of some other ways to improve the function?

```
help(loan_emi)
```

Help on function loan_emi in module __main__:

```
loan_emi(amount, duration, rate, down_payment=0)
    Calculates the equal montly installment (EMI) for a loan.
```

```
Arguments:
    amount - Total amount to be spent (loan + down payment)
    duration - Duration of the loan (in months)
    rate - Rate of interest (monthly)
    down_payment (optional) - Optional intial payment (deducted from amount)
```

Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. [Jovian](#) offers an easy way of saving and sharing your Jupyter notebooks online.

```
# Instal the library
!pip install jovian --upgrade --quiet
```

```
# Import the jovian module
import jovian
```

```
jovian.commit(project='python-functions-and-scope', environment=None)
```

```
[jovian] Attempting to save notebook..
[jovian] Updating notebook "aakashns/python-functions-and-scope" on https://jovian.ai/
[jovian] Uploading notebook..
[jovian] Committed successfully! https://jovian.ai/aakashns/python-functions-and-scope
'https://jovian.ai/aakashns/python-functions-and-scope'
```

The first time you run `jovian.commit`, you'll be asked to provide an API Key to securely upload the notebook to your Jovian account. You can get the API key from your [Jovian profile page](#) after logging in / signing up.

`jovian.commit` uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work.

Exercise - Data Analysis for Vacation Planning

You're planning a vacation, and you need to decide which city you want to visit. You have shortlisted four cities and identified the return flight cost, daily hotel cost, and weekly car rental cost. While renting a car, you need to pay for entire weeks, even if you return the car sooner.

City	Return Flight (\$)	Hotel per day (\$)	Weekly Car Rental (\$)
Paris	200	20	200
London	250	30	120
Dubai	370	15	80
Mumbai	450	10	70

Answer the following questions using the data above:

1. If you're planning a 1-week long trip, which city should you visit to spend the least amount of money?
2. How does the answer to the previous question change if you change the trip's duration to four days, ten days or two weeks?
3. If your total budget for the trip is \$1000, which city should you visit to maximize the duration of your trip? Which city should you visit if you want to minimize the duration?
4. How does the answer to the previous question change if your budget is \$600, \$2000, or \$1500?

Hint: To answer these questions, it will help to define a function `cost_of_trip` with relevant inputs like flight cost, hotel rate, car rental rate, and duration of the trip. You may find the `math.ceil` function useful for calculating the total cost of car rental.

BRUTE FORCING THIS THING!

(Optimized version is below brute force.)

```
cities = {
    'Paris': [200, 20 * 7, 200],
    'London': [250, 30 * 7, 120],
    'Dubai': [370, 15 * 7, 80],
    'Mumbai': [450, 10 * 7, 70]
}

def cheapest_week_trip_cost(city):
    week_trip = []

    for city in cities:
        cost = cities[city][0] + cities[city][1] + cities[city][2]
        week_trip.append((city, cost))
        print(week_trip)

    return min(week_trip, key=lambda x: x[1])

print("Cheapest city to visit for 1 week: ", cheapest_week_trip_cost(cities))
```

```
[('Paris', 540)]
[('Paris', 540), ('London', 580)]
[('Paris', 540), ('London', 580), ('Dubai', 555)]
[('Paris', 540), ('London', 580), ('Dubai', 555), ('Mumbai', 590)]
Cheapest city to visit for 1 week: ('Paris', 540)
```

```
cities2 = {
    'Paris': [200, 20, 200],
    'London': [250, 30, 120],
    'Dubai': [370, 15, 80],
    'Mumbai': [450, 10, 70]
}
```

```
def cheapest_trip_days(city, days):
    cities_cost = []
    for city, (flight, hotel_per_day, car_per_week) in cities2.items():
        cost = flight + hotel_per_day*days + car_per_week*math.ceil(days/7)
        cities_cost.append((city, cost))
    return min(cities_cost, key=lambda x: x[1])
```

```
print("Cheapest city to visit for 4 days: ", cheapest_trip_days(cities2, 4))
print("Cheapest city to visit for 10 days: ", cheapest_trip_days(cities2, 10))
print("Cheapest city to visit for 14 days: ", cheapest_trip_days(cities2, 14))
```

Cheapest city to visit for 4 days: ('Paris', 480)

Cheapest city to visit for 10 days: ('Dubai', 680)

Cheapest city to visit for 14 days: ('Mumbai', 730)

```
def longest_trip_possible(cities, dollars):

    max_city = 0
    best_city = ''

    for city, (flight, hotel_per_day, car_per_week) in cities2.items():
        time_in_city = max_time_city(flight, hotel_per_day, car_per_week, dollars)
        if time_in_city > max_city:
            best_city = city
            max_city = time_in_city

    return city, max_city

def shortest_trip_possible(cities, dollars):
    min_city = 100000
    min_city_name = ''

    for city, (flight, hotel_per_day, car_per_week) in cities2.items():
        time_in_city = max_time_city(flight, hotel_per_day, car_per_week, dollars)
        if time_in_city < min_city:
            min_city_name = city
            min_city = time_in_city

    return min_city_name, min_city

def max_time_city(flight, hotel_per_day, car_per_week, dollars):

    left_to_spend = dollars - flight
    day = 0

    while left_to_spend >= hotel_per_day:
        if day % 7 == 0:
            if left_to_spend >= car_per_week + hotel_per_day:
                left_to_spend -= car_per_week
            else:
                break

        left_to_spend -= hotel_per_day
        day += 1
```

```
# print(f"Day number:{day}, $ left: {left_to_spend}")  
  
return day
```

Maximum stay on \$600

```
longest_trip_possible(cities2, 600)
```

```
('Mumbai', 7)
```

Maximum stay on \$1000

```
longest_trip_possible(cities2, 1000)
```

```
('Mumbai', 27)
```

Maximum stay on \$1500

```
longest_trip_possible(cities2, 1500)
```

```
('Mumbai', 49)
```

Maximum stay on \$2000

```
longest_trip_possible(cities2, 2000)
```

```
('Mumbai', 77)
```

Minimum stay on \$600

```
shortest_trip_possible(cities, 600)
```

```
('Paris', 7)
```

Minimum stay on \$1000

```
shortest_trip_possible(cities, 1000)
```

```
('Paris', 14)
```

Minimum stay on \$1500

```
shortest_trip_possible(cities, 1500)
```

```
('Paris', 25)
```

Minimum stay on \$2000

```
shortest_trip_possible(cities, 2000)
```

```
('Paris', 35)
```

OPTIMIZE TIME!

```
cities2 = {  
    'Paris': [200, 20, 200],  
    'London': [250, 30, 120],  
    'Dubai': [370, 15, 80],  
    'Mumbai': [450, 10, 70]  
}
```

```
def longest_optimized(cities, dollars):  
  
    city_list = []  
    days = 0  
  
    for city, (flight, hotel_per_day, car_per_week) in cities2.items():  
  
        available = dollars - flight  
        weekly = (hotel_per_day * 7) + car_per_week  
        left_over = available % weekly  
        days = (available // weekly) * 7  
        while left_over >= car_per_week + hotel_per_day:  
            days += 1  
            left_over -= hotel_per_day  
  
        city_list.append([city, days])  
  
    return max(city_list, key=lambda x: x[1])
```

```
def shortest_optimized(cities, dollars):  
  
    city_list = []  
    days = 0  
  
    for city, (flight, hotel_per_day, car_per_week) in cities2.items():  
  
        available = dollars - flight  
        weekly = (hotel_per_day * 7) + car_per_week  
        left_over = available % weekly  
        days = (available // weekly) * 7
```

```
while left_over >= car_per_week + hotel_per_day:
    days += 1
    left_over -= hotel_per_day

city_list.append([city, days])

return min(city_list, key=lambda x: x[1])
```

Maximum stay on \$600 optimized

```
longest_optimized(cities2, 600)
```

```
['Paris', 7]
```

Maximum stay on \$1000 optimized

```
longest_optimized(cities2, 1000)
```

```
['Mumbai', 27]
```

Maximum stay on \$1500 optimized

```
longest_optimized(cities2, 1500)
```

```
['Mumbai', 49]
```

Maximum stay on \$2000 optimized

```
longest_optimized(cities2, 2000)
```

```
['Mumbai', 77]
```

Minimum stay on \$600 optimized

```
shortest_optimized(cities2, 600)
```

```
['Paris', 7]
```

Minimum stay on \$1000 optimized

```
shortest_optimized(cities2, 1000)
```

```
['Paris', 14]
```

Minimum stay on \$1500 optimized

```
shortest_optimized(cities2, 1500)
```

```
['Paris', 25]
```

Minimum stay on \$2000 optimized

```
shortest_optimized(cities2, 2000)
```

```
['Paris', 35]
```

Summary and Further Reading

With this, we complete our discussion of functions in Python. We've covered the following topics in this tutorial:

- Creating and using functions
- Functions with one or more arguments
- Local variables and scope
- Returning values using `return`
- Using default arguments to make a function flexible
- Using named arguments while invoking a function
- Importing modules and using library functions
- Reusing and improving functions to handle new use cases
- Handling exceptions with `try-except`
- Documenting functions using docstrings

This tutorial on functions in Python is by no means exhaustive. Here are a few more topics to learn about:

- Functions with an arbitrary number of arguments using (`*args` and `**kwargs`)
- Defining functions inside functions (and closures)
- A function that invokes itself (recursion)
- Functions that accept other functions as arguments or return other functions
- Functions that enhance other functions (decorators)

Following are some resources to learn about more functions in Python:

- Python Tutorial at W3Schools: <https://www.w3schools.com/python/>
- Practical Python Programming: <https://dabeaz-course.github.io/practical-python/Notes/Contents.html>
- Python official documentation: <https://docs.python.org/3/tutorial/index.html>

You are ready to move on to the next tutorial: "[Reading from and writing to files using Python](#)".

Questions for Revision

Try answering the following questions to test your understanding of the topics covered in this notebook:

1. What is a function?
2. What are the benefits of using functions?
3. What are some built-in functions in Python?
4. How do you define a function in Python? Give an example.

5. What is the body of a function?
6. When are the statements in the body of a function executed?
7. What is meant by calling or invoking a function? Give an example.
8. What are function arguments? How are they useful?
9. How do you store the result of a function in a variable?
10. What is the purpose of the return keyword in Python?
11. Can you return multiple values from a function?
12. Can a return statement be used inside an if block or a for loop?
13. Can the return keyword be used outside a function?
14. What is scope in a programming region?
15. How do you define a variable inside a function?
16. What are local & global variables?
17. Can you access the variables defined inside a function outside its body? Why or why not?
18. What do you mean by the statement "a function defines a scope within Python"?
19. Do for and while loops define a scope, like functions?
20. Do if-else blocks define a scope, like functions?
21. What are optional function arguments & default values? Give an example.
22. Why should the required arguments appear before the optional arguments in a function definition?
23. How do you invoke a function with named arguments? Illustrate with an example.
24. Can you split a function invocation into multiple lines?
25. Write a function that takes a number and rounds it up to the nearest integer.
26. What are modules in Python?
27. What is a Python library?
28. What is the Python Standard Library?
29. Where can you learn about the modules and functions available in the Python standard library?
30. How do you install a third-party library?
31. What is a module namespace? How is it useful?
32. What problems would you run into if Python modules did not provide namespaces?
33. How do you import a module?
34. How do you use a function from an imported module? Illustrate with an example.
35. Can you invoke a function inside the body of another function? Give an example.
36. What is the single responsibility principle, and how does it apply while writing functions?
37. What some characteristics of well-written functions?
38. Can you use if statements or while loops within a function? Illustrate with an example.
39. What are exceptions in Python? When do they occur?
40. How are exceptions different from syntax errors?
41. What are the different types of in-built exceptions in Python? Where can you learn about them?

42. How do you prevent the termination of a program due to an exception?
43. What is the purpose of the try-except statements in Python?
44. What is the syntax of the try-except statements? Give an example.
45. What happens if an exception occurs inside a try block?
46. How do you handle two different types of exceptions using except? Can you have multiple except blocks under a single try block?
47. How do you create an except block to handle any type of exception?
48. Illustrate the usage of try-except inside a function with an example.
49. What is a docstring? Why is it useful?
50. How do you display the docstring for a function?
51. What are *args and **kwargs? How are they useful? Give an example.
52. Can you define functions inside functions?
53. What is function closure in Python? How is it useful? Give an example.
54. What is recursion? Illustrate with an example.
55. Can functions accept other functions as arguments? Illustrate with an example.
56. Can functions return other functions as results? Illustrate with an example.
57. What are decorators? How are they useful?
58. Implement a function decorator which prints the arguments and result of wrapped functions.
59. What are some in-built decorators in Python?
60. What are some popular Python libraries?

Solution for Exercise

Exercise - Data Analysis for Vacation Planning

You're planning a vacation, and you need to decide which city you want to visit. You have shortlisted four cities and identified the return flight cost, daily hotel cost, and weekly car rental cost. While renting a car, you need to pay for entire weeks, even if you return the car sooner.

City	Return Flight (\$)	Hotel per day (\$)	Weekly Car Rental (\$)
Paris	200	20	200
London	250	30	120
Dubai	370	15	80
Mumbai	450	10	70

Answer the following questions using the data above:

1. If you're planning a 1-week long trip, which city should you visit to spend the least amount of money?
2. How does the answer to the previous question change if you change the trip's duration to four days, ten days or two weeks?

3. If your total budget for the trip is \$600, which city should you visit to maximize the duration of your trip? Which city should you visit if you want to minimize the duration?
4. How does the answer to the previous question change if your budget is \$1000, \$2000, or \$1500?

Hint: To answer these questions, it will help to define a function `cost_of_trip` with relevant inputs like flight cost, hotel rate, car rental rate, and duration of the trip. You may find the `math.ceil` function useful for calculating the total cost of car rental.

```
import math
```

```
Paris=[200,20,200,'Paris']  
London = [250,30,120,'London']  
Dubai = [370,15,80,'Dubai']  
Mumbai = [450,10,70,'Mumbai']  
Cities = [Paris,London,Dubai,Mumbai]
```

```
def cost_of_trip(flight,hotel_cost,car_rent,num_of_days=0):  
    return flight+(hotel_cost*num_of_days)+(car_rent*math.ceil(num_of_days/7))
```

```
def days_to_visit(days):  
    costs=[]  
    for city in Cities:  
        cost=cost_of_trip(city[0],city[1],city[2],days)  
        costs.append((cost,city[3]))  
    min_cost = min(costs)  
    return min_cost
```

1. If you're planning a 1-week long trip, which city should you visit to spend the least amount of money?

```
days_to_visit(7)
```

```
(540, 'Paris')
```

2. How does the answer to the previous question change if you change the trip's duration to four days, ten days or two weeks?

```
days_to_visit(4)
```

```
(480, 'Paris')
```

```
days_to_visit(10)
```

```
(680, 'Dubai')
```

```
days_to_visit(14)
```

```
(730, 'Mumbai')
```

3. If your total budget for the trip is \$600, which city should you visit to maximize the duration of your trip? Which city should you visit if you want to minimize the duration?

```
def given_budget(budget, less_days=False):
    days=1
    cost=0
    while cost<budget:
        #copy of city cost
        cost_before=cost
        try:
            #copy of costs dictionary, if exists
            costs_before=costs.copy()
        except:
            #if costs dictionary doesn't exist, create an empty dictionary
            costs_before={}
        costs={}
        for city in Cities:
            cost = cost_of_trip(city[0],city[1],city[2],days)
            costs[cost] = city[3]
        if less_days:
            cost=max(list(costs.keys()))
            ''' The while loop breaks only after cost>600 condition is met.
            when the condition is met, the costs dictionary updates to values that are
            so we check if it is exceeding, if it does, we return the values from the p
            if cost>=budget:
                return costs_before[cost_before],days-1
        else:
            cost=min(list(costs.keys()))
            if cost>=budget:
                return costs_before[cost_before],days-1
        days+=1
```

```
city_to_stay_maximum_days=given_budget(600)
```

```
print(city_to_stay_maximum_days)
```

```
('Paris', 7)
```

```
city_to_stay_minimum_days=given_budget(600, less_days=True)
```

```
print(city_to_stay_minimum_days)
```

```
('Mumbai', 7)
```

4. How does the answer to the previous question change if your budget is \$1000, \$2000, or \$1500?

- For 1000 dollars

```
city_to_stay_maximum_days=given_budget(1000)
print(city_to_stay_maximum_days)
```

('Mumbai', 26)

```
city_to_stay_minimum_days=given_budget(1000,less_days=True)
print(city_to_stay_minimum_days)
```

('London', 14)

- For 2000 dollars

```
city_to_stay_maximum_days=given_budget(2000)
print(city_to_stay_maximum_days)
```

('Mumbai', 77)

```
city_to_stay_minimum=given_budget(2000,less_days=True)
print(city_to_stay_minimum)
```

('London', 35)

- For 1500 dollars

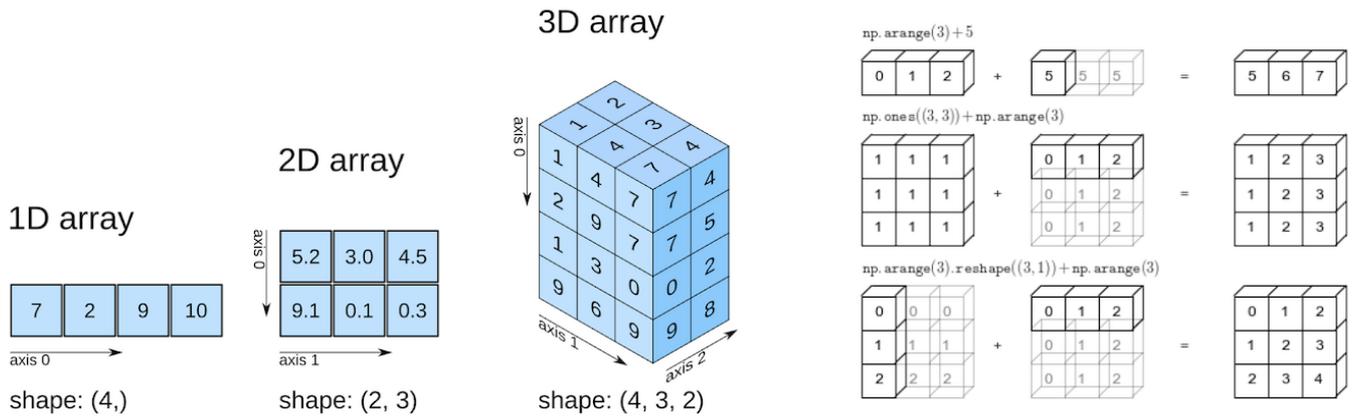
```
city_to_stay_maximum_days=given_budget(1500)
print(city_to_stay_maximum_days)
```

('Mumbai', 49)

```
city_to_stay_minimum_days=given_budget(1500,less_days=True)
print(city_to_stay_minimum_days)
```

('Paris', 24)

Numerical Computing with Python and Numpy



This tutorial series is a beginner-friendly introduction to programming and data analysis using the Python programming language. These tutorials take a practical and coding-focused approach. The best way to learn the material is to execute the code and experiment with it yourself.

This tutorial covers the following topics:

- Working with numerical data in Python
- Going from Python lists to Numpy arrays
- Multi-dimensional Numpy arrays and their benefits
- Array operations, broadcasting, indexing, and slicing
- Working with CSV data files using Numpy

How to run the code

This tutorial is an executable [Jupyter notebook](#) hosted on [Jovian](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Jupyter Notebooks: This tutorial is a [Jupyter notebook](#) - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc., instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things -

you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Working with numerical data

The "data" in *Data Analysis* typically refers to numerical data, e.g., stock prices, sales figures, sensor measurements, sports scores, database tables, etc. The [Numpy](#) library provides specialized data structures, functions, and other tools for numerical computing in Python. Let's work through an example to see why & how to use Numpy for working with numerical data.

Suppose we want to use climate data like the temperature, rainfall, and humidity to determine if a region is well suited for growing apples. A simple approach for doing this would be to formulate the relationship between the annual yield of apples (tons per hectare) and the climatic conditions like the average temperature (in degrees Fahrenheit), rainfall (in millimeters) & average relative humidity (in percentage) as a linear equation.

$$\text{yield_of_apples} = w_1 * \text{temperature} + w_2 * \text{rainfall} + w_3 * \text{humidity}$$

We're expressing the yield of apples as a weighted sum of the temperature, rainfall, and humidity. This equation is an approximation since the actual relationship may not necessarily be linear, and there may be other factors involved. But a simple linear model like this often works well in practice.

Based on some static analysis of historical data, we might come up with reasonable values for the weights w_1 , w_2 , and w_3 . Here's an example set of values:

$$w_1, w_2, w_3 = 0.3, 0.2, 0.5$$

Given some climate data for a region, we can now predict the yield of apples. Here's some sample data:

Region	Temp. (F)	Rainfall (mm)	Humidity (%)
Kanto	73	67	43
Johto	91	88	64
Hoenn	87	134	58
Sinnoh	102	43	37
Unova	69	96	70

To begin, we can define some variables to record climate data for a region.

```
kanto_temp = 73
kanto_rainfall = 67
kanto_humidity = 43
```

We can now substitute these variables into the linear equation to predict the yield of apples.

```
kanto_yield_apples = kanto_temp * w1 + kanto_rainfall * w2 + kanto_humidity * w3
kanto_yield_apples
```

```
print("The expected yield of apples in Kanto region is {} tons per hectare.".format(kar
```

The expected yield of apples in Kanto region is 56.8 tons per hectare.

To make it slightly easier to perform the above computation for multiple regions, we can represent the climate data for each region as a vector, i.e., a list of numbers.

```
kanto = [73, 67, 43]
johto = [91, 88, 64]
hoenn = [87, 134, 58]
sinnoh = [102, 43, 37]
unova = [69, 96, 70]
```

The three numbers in each vector represent the temperature, rainfall, and humidity data, respectively.

We can also represent the set of weights used in the formula as a vector.

```
weights = [w1, w2, w3]
```

We can now write a function `crop_yield` to calculate the yield of apples (or any other crop) given the climate data and the respective weights.

```
def crop_yield(region, weights):
    result = 0
    for x, w in zip(region, weights):
        result += x * w
    return result
```

```
crop_yield(kanto, weights)
```

56.8

```
crop_yield(johto, weights)
```

76.9

```
crop_yield(unova, weights)
```

74.9

Going from Python lists to Numpy arrays

The calculation performed by the `crop_yield` (element-wise multiplication of two vectors and taking a sum of the results) is also called the *dot product*. Learn more about dot product here:

<https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/dot-cross-products/v/vector-dot-product-and-vector-length> .

The Numpy library provides a built-in function to compute the dot product of two vectors. However, we must first convert the lists into Numpy arrays.

Let's install the Numpy library using the `pip` package manager.

```
!pip install numpy --upgrade --quiet
```

Next, let's import the `numpy` module. It's common practice to import `numpy` with the alias `np`.

```
import numpy as np
```

We can now use the `np.array` function to create Numpy arrays.

```
kanto = np.array([73, 67, 43])
```

```
kanto
```

```
array([73, 67, 43])
```

```
weights = np.array([w1, w2, w3])
```

```
weights
```

```
array([0.3, 0.2, 0.5])
```

Numpy arrays have the type `ndarray`.

```
type(kanto)
```

```
numpy.ndarray
```

```
type(weights)
```

```
numpy.ndarray
```

Just like lists, Numpy arrays support the indexing notation `[]`.

```
weights[0]
```

```
0.3
```

```
kanto[2]
```

```
43
```

Operating on Numpy arrays

We can now compute the dot product of the two vectors using the `np.dot` function.

```
np.dot(kanto, weights)
```

56.8

We can achieve the same result with low-level operations supported by Numpy arrays: performing an element-wise multiplication and calculating the resulting numbers' sum.

```
(kanto * weights).sum()
```

56.8

The `*` operator performs an element-wise multiplication of two arrays if they have the same size. The `sum` method calculates the sum of numbers in an array.

```
arr1 = np.array([1, 2, 3])  
arr2 = np.array([4, 5, 6])
```

```
arr1 * arr2
```

```
array([ 4, 10, 18])
```

```
arr2.sum()
```

15

Benefits of using Numpy arrays

Numpy arrays offer the following benefits over Python lists for operating on numerical data:

- **Ease of use:** You can write small, concise, and intuitive mathematical expressions like `(kanto * weights).sum()` rather than using loops & custom functions like `crop_yield`.
- **Performance:** Numpy operations and functions are implemented internally in C++, which makes them much faster than using Python statements & loops that are interpreted at runtime

Here's a comparison of dot products performed using Python loops vs. Numpy arrays on two vectors with a million elements each.

```
# Python lists  
arr1 = list(range(1000000))  
arr2 = list(range(1000000, 2000000))
```

```
# Numpy arrays  
arr1_np = np.array(arr1)  
arr2_np = np.array(arr2)
```

```
%%time  
result = 0  
for x1, x2 in zip(arr1, arr2):  
    result += x1*x2  
result
```

```
CPU times: user 146 ms, sys: 454 µs, total: 146 ms
```

```
Wall time: 144 ms
```

```
833332333333500000
```

```
%%time  
np.dot(arr1_np, arr2_np)
```

```
CPU times: user 2.72 ms, sys: 155 µs, total: 2.88 ms
```

```
Wall time: 1.72 ms
```

```
833332333333500000
```

As you can see, using `np.dot` is 100 times faster than using a `for` loop. This makes Numpy especially useful while working with really large datasets with tens of thousands or millions of data points.

Let's save our work before continuing.

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-numerical-computing-with-numpy" on  
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-numerical-computing-with-numpy
```

```
'https://jovian.ai/evanmarie/python-numerical-computing-with-numpy'
```

Multi-dimensional Numpy arrays

We can now go one step further and represent the climate data for all the regions using a single 2-dimensional Numpy array.

```
climate_data = np.array([[73, 67, 43],  
                        [91, 88, 64],  
                        [87, 134, 58],  
                        [102, 43, 37],  
                        [69, 96, 70]])
```

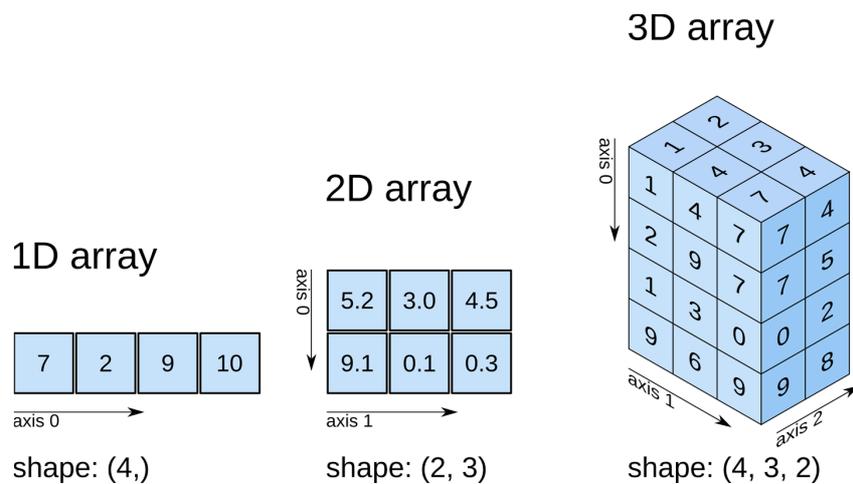
```
climate_data
```

```
array([[ 73,  67,  43],  
       [ 91,  88,  64],  
       [ 87, 134,  58],  
       [102,  43,  37],  
       [ 69,  96,  70]])
```

If you've taken a linear algebra class in high school, you may recognize the above 2-d array as a matrix with five rows and three columns. Each row represents one region, and the columns represent temperature, rainfall, and

humidity, respectively.

Numpy arrays can have any number of dimensions and different lengths along each dimension. We can inspect the length along each dimension using the `.shape` property of an array.



```
# 2D array (matrix)
climate_data.shape
```

(5, 3)

```
weights
```

```
array([0.3, 0.2, 0.5])
```

```
# 1D array (vector)
weights.shape
```

(3,)

```
# 3D array
arr3 = np.array([
    [[11, 12, 13],
     [13, 14, 15]],
    [[15, 16, 17],
     [17, 18, 19.5]]])
```

```
arr3.shape
```

(2, 2, 3)

All the elements in a numpy array have the same data type. You can check the data type of an array using the `.dtype` property.

```
weights.dtype
```

```
dtype('float64')
```

```
climate_data.dtype
```

```
dtype('int64')
```

If an array contains even a single floating point number, all the other elements are also converted to floats.

```
arr3.dtype
```

```
dtype('float64')
```

We can now compute the predicted yields of apples in all the regions, using a single matrix multiplication between `climate_data` (a 5x3 matrix) and `weights` (a vector of length 3). Here's what it looks like visually:

$$\begin{bmatrix} 73 & 67 & 43 \\ 91 & 88 & 64 \\ \vdots & \vdots & \vdots \\ 69 & 96 & 70 \end{bmatrix} \times \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \end{bmatrix}$$

You can learn about matrices and matrix multiplication by watching the first 3-4 videos of this playlist:

<https://www.youtube.com/watch?v=xyAuNHPsq-g&list=PLFD0EB975BA0CC1E0&index=1> .

We can use the `np.matmul` function or the `@` operator to perform matrix multiplication.

```
np.matmul(climate_data, weights)
```

```
array([56.8, 76.9, 81.9, 57.7, 74.9])
```

```
climate_data @ weights
```

```
array([56.8, 76.9, 81.9, 57.7, 74.9])
```

Working with CSV data files

Numpy also provides helper functions reading from & writing to files. Let's download a file `climate.txt`, which contains 10,000 climate measurements (temperature, rainfall & humidity) in the following format:

```
temperature,rainfall,humidity
25.00,76.00,99.00
39.00,65.00,70.00
59.00,45.00,77.00
84.00,63.00,38.00
66.00,50.00,52.00
41.00,94.00,77.00
91.00,57.00,96.00
49.00,96.00,99.00
67.00,20.00,28.00
...
```

This format of storing data is known as *comma-separated values* or CSV.

CSVs: A comma-separated values (CSV) file is a delimited text file that uses a comma to separate values. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. A CSV file typically stores tabular data (numbers and text) in plain text, in which case each line will have the same number of fields. (Wikipedia)

To read this file into a numpy array, we can use the `genfromtxt` function.

```
import urllib.request

urllib.request.urlretrieve(
    'https://gist.github.com/BirajCoder/a4ffcb76fd6fb221d76ac2ee2b8584e9/raw/4054f90adf
    'climate.txt')
```

```
('climate.txt', <http.client.HTTPMessage at 0x7f2d8629c850>)
```

```
climate_data = np.genfromtxt('climate.txt', delimiter=',', skip_header=1)
```

```
climate_data
```

```
array([[25., 76., 99.],
       [39., 65., 70.],
       [59., 45., 77.],
       ...,
       [99., 62., 58.],
       [70., 71., 91.],
       [92., 39., 76.]])
```

```
climate_data.shape
```

```
(10000, 3)
```

We can now perform a matrix multiplication using the `@` operator to predict the yield of apples for the entire dataset using a given set of weights.

```
weights = np.array([0.3, 0.2, 0.5])
```

```
yields = climate_data @ weights
```

```
yields
```

```
array([72.2, 59.7, 65.2, ..., 71.1, 80.7, 73.4])
```

```
yields.shape
```

```
(10000,)
```

Let's add the `yields` to `climate_data` as a fourth column using the `np.concatenate` function.

```
climate_results = np.concatenate((climate_data, yields.reshape(10000, 1)), axis=1)
```

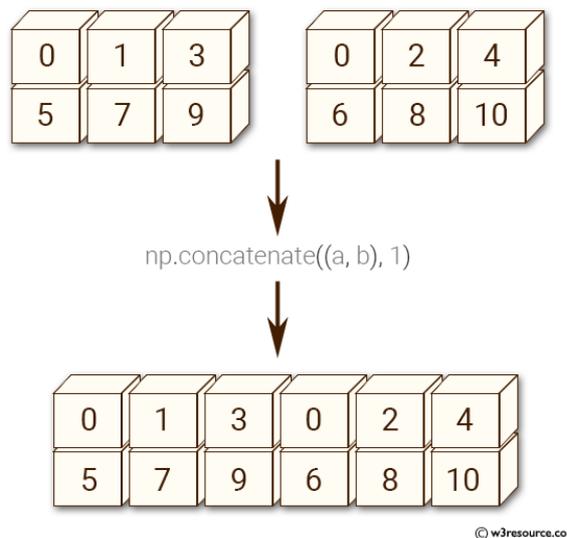
```
climate_results
```

```
array([[25. , 76. , 99. , 72.2],  
       [39. , 65. , 70. , 59.7],  
       [59. , 45. , 77. , 65.2],  
       ...,  
       [99. , 62. , 58. , 71.1],  
       [70. , 71. , 91. , 80.7],  
       [92. , 39. , 76. , 73.4]])
```

There are a couple of subtleties here:

- Since we wish to add new columns, we pass the argument `axis=1` to `np.concatenate`. The `axis` argument specifies the dimension for concatenation.
- The arrays should have the same number of dimensions, and the same length along each except the dimension used for concatenation. We use the [np.reshape](#) function to change the shape of `yields` from `(10000,)` to `(10000,1)`.

Here's a visual explanation of `np.concatenate` along `axis=1` (can you guess what `axis=0` results in?):



The best way to understand what a Numpy function does is to experiment with it and read the documentation to learn about its arguments & return values. Use the cells below to experiment with `np.concatenate` and `np.reshape`.

Let's write the final results from our computation above back to a file using the `np.savetxt` function.

```
climate_results
```

```
array([[25. , 76. , 99. , 72.2],
       [39. , 65. , 70. , 59.7],
       [59. , 45. , 77. , 65.2],
       ...,
       [99. , 62. , 58. , 71.1],
       [70. , 71. , 91. , 80.7],
       [92. , 39. , 76. , 73.4]])
```

```
np.savetxt('climate_results.txt',
           climate_results,
           fmt='%.2f',
           delimiter=',',
           header='temperature,rainfall,humidity,yeild_apples',
           comments='')
```

The results are written back in the CSV format to the file `climate_results.txt` .

```
temperature,rainfall,humidity,yeild_apples
25.00,76.00,99.00,72.20
39.00,65.00,70.00,59.70
59.00,45.00,77.00,65.20
84.00,63.00,38.00,56.80
...
```

Numpy provides hundreds of functions for performing operations on arrays. Here are some commonly used functions:

- Mathematics: `np.sum`, `np.exp`, `np.round`, arithmetic operators
- Array manipulation: `np.reshape`, `np.stack`, `np.concatenate`, `np.split`
- Linear Algebra: `np.matmul`, `np.dot`, `np.transpose`, `np.eigvals`
- Statistics: `np.mean`, `np.median`, `np.std`, `np.max`

How to find the function you need? The easiest way to find the right function for a specific operation or use-case is to do a web search. For instance, searching for "How to join numpy arrays" leads to [this tutorial on array concatenation](#).

You can find a full list of array functions here: <https://numpy.org/doc/stable/reference/routines.html>

Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. [Jovian](#) offers an easy way of saving and sharing your Jupyter notebooks online.

```
# Install the library
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-numerical-computing-with-numpy" on
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-numerical-computing-with-numpy
```

```
'https://jovian.ai/evanmarie/python-numerical-computing-with-numpy'
```

The first time you run `jovian.commit`, you'll be asked to provide an API Key to securely upload the notebook to your Jovian account. You can get the API key from your [Jovian profile page](#) after logging in / signing up.

`jovian.commit` uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work.

Arithmetic operations, broadcasting and comparison

Numpy arrays support arithmetic operators like `+`, `-`, `*`, etc. You can perform an arithmetic operation with a single number (also called scalar) or with another array of the same shape. Operators make it easy to write mathematical expressions with multi-dimensional arrays.

```
arr2 = np.array([[1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 1, 2, 3]])
```

```
arr3 = np.array([[11, 12, 13, 14],
                 [15, 16, 17, 18],
                 [19, 11, 12, 13]])
```

```
# Adding a scalar
arr2 + 3
```

```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12,  4,  5,  6]])
```

```
# Element-wise subtraction
arr3 - arr2
```

```
array([[10, 10, 10, 10],
       [10, 10, 10, 10],
       [10, 10, 10, 10]])
```

```
# Division by scalar
```

```
arr2 / 2
```

```
array([[0.5, 1. , 1.5, 2. ],
       [2.5, 3. , 3.5, 4. ],
       [4.5, 0.5, 1. , 1.5]])
```

```
# Element-wise multiplication
```

```
arr2 * arr3
```

```
array([[ 11,  24,  39,  56],
       [ 75,  96, 119, 144],
       [171,  11,  24,  39]])
```

```
# Modulus with scalar
```

```
arr2 % 4
```

```
array([[1, 2, 3, 0],
       [1, 2, 3, 0],
       [1, 1, 2, 3]])
```

Array Broadcasting

Numpy arrays also support *broadcasting*, allowing arithmetic operations between two arrays with different numbers of dimensions but compatible shapes. Let's look at an example to see how it works.

```
arr2 = np.array([[1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 1, 2, 3]])
```

```
arr2.shape
```

```
(3, 4)
```

```
arr4 = np.array([4, 5, 6, 7])
```

```
arr4.shape
```

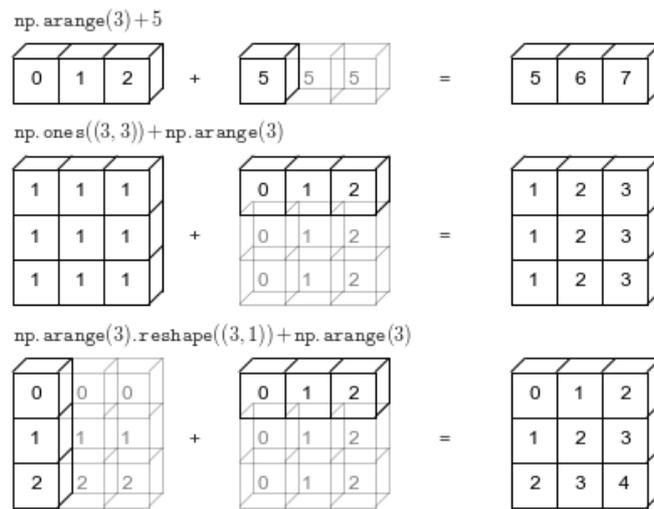
```
(4,)
```

```
arr2 + arr4
```

```
array([[ 5,  7,  9, 11],
       [ 9, 11, 13, 15],
       [13,  6,  8, 10]])
```

When the expression `arr2 + arr4` is evaluated, `arr4` (which has the shape `(4,)`) is replicated three times to match the shape `(3, 4)` of `arr2`. Numpy performs the replication without actually creating three copies of the

smaller dimension array, thus improving performance and using lower memory.



Broadcasting only works if one of the arrays can be replicated to match the other array's shape.

```
arr5 = np.array([7, 8])
```

```
arr5.shape
```

```
(2,)
```

```
arr2 + arr5
```

```
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_38/897499081.py in <module>
----> 1 arr2 + arr5
```

ValueError: operands could not be broadcast together with shapes (3,4) (2,)

In the above example, even if `arr5` is replicated three times, it will not match the shape of `arr2`. Hence `arr2 + arr5` cannot be evaluated successfully. Learn more about broadcasting here:

<https://numpy.org/doc/stable/user/basics.broadcasting.html>.

Array Comparison

Numpy arrays also support comparison operations like `==`, `!=`, `>` etc. The result is an array of booleans.

```
arr1 = np.array([[1, 2, 3], [3, 4, 5]])
arr2 = np.array([[2, 2, 3], [1, 2, 5]])
```

```
arr1 == arr2
```

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

```
arr1 != arr2
```

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

```
arr1 >= arr2
```

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

```
arr1 < arr2
```

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

Array comparison is frequently used to count the number of equal elements in two arrays using the `sum` method. Remember that `True` evaluates to `1` and `False` evaluates to `0` when booleans are used in arithmetic operations.

```
(arr1 == arr2).sum()
```

3

Array indexing and slicing

Numpy extends Python's list indexing notation using `[]` to multiple dimensions in an intuitive fashion. You can provide a comma-separated list of indices or ranges to select a specific element or a subarray (also called a slice) from a Numpy array.

```
arr3 = np.array([
    [[11, 12, 13, 14],
     [13, 14, 15, 19]],

    [[15, 16, 17, 21],
     [63, 92, 36, 18]],

    [[98, 32, 81, 23],
     [17, 18, 19.5, 43]]])
```

```
arr3.shape
```

(3, 2, 4)

```
# Single element
arr3[1, 1, 2]
```

36.0

```
# Subarray using ranges
arr3[1:, 0:1, :2]
```

```
array([[[15., 16.],
        [[98., 32.]])])
```

```
# Mixing indices and ranges
arr3[1:, 1, 3]
```

```
array([18., 43.]
```

```
# Mixing indices and ranges
arr3[1:, 1, :3]
```

```
array([[63. , 92. , 36. ],
       [17. , 18. , 19.5]])
```

```
# Using fewer indices
arr3[1]
```

```
array([[15., 16., 17., 21.],
       [63., 92., 36., 18.]])
```

```
# Using fewer indices
arr3[:2, 1]
```

```
array([[13., 14., 15., 19.],
       [63., 92., 36., 18.]])
```

```
# Using too many indices
arr3[1,3,2,1]
```

```
-----
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_38/2043861726.py in <module>
      1 # Using too many indices
----> 2 arr3[1,3,2,1]
```

IndexError: too many indices for array: array is 3-dimensional, but 4 were indexed

The notation and its results can seem confusing at first, so take your time to experiment and become comfortable with it. Use the cells below to try out some examples of array indexing and slicing, with different combinations of indices and ranges. Here are some more examples demonstrated visually:

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

```
arr3 = np.array([
    [[11, 12, 13, 14],
     [13, 14, 15, 19]],

    [[15, 16, 17, 21],
     [63, 92, 36, 18]],

    [[98, 32, 81, 23],
     [17, 18, 19.5, 43]]])
```

```
arr4 = np.array([
    [[10, 29, 38, 47, 56],
     [11, 22, 33, 44, 55],
     [91, 82, 73, 64, 45]],

    [[11, 12, 13, 14, 33],
     [13, 14, 15, 19, 44],
     [63, 92, 36, 18, 88]],

    [[10, 29, 38, 47, 56],
     [11, 22, 33, 44, 55],
     [91, 82, 73, 64, 45]]
])
```

```
arr4[2, 1, 2]
```

33

```
arr4[1, 2, 4]
```

88

Other ways of creating Numpy arrays

Numpy also provides some handy functions to create arrays of desired shapes with fixed or random values. Check out the [official documentation](#) or use the `help` function to learn more.

```
# All zeros
np.zeros((3, 2))
```

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

```
# All ones
np.ones([2, 2, 3])
```

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

```
[[1., 1., 1.],
 [1., 1., 1.]])
```

```
# Identity matrix
np.eye(3)
```

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

```
# Random vector
np.random.rand(5)
```

```
array([0.77980134, 0.61381521, 0.33015617, 0.13952484, 0.11753095])
```

```
# Random matrix
np.random.randn(2, 3) # rand vs. randn - what's the difference?
```

```
array([[ -0.22593637,  0.10270113, -1.76438044],
       [ 1.12805151,  0.30789975, -1.31822607]])
```

```
# Fixed value
np.full([2, 3], 42)
```

```
array([[42, 42, 42],
       [42, 42, 42]])
```

```
# Range with start, end and step
np.arange(10, 90, 3)
```

```
array([10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49, 52, 55, 58,
       61, 64, 67, 70, 73, 76, 79, 82, 85, 88])
```

```
# Equally spaced numbers in a range
np.linspace(3, 27, 9)
```

```
array([ 3.,  6.,  9., 12., 15., 18., 21., 24., 27.])
```

Save and commit

Let's record a snapshot of our work using `jovian.commit`.

```
# Install the library
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit()
```

Exercises

Try the following exercises to become familiar with Numpy arrays and practice your skills:

- Assignment on Numpy array functions: <https://jovian.ai/aakashns/numpy-array-operations>
- (Optional) 100 numpy exercises: <https://jovian.ai/aakashns/100-numpy-exercises>

Summary and Further Reading

With this, we complete our discussion of numerical computing with Numpy. We've covered the following topics in this tutorial:

- Going from Python lists to Numpy arrays
- Operating on Numpy arrays
- Benefits of using Numpy arrays over lists
- Multi-dimensional Numpy arrays
- Working with CSV data files
- Arithmetic operations and broadcasting
- Array indexing and slicing
- Other ways of creating Numpy arrays

Check out the following resources for learning more about Numpy:

- Official tutorial: <https://numpy.org/devdocs/user/quickstart.html>
- Numpy tutorial on W3Schools: https://www.w3schools.com/python/numpy_intro.asp
- Advanced Numpy (exploring the internals): http://scipy-lectures.org/advanced/advanced_numpy/index.html

You are ready to move on to the next tutorial: [Analyzing Tabular Data using Pandas](#).

Questions for Revision

Try answering the following questions to test your understanding of the topics covered in this notebook:

1. What is a vector?
2. How do you represent vectors using a Python list? Give an example.
3. What is a dot product of two vectors?
4. Write a function to compute the dot product of two vectors.
5. What is Numpy?
6. How do you install Numpy?
7. How do you import the numpy module?
8. What does it mean to import a module with an alias? Give an example.
9. What is the commonly used alias for numpy?
10. What is a Numpy array?

11. How do you create a Numpy array? Give an example.
12. What is the type of Numpy arrays?
13. How do you access the elements of a Numpy array?
14. How do you compute the dot product of two vectors using Numpy?
15. What happens if you try to compute the dot product of two vectors which have different sizes?
16. How do you compute the element-wise product of two Numpy arrays?
17. How do you compute the sum of all the elements in a Numpy array?
18. What are the benefits of using Numpy arrays over Python lists for operating on numerical data?
19. Why do Numpy array operations have better performance compared to Python functions and loops?
20. Illustrate the performance difference between Numpy array operations and Python loops using an example.
21. What are multi-dimensional Numpy arrays?
22. Illustrate the creation of Numpy arrays with 2, 3, and 4 dimensions.
23. How do you inspect the number of dimensions and the length along each dimension in a Numpy array?
24. Can the elements of a Numpy array have different data types?
25. How do you check the data type of the elements of a Numpy array?
26. What is the data type of a Numpy array?
27. What is the difference between a matrix and a 2D Numpy array?
28. How do you perform matrix multiplication using Numpy?
29. What is the @ operator used for in Numpy?
30. What is the CSV file format?
31. How do you read data from a CSV file using Numpy?
32. How do you concatenate two Numpy arrays?
33. What is the purpose of the axis argument of np.concatenate?
34. When are two Numpy arrays compatible for concatenation?
35. Give an example of two Numpy arrays that can be concatenated.
36. Give an example of two Numpy arrays that cannot be concatenated.
37. What is the purpose of the np.reshape function?
38. What does it mean to "reshape" a Numpy array?
39. How do you write a numpy array into a CSV file?
40. Give some examples of Numpy functions for performing mathematical operations.
41. Give some examples of Numpy functions for performing array manipulation.
42. Give some examples of Numpy functions for performing linear algebra.
43. Give some examples of Numpy functions for performing statistical operations.
44. How do you find the right Numpy function for a specific operation or use case?
45. Where can you see a list of all the Numpy array functions and operations?
46. What are the arithmetic operators supported by Numpy arrays? Illustrate with examples.
47. What is array broadcasting? How is it useful? Illustrate with an example.

48. Give some examples of arrays that are compatible for broadcasting?
49. Give some examples of arrays that are not compatible for broadcasting?
50. What are the comparison operators supported by Numpy arrays? Illustrate with examples.
51. How do you access a specific subarray or slice from a Numpy array?
52. Illustrate array indexing and slicing in multi-dimensional Numpy arrays with some examples.
53. How do you create a Numpy array with a given shape containing all zeros?
54. How do you create a Numpy array with a given shape containing all ones?
55. How do you create an identity matrix of a given shape?
56. How do you create a random vector of a given length?
57. How do you create a Numpy array with a given shape with a fixed value for each element?
58. How do you create a Numpy array with a given shape containing randomly initialized elements?
59. What is the difference between `np.random.rand` and `np.random.randn`? Illustrate with examples.
60. What is the difference between `np.arange` and `np.linspace`? Illustrate with examples.

100 numpy exercises

Source: <https://github.com/rougier/numpy-100>

This was my first dive into Numpy, which I thoroughly enjoyed. Numpy might be one of my new favorite python experiences! My variable names and such get a little silly about half way through this assignment, because let's be honest, this was A LOT. But I truly enjoyed the assignment!

```
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit(project='100-numpy-exercises')
```

```
[jovian] Creating a new project "evanmarie/100-numpy-exercises"
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/100-numpy-exercises
```

```
'https://jovian.ai/evanmarie/100-numpy-exercises'
```

`jovian.commit` uploads the notebook to your [Jovian.ml](https://jovian.ml) account, captures the Python environment and creates a shareable link for your notebook as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work. Learn more: <https://jovian.ml/docs/>

```
# Uncomment the next line if you need install numpy  
# !pip install numpy --upgrade
```

1. Import the numpy package under the name np (★☆☆)

```
import numpy as np
```

2. Print the numpy version and the configuration (★☆☆)

```
print(np.__version__)  
print(np.__config__.show())
```

1.20.3

blas_info:

```
libraries = ['cblas', 'blas', 'cblas', 'blas']  
library_dirs = ['/opt/conda/lib']  
include_dirs = ['/opt/conda/include']  
language = c  
define_macros = [('HAVE_CBLAS', None)]
```

blas_opt_info:

```
define_macros = [('NO_ATLAS_INFO', 1), ('HAVE_CBLAS', None)]
```

```

libraries = ['cblas', 'blas', 'cblas', 'blas']
library_dirs = ['/opt/conda/lib']
include_dirs = ['/opt/conda/include']
language = c
lapack_info:
  libraries = ['lapack', 'blas', 'lapack', 'blas']
  library_dirs = ['/opt/conda/lib']
  language = f77
lapack_opt_info:
  libraries = ['lapack', 'blas', 'lapack', 'blas', 'cblas', 'blas', 'cblas', 'blas']
  library_dirs = ['/opt/conda/lib']
  language = c
  define_macros = [('NO_ATLAS_INFO', 1), ('HAVE_CBLAS', None)]
  include_dirs = ['/opt/conda/include']
None

```

3. Create a null vector of size 10 (☆☆☆)

```

array_1 = np.zeros(10)
array_1

```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]
```

4. How to find the memory size of any array (☆☆☆)

```
array_1.itemsize * array_1.size # Each float takes up 8 bytes times 10
```

```
80
```

5. How to get the documentation of the numpy add function from the command line? (☆☆☆)

```
help(np.add)
```

Help on ufunc:

```
add = <ufunc 'add'>
```

```
add(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None,
subok=True[, signature, extobj])
```

Add arguments element-wise.

Parameters

x1, x2 : array_like

The arrays to be added.

If `x1.shape != x2.shape`, they must be broadcastable to a common shape (which becomes the shape of the output).

`out` : ndarray, None, or tuple of ndarray and None, optional

A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

`where` : array_like, optional

This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value.

Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

****kwargs**

For other keyword-only arguments, see the [:ref:`ufunc docs <ufuncs.kwargs>](#).

Returns

`add` : ndarray or scalar

The sum of `x1` and `x2`, element-wise.

This is a scalar if both `x1` and `x2` are scalars.

Notes

Equivalent to `x1 + x2` in terms of array broadcasting.

Examples

```
>>> np.add(1.0, 4.0)
```

```
5.0
```

```
>>> x1 = np.arange(9.0).reshape((3, 3))
```

```
>>> x2 = np.arange(3.0)
```

```
>>> np.add(x1, x2)
```

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

The `+` operator can be used as a shorthand for `np.add` on ndarrays.

```
>>> x1 = np.arange(9.0).reshape((3, 3))
```

```
>>> x2 = np.arange(3.0)
```

```
>>> x1 + x2
array([[ 0.,  2.,  4.],
       [ 3.,  5.,  7.],
       [ 6.,  8., 10.]])
```

6. Create a null vector of size 10 but the fifth value which is 1 (☆☆☆)

```
array_2 = np.zeros(10)
array_2[4] = 1
array_2
```

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

7. Create a vector with values ranging from 10 to 49 (☆☆☆)

```
array_3 = np.arange(10, 50)
array_3
```

```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
       27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
       44, 45, 46, 47, 48, 49])
```

8. Reverse a vector (first element becomes last) (☆☆☆)

```
array_3[::-1]
```

```
array([49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33,
       32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16,
       15, 14, 13, 12, 11, 10])
```

9. Create a 3×3 matrix with values ranging from 0 to 8 (☆☆☆)

```
array_4 = np.arange(0, 9).reshape(3, 3)
array_4
```

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

10. Find indices of non-zero elements from [1,2,0,0,4,0] (☆☆☆)

```
arr_example = np.array([1,2,0,0,4,0])
print(np.where(arr_example == 0))
```

```
(array([2, 3, 5]),)
```

Save your progress by committing your work to Jovian

```
import jovian
```

```
jovian.commit(project='numpy-100-exercises')
```

```
[jovian] Updating notebook "evanmarie/numpy-100-exercises" on https://jovian.ai  
[jovian] Committed successfully! https://jovian.ai/evanmarie/numpy-100-exercises  
'https://jovian.ai/evanmarie/numpy-100-exercises'
```

11. Create a 3×3 identity matrix (★☆☆)

```
identity = np.eye(3)  
identity
```

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

12. Create a 3×3×3 array with random values (★☆☆)

```
random_array = np.random.random((3, 3, 3))  
random_array
```

```
array([[[0.19850512, 0.00964888, 0.10568904],  
       [0.73420047, 0.4867579 , 0.63050892],  
       [0.96879387, 0.51525811, 0.67944852]],  
      [[0.34238019, 0.94720966, 0.46233957],  
       [0.87591195, 0.70529463, 0.26642908],  
       [0.98817607, 0.2292137 , 0.7387402 ]],  
      [[0.20370091, 0.06154962, 0.74795698],  
       [0.86172547, 0.51455635, 0.57400702],  
       [0.82936599, 0.49070134, 0.31329223]]])
```

13. Create a 10×10 array with random values and find the minimum and maximum values (★☆☆)

```
random10 = np.random.random((10, 10))  
print(random10.min(), random10.max())
```

```
0.00033851810711504893 0.9965136001916162
```

14. Create a random vector of size 30 and find the mean value (★☆☆)

```
thirty = np.random.random(30)  
print(thirty.mean())
```

```
0.498162578683955
```

15. Create a 2d array with 1 on the border and 0 inside (★☆☆)

```
two_d = np.zeros((5, 5))
two_d[0] = 1
two_d[-1] = 1
two_d[:, 0] = 1
two_d[:, -1] = 1
two_d
```

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

16. How to add a border (filled with 0's) around an existing array? (★☆☆)

```
zeros_array = np.zeros((7, 7))
zeros_array[1:-1, 1:-1] = two_d
zeros_array
```

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

17. What is the result of the following expression? (★☆☆)

```
0 * np.nan
np.nan == np.nan
np.inf > np.nan
np.nan - np.nan
np.nan in set([np.nan])
0.3 == 3 * 0.1
```

0 * np.nan is nan
np.nan == np.nan is False
np.inf > np.nan is False
np.nan - np.nan is nan
np.nan in set([np.nan]) is True ({nan})
0.3 == 3 * 0.1 is False (is 0.30000000000000004)

18. Create a 5×5 matrix with values 1,2,3,4 just below the diagonal (★☆☆)

```
five_squared = np.diag([1, 2, 3, 4], -1)
five_squared
```

```
array([[0, 0, 0, 0, 0],
       [1, 0, 0, 0, 0],
       [0, 2, 0, 0, 0],
       [0, 0, 3, 0, 0],
       [0, 0, 0, 4, 0]])
```

19. Create a 8×8 matrix and fill it with a checkerboard pattern (★☆☆)

```
eights2 = np.zeros((8, 8))
np.tile(np.eye(2), (4,4))
```

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

20. Consider a (6,7,8) shape array, what is the index (x,y,z) of the 100th element?

```
crazy_array = np.arange(1, (6*7*8 + 1)).reshape(6, 7, 8)
crazy_array
```

```
array([[[ 1,  2,  3,  4,  5,  6,  7,  8],
        [ 9, 10, 11, 12, 13, 14, 15, 16],
        [17, 18, 19, 20, 21, 22, 23, 24],
        [25, 26, 27, 28, 29, 30, 31, 32],
        [33, 34, 35, 36, 37, 38, 39, 40],
        [41, 42, 43, 44, 45, 46, 47, 48],
        [49, 50, 51, 52, 53, 54, 55, 56]],

       [[ 57,  58,  59,  60,  61,  62,  63,  64],
        [ 65,  66,  67,  68,  69,  70,  71,  72],
        [ 73,  74,  75,  76,  77,  78,  79,  80],
        [ 81,  82,  83,  84,  85,  86,  87,  88],
        [ 89,  90,  91,  92,  93,  94,  95,  96],
        [ 97,  98,  99, 100, 101, 102, 103, 104],
        [105, 106, 107, 108, 109, 110, 111, 112]],

       [[113, 114, 115, 116, 117, 118, 119, 120],
        [121, 122, 123, 124, 125, 126, 127, 128],
        [129, 130, 131, 132, 133, 134, 135, 136],
        [137, 138, 139, 140, 141, 142, 143, 144],
        [145, 146, 147, 148, 149, 150, 151, 152],
        [153, 154, 155, 156, 157, 158, 159, 160],
        [161, 162, 163, 164, 165, 166, 167, 168]],

       [[169, 170, 171, 172, 173, 174, 175, 176],
```

```

[177, 178, 179, 180, 181, 182, 183, 184],
[185, 186, 187, 188, 189, 190, 191, 192],
[193, 194, 195, 196, 197, 198, 199, 200],
[201, 202, 203, 204, 205, 206, 207, 208],
[209, 210, 211, 212, 213, 214, 215, 216],
[217, 218, 219, 220, 221, 222, 223, 224]],

[[225, 226, 227, 228, 229, 230, 231, 232],
 [233, 234, 235, 236, 237, 238, 239, 240],
 [241, 242, 243, 244, 245, 246, 247, 248],
 [249, 250, 251, 252, 253, 254, 255, 256],
 [257, 258, 259, 260, 261, 262, 263, 264],
 [265, 266, 267, 268, 269, 270, 271, 272],
 [273, 274, 275, 276, 277, 278, 279, 280]],

[[281, 282, 283, 284, 285, 286, 287, 288],
 [289, 290, 291, 292, 293, 294, 295, 296],
 [297, 298, 299, 300, 301, 302, 303, 304],
 [305, 306, 307, 308, 309, 310, 311, 312],
 [313, 314, 315, 316, 317, 318, 319, 320],
 [321, 322, 323, 324, 325, 326, 327, 328],
 [329, 330, 331, 332, 333, 334, 335, 336]]])

```

```
crazy_array[1, 5, 3]
```

100

Save your progress by committing your work to Jovian

```
import jovian
```

```
jovian.commit()
```

```

[jovian] Updating notebook "evanmarie/numpy-100-exercises" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/numpy-100-exercises
'https://jovian.ai/evanmarie/numpy-100-exercises'

```

21. Create a checkerboard 8×8 matrix using the tile function (★☆☆)

```

eights2 = np.zeros((8, 8))
np.tile(np.eye(2), (4,4))

```

```

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

```

```
[1., 0., 1., 0., 1., 0., 1., 0.],  
[0., 1., 0., 1., 0., 1., 0., 1.]])
```

22. Normalize a 5×5 random matrix (★☆☆)

```
rando = np.random.random((5, 5))  
rando
```

```
array([[0.44580707, 0.57502738, 0.5888955 , 0.06127456, 0.24232189],  
       [0.85266269, 0.56492298, 0.30518092, 0.51954604, 0.63795381],  
       [0.35370109, 0.45429207, 0.93720231, 0.65250434, 0.85917984],  
       [0.35629031, 0.60409602, 0.49500875, 0.48083454, 0.26082955],  
       [0.49279586, 0.95755474, 0.98170708, 0.18931823, 0.98689634]])
```

23. Create a custom dtype that describes a color as four unsigned bytes (RGBA) (★☆☆)

```
my_datatype = np.dtype(('i4', [('red', 'u1'), ('green', 'u1'), ('blue', 'u1'), ('alpha',  
myarray = np.array(0x12345678, dtype = my_datatype)  
myarray['red']
```

```
array(120, dtype=uint8)
```

24. Multiply a 5×3 matrix by a 3×2 matrix (real matrix product) (★☆☆)

```
five_three = np.arange(15).reshape(5, 3)  
three_two = np.arange(6).reshape(3, 2)  
together = np.dot(five_three, three_two)  
together
```

```
array([[ 10,  13],  
       [ 28,  40],  
       [ 46,  67],  
       [ 64,  94],  
       [ 82, 121]])
```

25. Given a 1D array, negate all elements which are between 3 and 8, in place. (★☆☆)

```
negate_me = np.arange(15)
```

```
# getting indices of integers between 3 and 8 and setting them to new values
```

```
negate_me[np.logical_and(negate_me>3, negate_me<8)] = -negate_me[np.logical_and(negate_me  
negate_me
```

```
array([ 0,  1,  2,  3, -4, -5, -6, -7,  8,  9, 10, 11, 12, 13, 14])
```

```
messing_around = np.arange(27).reshape(3, 3, 3)
```

```
messing_around
```

```
array([[[ 0,  1,  2],
```

```
[ 3,  4,  5],
 [ 6,  7,  8]],

[[ 9, 10, 11],
 [12, 13, 14],
 [15, 16, 17]],

[[18, 19, 20],
 [21, 22, 23],
 [24, 25, 26]]])
```

```
messing_around[np.logical_and(messing_around > 10, messing_around < 20)] = -messing_arc
```

```
messing_around
```

```
array([[[ 0,  1,  2],
 [ 3,  4,  5],
 [ 6,  7,  8]],

 [[ 9, 10, -11],
 [-12, -13, -14],
 [-15, -16, -17]],

 [[-18, -19,  20],
 [ 21,  22,  23],
 [ 24,  25,  26]])
```

26. What is the output of the following script? (☆☆☆)

```
# Author: Jake VanderPlas

print(sum(range(5),-1))
from numpy import *
print(sum(range(5),-1))
```

9

27. Consider an integer vector Z, which of these expressions are legal? (☆☆☆)

```
Z**Z
2 << Z >> 2
Z <- Z
1j*Z
Z/1/1
Z<Z>Z
```

```
Z**Z - legal
2 << Z >> 2 - legal
Z <- Z - legal
1j*Z - legal
Z/1/1 - legal
Z<Z>Z - not legal
```

File "/tmp/ipykernel_37/2362510792.py", line 6

```
Z<Z>Z - not legal
      ^
```

SyntaxError: invalid syntax

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

28. What are the result of the following expressions?

```
np.array(0) / np.array(0)
np.array(0) // np.array(0)
np.array([np.nan]).astype(int).astype(float)
```

np.array(0) / np.array(0) - nan
np.array(0) // np.array(0) - 0
np.array([np.nan]).astype(int).astype(float) - array([-9.22337204e+18])

29. How to round away from zero a float array ? (☆☆☆)

```
floaties = np.array([1.23, 2.22, 3.33, -1.11, 4.44, -5.55])

round_floaties = np.zeros_like(floaties, dtype = int)

round_floaties[floaties >= 0] = np.ceil(floaties[floaties >= 0])
round_floaties[floaties < 0] = np.floor(floaties[floaties < 0])

print("Rounded Floaties: ", round_floaties)
```

Rounded Floaties: [2 3 4 -2 5 -6]

30. How to find common values between two arrays? (☆☆☆)

```
array_uno = np.arange(15).reshape(3, 5)
array_zwei = np.arange(9).reshape(3, 3)

print("Array Uno: \n", array_uno)
print("Array Zwei: \n", array_zwei)
print("Intersection time: ", np.intersect1d(array_uno, array_zwei))
```

Array Uno:

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
```

```
[10 11 12 13 14]]
```

Array Zwei:

```
[[0 1 2]
```

```
[3 4 5]
```

```
[6 7 8]]
```

Intersection time: [0 1 2 3 4 5 6 7 8]

Save your progress by committing your work to Jovian

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/numpy-100-exercises" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/numpy-100-exercises
'https://jovian.ai/evanmarie/numpy-100-exercises'
```

31. How to ignore all numpy warnings (not recommended)? (☆☆☆)

```
np.seterr(all="ignore")
```

```
{'divide': 'warn', 'over': 'warn', 'under': 'ignore', 'invalid': 'warn'}
```

32. Is the following expressions true? (☆☆☆)

```
np.sqrt(-1) == np.emath.sqrt(-1)
```

No, `np.sqrt(-1) == nan`, and `np.emath.sqrt(-1) == 1j`

33. How to get the dates of yesterday, today and tomorrow? (☆☆☆)

```
today = np.datetime64('today', 'D')
yesterday = np.datetime64('today', 'D') - np.timedelta64(1, 'D')
tomorrow = np.datetime64('today', 'D') + np.timedelta64(1, 'D')

print("Today is ", today)
print('Tomorrow is ', tomorrow)
print('Yesterday was ', yesterday)
```

Today is 2022-09-06

Tomorrow is 2022-09-07

Yesterday was 2022-09-05

34. How to get all the dates corresponding to the month of July 2016? (☆☆☆)

```
np.arange(np.datetime64('2016-07-01'), np.datetime64('2016-08-01'))
```

```
array(['2016-07-01', '2016-07-02', '2016-07-03', '2016-07-04',
```

```
'2016-07-05', '2016-07-06', '2016-07-07', '2016-07-08',
'2016-07-09', '2016-07-10', '2016-07-11', '2016-07-12',
'2016-07-13', '2016-07-14', '2016-07-15', '2016-07-16',
'2016-07-17', '2016-07-18', '2016-07-19', '2016-07-20',
'2016-07-21', '2016-07-22', '2016-07-23', '2016-07-24',
'2016-07-25', '2016-07-26', '2016-07-27', '2016-07-28',
'2016-07-29', '2016-07-30', '2016-07-31'], dtype='datetime64[D]')
```

35. How to compute $((A+B)*(-A/2))$ in place (without copy)? (★★☆)

```
a = np.random.random((3,3))
b = np.random.random((3,3))

c = a
print("Before the equation, c: \n", c)
np.multiply(np.multiply(np.add(a, b, out=a), a, out=a), 0.5, out=a)
print("After the equation, c: \n", c)
```

Before the equation, c:

```
[[0.54199369 0.73523583 0.1582846 ]
 [0.74767132 0.68661553 0.28255951]
 [0.06694817 0.26148869 0.19346578]]
```

After the equation, c:

```
[[0.85803092 0.70460062 0.59382685]
 [0.89784795 0.50576909 0.09696205]
 [0.52071283 0.34422559 0.40109776]]
```

36. Extract the integer part of a random array of positive numbers using 4 different methods (★★☆)

```
best_array = np.random.random((3, 3))
print("best_array OG: \n", best_array)
print("np.round: \n", np.round(best_array))
print('np.ceil: \n', np.ceil(best_array))
print('np.floor: \n', np.floor(best_array))
print('dtype = int: \n', np.array(best_array, dtype = int))
```

best_array OG:

```
[[0.66221418 0.73239889 0.52296533]
 [0.20069796 0.05912904 0.11679739]
 [0.31273546 0.54802996 0.08372067]]
```

np.round:

```
[[1. 1. 1.]
 [0. 0. 0.]
 [0. 1. 0.]]
```

np.ceil:

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
np.floor:
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
dtype = int:
[[0 0 0]
 [0 0 0]
 [0 0 0]]
```

37. Create a 5×5 matrix with row values ranging from 0 to 4 (★★☆)

```
more_fives = np.zeros((5, 5))
print("Array before: \n", more_fives)

more_fives += np.arange(5)
print("Array after: \n", more_fives)
```

Array before:

```
[[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.]]
```

Array after:

```
[[0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]
 [0. 1. 2. 3. 4.]]
```

38. Consider a generator function that generates 10 integers and use it to build an array (★★☆)

```
def number_maker():
    for number in range(11):
        yield number

generated_array = np.fromiter(number_maker(), dtype=int)

print("Tada! An generated array! \n", generated_array)
```

Tada! An generated array!

```
[ 0  1  2  3  4  5  6  7  8  9 10]
```

39. Create a vector of size 10 with values ranging from 0 to 1, both excluded (★★☆)

```
skinny_array = np.linspace(0,1,12)[1:-1]
print(skinny_array)
```

```
[0.09090909 0.18181818 0.27272727 0.36363636 0.45454545 0.54545455
 0.63636364 0.72727273 0.81818182 0.90909091]
```

40. Create a random vector of size 10 and sort it (★★☆)

```
rando_dando = np.random.random(10)
rando_dando.sort()
print(rando_dando)
```

```
[0.02403261 0.50761808 0.5484412  0.58432952 0.68199952 0.7497707
 0.76654717 0.88180687 0.89915204 0.99256125]
```

Save your progress by committing your work to Jovian

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/numpy-100-exercises" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/numpy-100-exercises
'https://jovian.ai/evanmarie/numpy-100-exercises'
```

41. How to sum a small array faster than np.sum? (★★☆)

```
%%time
sum_me = np.arange(15)
sum = np.add.reduce(sum_me)
```

```
CPU times: user 58 µs, sys: 14 µs, total: 72 µs
Wall time: 51.7 µs
```

```
%%time
sum_me = np.arange(15)
sum = np.sum(sum_me)
```

```
CPU times: user 84 µs, sys: 20 µs, total: 104 µs
Wall time: 73 µs
```

42. Consider two random array A and B, check if they are equal (★★☆)

```
%%time
random_a_array = np.random.random((3, 3))
random_b_array = np.random.random((3, 3))
```

```
print("It is very unlikely this will be True. It is ", np.all(random_a_array == random_
```

It is very unlikely this will be True. It is False
CPU times: user 422 µs, sys: 102 µs, total: 524 µs
Wall time: 325 µs

```
%%time  
random_a_array = np.random.random((3, 3))  
random_b_array = np.random.random((3, 3))  
  
print("It is very unlikely this will be True. It is ", np.array_equal(random_a_array, r
```

It is very unlikely this will be True. It is False
CPU times: user 647 µs, sys: 157 µs, total: 804 µs
Wall time: 478 µs

43. Make an array immutable (read-only) (★★☆)

```
immute_me = np.random.random((5, 5))  
  
immute_me.flags.writeable = False  
# immute_me[1] = 5  
# This produces a ValueError
```

44. Consider a random 10×2 matrix representing cartesian coordinates, convert them to polar coordinates (★★☆)

```
cartesians = np.random.random((10, 2))  
  
x,y = cartesians.T  
radius = np.sqrt(np.sum(cartesians*cartesians, 1))  
theta = np.arctan(y, x)  
theta[x<0] += np.pi  
  
polars = np.vstack((radius, theta)).T  
print("Polars: \n", polars)
```

Polars:

```
[[1.27515662 0.73976411]  
 [0.54925144 0.39801951]  
 [0.23653274 0.01029341]  
 [0.27164798 0.09496444]  
 [0.36780169 0.05331711]  
 [1.03308727 0.69409103]  
 [1.01639692 0.5122008 ]  
 [1.24331596 0.64220638]
```

```
[0.9171994 0.61102944]
[0.92459774 0.74527171]]
```

45. Create random vector of size 10 and replace the maximum value by 0 (★★☆)

```
limit_my_max = np.random.randn(10)
print("Before: ", limit_my_max)
limit_my_max[limit_my_max.argmax()] = 0
print("After: ", limit_my_max)
```

```
Before: [-2.00091912  0.53256148 -1.30533324  0.49131961  0.38876112  0.68220756
 1.11240392 -0.91431662 -0.66670342  1.13805664]
After: [-2.00091912  0.53256148 -1.30533324  0.49131961  0.38876112  0.68220756
 1.11240392 -0.91431662 -0.66670342  0.          ]
```

46. Create a structured array with x and y coordinates covering the [0,1]x[0,1] area (★★☆)

```
meshy_array = np.zeros((5,5), [("x", float), ("y", float)])
meshy_array['x'], meshy_array['y'] = np.meshgrid(np.linspace(0,1,5), np.linspace(0,1,5))
print("Meshy Coordinates: \n", meshy_array['x'], meshy_array['y'])
```

Meshy Coordinates:

```
[[0.  0.25 0.5  0.75 1.  ]
 [0.  0.25 0.5  0.75 1.  ]
 [0.  0.25 0.5  0.75 1.  ]
 [0.  0.25 0.5  0.75 1.  ]
 [0.  0.25 0.5  0.75 1.  ]] [[0.  0.  0.  0.  0.  ]
 [0.25 0.25 0.25 0.25 0.25]
 [0.5  0.5  0.5  0.5  0.5  ]
 [0.75 0.75 0.75 0.75 0.75]
 [1.  1.  1.  1.  1.  ]]
```

47. Given two arrays, X and Y, construct the Cauchy matrix C ($C_{ij} = 1/(x_i - y_j)$)

```
x_boy = np.arange(5, 25, 2)
y_girl = np.arange(25, 45, 2).reshape(-1,1)

cauchy = 1.0 / (x_boy - y_girl)

print("My Cauchy: \n", cauchy)
```

My Cauchy:

```
[[ -0.05          -0.05555556 -0.0625          -0.07142857 -0.08333333 -0.1
  -0.125          -0.16666667 -0.25           -0.5           ]
 [-0.04545455 -0.05          -0.05555556 -0.0625          -0.07142857 -0.08333333
```

```

-0.1      -0.125      -0.16666667 -0.25      ]
[-0.04166667 -0.04545455 -0.05      -0.05555556 -0.0625      -0.07142857
-0.08333333 -0.1      -0.125      -0.16666667]
[-0.03846154 -0.04166667 -0.04545455 -0.05      -0.05555556 -0.0625
-0.07142857 -0.08333333 -0.1      -0.125      ]
[-0.03571429 -0.03846154 -0.04166667 -0.04545455 -0.05      -0.05555556
-0.0625      -0.07142857 -0.08333333 -0.1      ]
[-0.03333333 -0.03571429 -0.03846154 -0.04166667 -0.04545455 -0.05
-0.05555556 -0.0625      -0.07142857 -0.08333333]
[-0.03125     -0.03333333 -0.03571429 -0.03846154 -0.04166667 -0.04545455
-0.05      -0.05555556 -0.0625      -0.07142857]
[-0.02941176 -0.03125     -0.03333333 -0.03571429 -0.03846154 -0.04166667
-0.04545455 -0.05      -0.05555556 -0.0625      ]
[-0.02777778 -0.02941176 -0.03125     -0.03333333 -0.03571429 -0.03846154
-0.04166667 -0.04545455 -0.05      -0.05555556]
[-0.02631579 -0.02777778 -0.02941176 -0.03125     -0.03333333 -0.03571429
-0.03846154 -0.04166667 -0.04545455 -0.05      ]]

```

48. Print the minimum and maximum representable value for each numpy scalar type (★★☆)

```

for dtype in [np.int8, np.int32, np.int64]:
    print("np.iinfo(dtype): \n", np.iinfo(dtype))

```

```
np.iinfo(dtype):
```

```
Machine parameters for int8
```

```
-----
min = -128
```

```
max = 127
-----
```

```
np.iinfo(dtype):
```

```
Machine parameters for int32
```

```
-----
min = -2147483648
```

```
max = 2147483647
-----
```

```
np.iinfo(dtype):
```

```
Machine parameters for int64
```

```
-----
min = -9223372036854775808
```

```
max = 9223372036854775807
```

49. How to print all the values of an array? (★★☆)

```
print_me = np.arange(3, 33, 3)
np.set_printoptions(threshold=np.inf)

print(print_me)
```

```
[ 3  6  9 12 15 18 21 24 27 30]
```

50. How to find the closest value (to a given scalar) in a vector? (★★☆)

```
fred = np.arange(100)
albert = np.random.uniform(0, 100)
chelsea = np.abs(fred - albert).argmin()

print("Voila! ", fred[chelsea])
```

```
Voila!  2
```

Save and commit your work.

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/numpy-100-exercises" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/numpy-100-exercises
'https://jovian.ai/evanmarie/numpy-100-exercises'
```

51. Create a structured array representing a position (x,y) and a color (r,g,b) (★★☆)

```
data = np.zeros(3, dtype=[('position', [ ('x', float),
                                          ('y', float)]),
                          ('color',    [ ('r', float),
                                          ('g', float),
                                          ('b', float)])])

print("Structured array with x,y and a color: \n", data)
```

Structured array with x,y and a color:

```
[((0., 0.), (0., 0., 0.)) ((0., 0.), (0., 0., 0.))
((0., 0.), (0., 0., 0.))]
```

52. Consider a random vector with shape (100,2) representing coordinates, find point by point distances (★★☆)

```

# Use np.atleast_2d to convert both vectors to 2D
# Use broadcasting to subtract vector_2 from vector_1
# Use np.square to square the difference matrix
# Use np.sum to sum the squared difference matrix
# Use np.sqrt to take the square root of the sum
# Use np.squeeze to remove the last dimension

vector_1 = np.random.random((100, 2))
vector_2 = np.random.random((100, 2))
np.atleast_2d(vector_1, vector_2)

distances = np.sqrt(np.sum(np.square(vector_1 - vector_2), axis = 1))
print("distances: \n", distances)

```

distances:

```

[0.37500514 0.40194236 0.87408422 0.36504844 0.25690948 0.43830842
0.29931605 0.44108131 0.82064161 0.24875154 0.25418197 0.79636346
0.76734041 0.91839932 0.33154296 0.90060696 0.29553676 0.36634957
0.62841239 0.70250195 0.50243344 0.88653035 0.18417443 0.16241003
0.39532677 0.45180055 0.93805795 0.14490657 0.23748043 0.20604392
0.87313928 0.66615308 0.17292354 0.27489274 1.13487026 0.27018671
1.098745 0.8094849 0.94058422 0.84229834 0.16301062 0.78206235
0.77743234 0.30916985 0.47095811 0.54619457 0.60406344 0.54795786
0.18226784 0.43539416 0.83338138 0.8537773 0.97472522 0.35584677
0.43166337 0.08900545 0.46872347 0.71792729 0.79601286 0.19556448
0.98615029 0.73866868 0.31051881 1.02478305 0.60000085 0.45677661
0.46391797 0.61743916 0.57997856 0.6898347 0.32225085 0.18407733
0.86593103 0.44238151 0.20921601 0.4421152 0.58563656 0.79281233
0.10533349 0.61292223 0.49194887 1.18678628 0.05927907 0.77785762
0.86884598 0.93357024 0.58938209 0.39165922 0.61747163 0.45668966
0.89154288 0.78834916 0.87454739 0.85549846 0.65671255 0.89349077
0.36937932 0.91841937 0.24283276 1.11379855]

```

53. How to convert a float (32 bits) array into an integer (32 bits) in place?

```

# Use the astype method to convert the data type of the array

floats_away = np.arange(10, dtype = np.float32)
floats_converted = floats_away.astype(np.int32, copy = False)
print("floats_converted: \n", floats_converted)

```

floats_converted:

```
[0 1 2 3 4 5 6 7 8 9]
```

54. How to read the following file? (☆☆)

```

1, 2, 3, 4, 5
6, , , 7, 8

```

```
, , 9,10,11
```

```
from io import StringIO # StringIO behaves like a file object
```

```
string_to_read = StringIO('''
1, 2, 3, 4, 5
6, , , 7, 8
, , 9,10,11''')

numbers_to_read = np.genfromtxt(string_to_read, delimiter = ",", dtype = int)
print("numbers_to_read: \n", numbers_to_read)
```

```
numbers_to_read:
[[ 1  2  3  4  5]
 [ 6 -1 -1  7  8]
 [-1 -1  9 10 11]]
```

55. What is the equivalent of enumerate for numpy arrays? (★★☆)

```
np.ndenumerate
# Example:
np.ndenumerate(numbers_to_read)
# Creates a ndenumerate object that can be iterated over
```

```
<numpy.ndenumerate at 0x7f742c206e50>
```

56. Generate a generic 2D Gaussian-like array (★★☆)

```
# Use np.meshgrid to create a rectangular grid out of an array of x values and an array
# Use the standard equation for a 2D Gaussian to compute the height of each point on the
# Use np.exp to compute the exponential of all elements in the array
# Use np.sqrt to compute the square root of all elements in the array

x = np.linspace(-1, 1, 10)
y = np.linspace(-1, 1, 10)
x,y = np.meshgrid(x, y)
d = np.sqrt(x*x + y*y)
sigma, mu = 1.0, 0.0
g = np.exp(-((d-mu)**2 / (2.0 * sigma**2)))
print("g: \n", g)
```

```
g:
[[0.36787944 0.44822088 0.51979489 0.57375342 0.60279818 0.60279818
 0.57375342 0.51979489 0.44822088 0.36787944]
 [0.44822088 0.54610814 0.63331324 0.69905581 0.73444367 0.73444367
 0.69905581 0.63331324 0.54610814 0.44822088]
 [0.51979489 0.63331324 0.73444367 0.81068432 0.85172308 0.85172308
```

```

0.81068432 0.73444367 0.63331324 0.51979489]
[0.57375342 0.69905581 0.81068432 0.89483932 0.9401382 0.9401382
0.89483932 0.81068432 0.69905581 0.57375342]
[0.60279818 0.73444367 0.85172308 0.9401382 0.98773022 0.98773022
0.9401382 0.85172308 0.73444367 0.60279818]
[0.60279818 0.73444367 0.85172308 0.9401382 0.98773022 0.98773022
0.9401382 0.85172308 0.73444367 0.60279818]
[0.57375342 0.69905581 0.81068432 0.89483932 0.9401382 0.9401382
0.89483932 0.81068432 0.69905581 0.57375342]
[0.51979489 0.63331324 0.73444367 0.81068432 0.85172308 0.85172308
0.81068432 0.73444367 0.63331324 0.51979489]
[0.44822088 0.54610814 0.63331324 0.69905581 0.73444367 0.73444367
0.69905581 0.63331324 0.54610814 0.44822088]
[0.36787944 0.44822088 0.51979489 0.57375342 0.60279818 0.60279818
0.57375342 0.51979489 0.44822088 0.36787944]]

```

57. How to randomly place p elements in a 2D array? (★★☆)

```

# Use np.put to randomly place p elements in an array
# Use np.random.choice to generate p unique indices
# Use np.put to place p elements in the array at the indices generated by np.random.choice
# OR
# Use np.putmask to place p elements in the array at random locations

my_number = 3
my_random_array = np.zeros((5, 5))
np.put(my_random_array, np.random.choice(range(5*5), my_number, replace = False), 1)
print("my_random_array: \n", my_random_array)

# OR

my_random_array2 = np.zeros((5, 5))
np.putmask(my_random_array2, np.random.rand(5, 5) < 0.5, 1)
print("my_random_array2: \n", my_random_array2)

```

```

my_random_array:
[[0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0.]]

my_random_array2:
[[1. 0. 1. 1. 0.]
 [0. 1. 1. 1. 0.]
 [1. 0. 0. 1. 0.]]

```

```
[0. 0. 0. 1. 1.]
[0. 1. 0. 1. 1.]]
```

58. Subtract the mean of each row of a matrix (★★☆)

```
# Subtract the mean of each row of a matrix
# Use np.mean to compute the mean of the flattened array
# Use broadcasting to subtract the mean from each row

my_matrix = np.random.rand(5, 10)
my_matrix_mean = my_matrix - my_matrix.mean(axis = 1, keepdims = True)
print("my_matrix_mean: \n", my_matrix_mean)
```

```
my_matrix_mean:
[[-0.087331    0.28045704  0.09884103  0.24028911 -0.05410158 -0.54227666
 -0.44091135  0.09796806  0.36679186  0.0402735 ]
 [-0.33913262  0.46283841 -0.41976427 -0.44509772  0.1420593   0.15908923
 -0.19847702  0.43547772  0.44190862 -0.23890165]
 [ 0.06450739 -0.34022669 -0.30249776 -0.10469277  0.16732515 -0.25225626
 -0.05973743  0.29430727  0.19188914  0.34138195]
 [-0.31738591  0.41807973  0.26420105 -0.04011138  0.45053307 -0.19966895
 -0.37402804 -0.14936405  0.08661854 -0.13887405]
 [ 0.21240694  0.29108887  0.41998414 -0.35085964 -0.09182055 -0.18847607
 -0.48727223 -0.09486634  0.25774539  0.03206949]]
```

59. How to sort an array by the nth column? (★★☆)

```
def sort_me_baby(array, n):
    return array[np.argsort(array[:, n])]

array_to_sort = np.random.randint(0, 10, (5, 5))

print("sort_me_baby(array_to_sort, 3): \n", sort_me_baby(array_to_sort, 3))
```

```
sort_me_baby(array_to_sort, 3):
[[8 7 5 0 9]
 [6 2 6 4 7]
 [1 5 6 5 1]
 [2 0 9 8 8]
 [1 8 5 9 0]]
```

60. How to tell if a given 2D array has null columns? (★★☆)

```
my_array_to_investigate = np.random.randint(0, 10, (6, 10))

print("my_array_to_investigate: \n", my_array_to_investigate)
```

```
print('\n')
print("Array has null columns? \n",
      (my_array_to_investigate.all(axis = 0)).any())
print('\n')
```

```
my_array_to_investigate:
[[7 5 3 7 3 8 1 0 9 2]
 [1 0 3 9 0 8 9 0 6 5]
 [6 5 5 0 0 2 7 4 6 7]
 [4 2 6 7 4 7 8 8 6 3]
 [7 9 1 0 6 2 2 1 4 1]
 [6 0 5 8 4 6 5 0 1 9]]
```

```
Array has null columns?
True
```

Save and commit your work.

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
[jovian] Updating notebook "aakashns/numpy-100-exercises" on https://jovian.ml/
[jovian] Uploading notebook..
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ml/aakashns/numpy-100-exercises
'https://jovian.ml/aakashns/numpy-100-exercises'
```

61. Find the nearest value from a given value in an array (★★☆)

```
# Use np.abs to compute the absolute value of the difference between the value and each
# in the array
# Use np.argmax to return the indices of the minimum values along the given axis

my_array_to_search = np.random.randint(0, 10, 10)
search_value = 5
results = my_array_to_search[np.abs(my_array_to_search - search_value).argmin()]

print("my_array_to_search: \n", my_array_to_search)
print("results: \n", results)
```

```
my_array_to_search:
[0 0 3 8 9 6 0 9 8 9]
```

results:

6

62. Considering two arrays with shape (1,3) and (3,1), how to compute their sum using an iterator? (★★☆)

```
# Use np.nditer to iterate over the arrays

array_3_1 = np.arange(3).reshape(3, 1)
array_1_3 = np.arange(3).reshape(1, 3)
array_sum = np.nditer([array_3_1, array_1_3, None])
print("array_3_1: \n", array_3_1, "\n")
print("array_1_3: \n", array_1_3, "\n")
print("array_sum: \n", [array_sum.operands[2] for a, b, c in array_sum if c == a + b],
```

array_3_1:

```
[[0]
 [1]
 [2]]
```

array_1_3:

```
[[0 1 2]]
```

array_sum:

```
[]
```

63. Create an array class that has a name attribute (★★☆)

```
# Use a class to create an array with a name attribute
# Use the __array_finalize__ method to set the name attribute of the returned array

class MyArrayClass(np.ndarray):
    def __new__(cls, array, name = 'a_name'):
        object = np.asarray(array).view(cls)
        object.name = name
        return object

    def __array_finalize__(self, object):
        if object is None: return
        self.info = getattr(object, 'name', 'a_name')

a_classy_array = MyArrayClass(np.arange(10), name = 'a_classy_array')

print("a_classy_array: \n", a_classy_array, "\n")
```

a_classy_array:

```
[0 1 2 3 4 5 6 7 8 9]
```

64. Consider a given vector, how to add 1 to each element indexed by a second vector (be careful with repeated indices)? (★★★)

```
# Use np.add.at to add 1 to each element of a, at indices in b

little_array_a = np.ones(10)
little_array_b = np.random.randint(0, len(little_array_a), 23)

print("little_array_a: \n", little_array_a, "\n")
print("little_array_b: \n", little_array_b, "\n")

np.add.at(little_array_a, little_array_b, 1)

print("little_array_a: \n", little_array_a, "\n")
```

little_array_a:

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

little_array_b:

```
[8 6 4 5 1 1 0 9 7 0 0 3 1 1 5 1 6 4 0 1 9 5 6]
```

little_array_a:

```
[5. 7. 1. 2. 3. 4. 4. 2. 2. 3.]
```

65. How to accumulate elements of a vector (X) to an array (F) based on an index list (I)? (★★★)

```
# Use np.bincount to count the number of occurrences of each value in an array of non-r
# ints

x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
i = [7, 4, 7, 1, 2, 3, 9, 3, 9, 3]
f = np.bincount(i, weights = x)

print("x: \n", x, "\n")
print("i: \n", i, "\n")
print("f: \n", f, "\n")
```

x:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

i:

```
[7, 4, 7, 1, 2, 3, 9, 3, 9, 3]
```

f:

[0. 4. 5. 24. 2. 0. 0. 4. 0. 16.]

66. Considering a (w,h,3) image of (dtype=ubyte), compute the number of unique colors (★★★)

67. Considering a four dimensions array, how to get sum over the last two axis at once? (★★★)

68. Considering a one-dimensional vector D, how to compute means of subsets of D using a vector S of same size describing subset indices? (★★★)

69. How to get the diagonal of a dot product? (★★★)

70. Consider the vector [1, 2, 3, 4, 5], how to build a new vector with 3 consecutive zeros interleaved between each value? (★★★)

Save and commit your work

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

```
[jovian] Updating notebook "aakashns/numpy-100-exercises" on https://jovian.ml/
```

```
[jovian] Uploading notebook..
```

```
[jovian] Capturing environment..
```

```
[jovian] Committed successfully! https://jovian.ml/aakashns/numpy-100-exercises
```

```
'https://jovian.ml/aakashns/numpy-100-exercises'
```

71. Consider an array of dimension (5,5,3), how to multiply it by an array with dimensions (5,5)? (★★★)

72. How to swap two rows of an array? (★★★)

73. Consider a set of 10 triplets describing 10 triangles (with shared vertices), find the set of unique line segments composing all the triangles (★★★)

74. Given an array C that is a bincount, how to produce an array A such that `np.bincount(A) == C`? (★★★)

75. How to compute averages using a sliding window over an array? (★★★)

76. Consider a one-dimensional array Z, build a two-dimensional array whose first row is `(Z[0],Z[1],Z[2])` and each subsequent row is shifted by 1 (last row should be `(Z[-3],Z[-2],Z[-1])`) (★★★)

77. How to negate a boolean, or to change the sign of a float inplace? (★★★)

78. Consider 2 sets of points P0,P1 describing lines (2d) and a point p, how to compute distance from p to each line i `(P0[i],P1[i])`? (★★★)

79. Consider 2 sets of points P0,P1 describing lines (2d) and a set of points P, how to compute distance from each point j `(P[j])` to each line i `(P0[i],P1[i])`? (★★★)

80. Consider an arbitrary array, write a function that extract a subpart with a fixed shape and centered on a given element (pad with a `fill` value when necessary) (★★★)

Save and commit your work.

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

```
[jovian] Updating notebook "aakashns/numpy-100-exercises" on https://jovian.ml/
```

```
[jovian] Uploading notebook..
```

```
[jovian] Capturing environment..
```

```
[jovian] Committed successfully! https://jovian.ml/aakashns/numpy-100-exercises
```

```
'https://jovian.ml/aakashns/numpy-100-exercises'
```

81. Consider an array $Z = [1,2,3,4,5,6,7,8,9,10,11,12,13,14]$, how to generate an array $R = [[1,2,3,4], [2,3,4,5], [3,4,5,6], \dots, [11,12,13,14]]$? (★★★)

82. Compute a matrix rank (★★★)

83. How to find the most frequent value in an array?

84. Extract all the contiguous 3×3 blocks from a random 10×10 matrix (★★★)

85. Create a 2D array subclass such that $Z[i,j] == Z[j,i]$ (★★★)

86. Consider a set of p matrices with shape (n,n) and a set of p vectors with shape $(n,1)$. How to compute the sum of the p matrix products at once? (result has shape $(n,1)$) (★★★)

87. Consider a 16×16 array, how to get the block-sum (block size is 4×4)? (★★★)

88. How to implement the Game of Life using numpy arrays? (★★★)

89. How to get the n largest values of an array (★★★)

90. Given an arbitrary number of vectors, build the cartesian product (every combinations of every item) (★★★)

Save and commit your work

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

```
[jovian] Updating notebook "aakashns/numpy-100-exercises" on https://jovian.ml/
```

```
[jovian] Uploading notebook..
```

```
[jovian] Capturing environment..
```

```
[jovian] Committed successfully! https://jovian.ml/aakashns/numpy-100-exercises
```

```
'https://jovian.ml/aakashns/numpy-100-exercises'
```

91. How to create a record array from a regular array? (★★★)

92. Consider a large vector Z, compute Z to the power of 3 using 3 different methods (★★★)

93. Consider two arrays A and B of shape (8,3) and (2,2). How to find rows of A that contain elements of each row of B regardless of the order of the elements in B? (★★★)

94. Considering a 10×3 matrix, extract rows with unequal values (e.g. [2,2,3]) (★★★)

95. Convert a vector of ints into a matrix binary representation (★★★)

96. Given a two dimensional array, how to extract unique rows? (★★★)

97. Considering 2 vectors A & B, write the einsum equivalent of inner, outer, sum, and mul function (★★★)

98. Considering a path described by two vectors (X,Y), how to sample it using equidistant samples (★★★)?

99. Given an integer n and a 2D array X, select from X the rows which can be interpreted as draws from a multinomial distribution with n degrees, i.e., the rows which only contain integers and which sum to n. (★★★)

100. Compute bootstrapped 95% confidence intervals for the mean of a 1D array X (i.e., resample the elements of an array with replacement N times, compute the mean of each sample, and then compute percentiles over the means). (★★★)

Save and commit your work

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
```

Congratulations on completing the 100 exercises, well done!

What to do next?

- Share your completed notebook on Facebook, LinkedIn or Twitter and challenge your friends.

- Share your solutions and help others on the forum: <https://jovian.ml/forum/t/100-numpy-exercises-hints-discussions-help/10561>
- Check out our course on "Data Analysis with Python: Zero to Pandas" - <https://jovian.ml/learn/data-analysis-with-python-zero-to-pandas>
- Star this repository to show your appreciation for the original author of this notebook: <https://github.com/rougier/numpy-100>

Hey, Numpy, You're So Fine!

So many arrays, so little... STOP! Numpy Time!

(Too many song references?)

1920px-NumPy_logo_2020.svg.png

NumPy stands for Numerical Python and is a library for Python created for working with arrays and performing a wide variety of numerical computations. Numpy is centered around matrices and vectors and all the many mathematical calculations, manipulations, and many intriguing operations that can be performed on them. (I get excited about numbers, can you tell?!

Numpy is open source and quite invaluable to the data science community. It is a very integral library in the fields of Machine Learning, Data Science and Analysis, and many more.

These are some of the functions I found interesting when working on the 100 Numpy exercises, which helped me a great deal in this assignment:

- np.tile
- np.intersect1d
- np.fromiter
- np.add.reduce
- np.ndenumerate

```
!pip install jovian --upgrade -q
```

```
import jovian
```

```
jovian.commit(project='numpy-array-operations')
```

```
[jovian] Updating notebook "evanmarie/numpy-array-operations" on https://jovian.ai  
[jovian] Committed successfully! https://jovian.ai/evanmarie/numpy-array-operations  
'https://jovian.ai/evanmarie/numpy-array-operations'
```

Let's begin by importing Numpy and listing out the functions covered in this notebook.

```
import numpy as np
```

Let's get into those functions! ✓

- ▷ function1 = np.tile
- ▷ function2 = np.intersect1d
- ▷ function3 = np.fromiter
- ▷ function4 = np.add.reduce, np.multiply.reduce, etc.
- ▷ function5 = np.ndenumerate

► Function 1 - np.tile

- ▷ np.tile is a function that repeats an array a given number of times, along each axis.
- ▷ The syntax is: np.tile(A, reps)
- ▷ A is the array to be repeated, and reps is the number of times to repeat the array.
- ▷ The output is an array with the same shape as A, but with each axis repeated reps times.
- ▷ np.tile takes a single argument, reps, which can be an integer or a tuple of integers.

```
# Example 1: Repeat a 1D array
```

```
baby_array = np.arange(9).reshape(3, 3)
print("baby_array: \n", baby_array, "\n")

octuplets = np.tile(baby_array, (2, 2, 2))
print("octuplets: \n", octuplets, "\n")
```

baby_array:

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

octuplets:

```
[[[0 1 2 0 1 2]
 [3 4 5 3 4 5]
 [6 7 8 6 7 8]
 [0 1 2 0 1 2]
 [3 4 5 3 4 5]
 [6 7 8 6 7 8]]
```

```
[[[0 1 2 0 1 2]
 [3 4 5 3 4 5]
 [6 7 8 6 7 8]
 [0 1 2 0 1 2]
 [3 4 5 3 4 5]
 [6 7 8 6 7 8]]]
```

↑↑ Here, I used np.tile to copy baby_array over 8 times. I can see how incredibly useful this function can be in manipulating and working with data.

```
# Example 2: Repeat a 2D array
```

```
big_sis_array = np.arange(16).reshape(4, 4)
print("big_sis_array: \n", big_sis_array, "\n")

too_many_sisters = np.tile(big_sis_array, (4, 4, 4))
print("too_many_sisters: \n", too_many_sisters, "\n")
```

big_sis_array:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
[12 13 14 15]]
```

too_many_sisters:

```
[[[ 0  1  2 ...  1  2  3]
 [ 4  5  6 ...  5  6  7]
 [ 8  9 10 ...  9 10 11]
 ...
 [ 4  5  6 ...  5  6  7]
 [ 8  9 10 ...  9 10 11]
 [12 13 14 ... 13 14 15]]]
```

```
[[ 0  1  2 ...  1  2  3]
 [ 4  5  6 ...  5  6  7]
 [ 8  9 10 ...  9 10 11]
 ...
 [ 4  5  6 ...  5  6  7]
 [ 8  9 10 ...  9 10 11]
 [12 13 14 ... 13 14 15]]]
```

```
[[ 0  1  2 ...  1  2  3]
 [ 4  5  6 ...  5  6  7]
 [ 8  9 10 ...  9 10 11]
 ...
 [ 4  5  6 ...  5  6  7]
 [ 8  9 10 ...  9 10 11]
 [12 13 14 ... 13 14 15]]]
```

```
[[ 0  1  2 ...  1  2  3]
 [ 4  5  6 ...  5  6  7]
 [ 8  9 10 ...  9 10 11]
 ...
 [ 4  5  6 ...  5  6  7]
 [ 8  9 10 ...  9 10 11]
 [12 13 14 ... 13 14 15]]]
```

↑↑ Here, I took a larger array and created an even more exquisite fractal expression of it with `np.tile()`.

```
# Example 3: epic failure
```

```
a_sad_array = np.arange(8).reshape(2, 2, 2)
print("a_sad_array: \n", a_sad_array, "\n")

epic_fail = np.tile(a_sad_array, (2, -1, 2))
print("epic_fail: \n", epic_fail, "\n")

# OH NO! What happened?
# ValueError: negative dimensions are not allowed - epic fail!
```

a_sad_array:

```
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

```
-----
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_38/3095199431.py in <module>
      4 print("a_sad_array: \n", a_sad_array, "\n")
      5
----> 6 epic_fail = np.tile(a_sad_array, (2, -1, 2))
      7 print("epic_fail: \n", epic_fail, "\n")
      8

<__array_function__ internals> in tile(*args, **kwargs)

/opt/conda/lib/python3.9/site-packages/numpy/lib/shape_base.py in tile(A, reps)
    1256     for dim_in, nrep in zip(c.shape, tup):
    1257         if nrep != 1:
-> 1258             c = c.reshape(-1, n).repeat(nrep, 0)
    1259             n //= dim_in
    1260     return c.reshape(shape_out)
```

ValueError: negative dimensions are not allowed

↑↑ The **ValueError** is thrown, because `.tile()` does not know what to do with a negative integer as a dimension. This can be remedied by simply never giving `.tile()` a negative dimension.

↑↑ **SUMMARY**: The `.tile()` function is definitely useful when there is data that must be replicated in multiple directions.

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/numpy-array-operations" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/numpy-array-operations
'https://jovian.ai/evanmarie/numpy-array-operations'
```

► Function 2 - np.intersect1d

- ▷ np.intersect1d is a function that finds the intersection of two arrays.
- ▷ The syntax is: np.intersect1d(a, b)
- ▷ a and b are the arrays to be intersected.
- ▷ The output is an array containing the elements common to both a and b.

```
# Example 1: Find the intersection of two 1D arrays
```

```
array_eins = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])  
array_zwei = np.arange(8, 16, 2)
```

```
print("array_eins: \n", array_eins, "\n")  
print("array_zwei: \n", array_zwei, "\n")
```

```
crossroads = np.intersect1d(array_eins, array_zwei)  
print("crossroads: \n", crossroads, "\n")
```

```
array_eins:
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16]
```

```
array_zwei:
```

```
[ 8 10 12 14]
```

```
crossroads:
```

```
[ 8 10 12 14]
```

↑↑ Here, np.intersect1d() found the items that were identical between the two arrays it was given.

```
# Example 2: Something a bit more complicated
```

```
array_drei = np.arange(0, 65, 4)  
array_vier = np.arange(24, 88, 2)
```

```
print("array_drei: \n", array_drei, "\n")  
print("array_vier: \n", array_vier, "\n")
```

```
crossroads2 = np.intersect1d(array_drei, array_vier)  
print("crossroads2: \n", crossroads2, "\n")
```

```
array_drei:
```

```
[ 0  4  8 12 16 20 24 28 32 36 40 44 48 52 56 60 64]
```

```
array_vier:
```

```
[24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70  
72 74 76 78 80 82 84 86]
```

crossroads2:

```
[24 28 32 36 40 44 48 52 56 60 64]
```

↑↑ In this example, `np.intersect1d()` found the common items between larger pair of data.

```
# Example 3: Time for more failure!

a_sad_little_array = np.arange(8).reshape(2, 2, 2)
a_lonely_array = np.arange(9).reshape(3, 3)

print("a_sad_little_array: \n", a_sad_little_array, "\n")
print("a_lonely_array: \n", a_lonely_array, "\n")

lonely = np.intersect1d(a_sad_little_array, a_lonely_array, no_arrays=True)
print("lonely: \n", lonely, "\n")

# OH NO! What happened?
# TypeError: _intersect1d_dispatcher() got an unexpected keyword argument 'no_arrays'
# epic fail! intersect1d takes only two arguments.
```

a_sad_little_array:

```
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

a_lonely_array:

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

TypeError

Traceback (most recent call last)

/tmp/ipykernel_38/2869950181.py in <module>

```
7 print("a_lonely_array: \n", a_lonely_array, "\n")
```

```
8
```

```
----> 9 lonely = np.intersect1d(a_sad_little_array, a_lonely_array, no_arrays=True)
```

```
10 print("lonely: \n", lonely, "\n")
```

```
11
```

```
<__array_function__ internals> in intersect1d(*args, **kwargs)
```

TypeError: `_intersect1d_dispatcher()` got an unexpected keyword argument 'no_arrays'

↑↑ It is not easy to make `np.intersect1d()` fail, but if you pass it three arguments, it sure will! This is remedied by using as directed. [?](#)

SUMMARY: The `np.intersect1d()` function is a very useful tool, especially for finding data similarities, duplicates, general large-scale searches for common data, etc.

```
jovian.commit()
```

► Function 3 - `np.fromiter`

- ▷ `np.fromiter` is a function that creates a new 1D array from an iterable object.
- ▷ The syntax is: `np.fromiter(iterable, dtype, count=-1)`
- ▷ `iterable` is the iterable object to be converted to an array.
- ▷ `dtype` is the data type of the output array.
- ▷ `count` is the number of items to be read from iterable.
- ▷ The default count is -1, which means all data is read.

```
# Example 1: Create a 1D array from a list
```

```
a_beautiful_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("a_beautiful_list: \n", a_beautiful_list, "\n")

a_beautiful_array = np.fromiter(a_beautiful_list, dtype=int, count = 3)
print("a_beautiful_array: \n", a_beautiful_array, "\n")
```

```
a_beautiful_list:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
a_beautiful_array:
[1 2 3]
```

↑↑ Here, I used `np.fromiter()` to iterate over a list of integers 3 times. It is nothing fancy in this example, but its usefulness on a larger scale can be easily imagined.

```
# Example 2: Create a 1D array from a string
```

```
a_lovely_string = "Hey, World! It is time we had a little talk."
print("a_lovely_string: \n", a_lovely_string, "\n")

a_lovely_array = np.fromiter(a_lovely_string, dtype='S1')
print("a_lovely_array: \n", a_lovely_array, "\n")
```

```
a_lovely_string:
Hey, World! It is time we had a little talk.
```

```
a_lovely_array:
[b'H' b'e' b'y' b', ' b' ' b'W' b'o' b'r' b'l' b'd' b'!' b' ' b'I' b't'
 b' ' b'i' b's' b' ' b't' b'i' b'm' b'e' b' ' b'w' b'e' b' ' b'h' b'a'
 b'd' b' ' b'a' b' ' b'l' b'i' b't' b't' b'l' b'e' b' ' b't' b'a' b'l'
```

```
b'k' b'.']
```

↑↑ This time, I used `np.fromiter()` on a string. At first, I thought it was silly, but I wanted still wanted to try it out. Then I saw the output, and it felt even sillier than it originally had. `.fromiter()` gave me back the binary information from my original string stored as a densely packed array of bytes.

```
# Example 3: Who is ready to fail! It is easy with this one!!

unsure_dictionary = {1: "why", 2: "where", 3: "when", 4: "who", 5: "how"}
print("unsure_dictionary: \n", unsure_dictionary, "\n")

unsure_array = np.fromiter(unsure_dictionary, dtype=str, count=0)
print("unsure_array: \n", unsure_array, "\n")
# NameError: name 'str' is not defined AND
# ValueError: Must specify length when using variable-size data-type.
```

```
unsure_dictionary:
```

```
{1: 'why', 2: 'where', 3: 'when', 4: 'who', 5: 'how'}
```

```
-----
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_38/3415108785.py in <module>
      4 print("unsure_dictionary: \n", unsure_dictionary, "\n")
      5
----> 6 unsure_array = np.fromiter(unsure_dictionary, dtype='str', count=0)
      7 print("unsure_array: \n", unsure_array, "\n")
      8 # NameError: name 'str' is not defined AND
```

ValueError: Must specify length when using variable-size data-type.

↑↑ Here, I really gave `.fromiter()` a hard time and threw two errors. One was because I put the data type as 'str', as we often refer to strings. But it definitely did not know what I was asking for. This can be remedied by being sure to know the datatypes it utilizes and how to indicate them.

↑↑ I also caused a value error by asking for a count of 0, which it seems to be interpreting as a default value and simply not comprehending my pathetic communication. Poor, `.fromiter()`.

↑↑ **SUMMARY:** The `.fromiter()` function takes an iterable object and turns it into a one-dimensional array. So it would be very useful in the process of creating one-dimensional arrays from other data to work with.

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/numpy-array-operations" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/numpy-array-operations
'https://jovian.ai/evanmarie/numpy-array-operations'
```

► Function 4 - `np.add.reduce`

- ▷ these are all functions that reduce an array along a given axis using a given operation.
- ▷ The syntax is: `np.add.reduce(a, axis=None, dtype=None, out=None, keepdims=False)`
- ▷ `a` is the array to be reduced.
- ▷ `axis` is the axis along which the reduction is to be performed. The default is `None`.
- ▷ `dtype` is the data type of the output array. The default is `None`.
- ▷ `out` is the array in which to place the output. The default is `None`.
- ▷ `keepdims` is a boolean value
- ▷ `keepdims` determines whether the reduced axis is left in the result as a dimension with size one.
- ▷ The default `keepdims` is `False`.

```
# Example 1: Reduce an array along the first axis

add_to_me = np.arange(27).reshape(3, 3, 3)
print("Before .add.reduce(): \n", add_to_me, "\n")

added_to_it = np.add.reduce(add_to_me, axis = 1)
print("After .add.reduce(): \n", added_to_it, "\n")
```

Before `.add.reduce()`:

```
[[[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]]

 [[ 9 10 11]
  [12 13 14]
  [15 16 17]]

 [[18 19 20]
  [21 22 23]
  [24 25 26]]]
```

After `.add.reduce()`:

```
[[ 9 12 15]
 [36 39 42]
 [63 66 69]]
```

↑↑ **SUMMARY:** With `np.add.reduce()`, it initially seems like there is little to no point for this function to exist, considering there is `np.sum()`, via the 100 Numpy exercise assignment, I learned that `.add.reduce` is faster, as it is what `.sum()` calls in the when `.sum()` itself is called.

↑↑ I think what I appreciate most about this function is honestly the name. While it may sound odd, the idea of REDUCING 3D down, as in this example, somehow helped me to visualize 3D arrays better due to a EUREKA moment I had when learning about this function. That has crossed over into helping me understand much more. So this one might just be a personal thing for me.

```
# Example 2: using multiply.reduce to multiply all the elements of an array
```

```
multi_me = np.arange(256).reshape(4, 4, 4, 4)
print("Before .multiply.reduce(): \n", multi_me, "\n")

multied_you = np.multiply.reduce(multi_me, axis = 0)
print("After .multiply.reduce(): \n", multied_you, "\n")
```

Before .multiply.reduce():

```
[[[[ 0  1  2  3]
   [ 4  5  6  7]
   [ 8  9 10 11]
   [12 13 14 15]]

  [[ 16 17 18 19]
   [ 20 21 22 23]
   [ 24 25 26 27]
   [ 28 29 30 31]]

  [[ 32 33 34 35]
   [ 36 37 38 39]
   [ 40 41 42 43]
   [ 44 45 46 47]]

  [[ 48 49 50 51]
   [ 52 53 54 55]
   [ 56 57 58 59]
   [ 60 61 62 63]]]

[[[ 64 65 66 67]
   [ 68 69 70 71]
   [ 72 73 74 75]
   [ 76 77 78 79]]

  [[ 80 81 82 83]
   [ 84 85 86 87]
   [ 88 89 90 91]
   [ 92 93 94 95]]

  [[ 96 97 98 99]
   [100 101 102 103]
   [104 105 106 107]
   [108 109 110 111]]]
```

[[112 113 114 115]
[116 117 118 119]
[120 121 122 123]
[124 125 126 127]]]

[[[128 129 130 131]
[132 133 134 135]
[136 137 138 139]
[140 141 142 143]]]

[[144 145 146 147]
[148 149 150 151]
[152 153 154 155]
[156 157 158 159]]]

[[160 161 162 163]
[164 165 166 167]
[168 169 170 171]
[172 173 174 175]]]

[[176 177 178 179]
[180 181 182 183]
[184 185 186 187]
[188 189 190 191]]]

[[[192 193 194 195]
[196 197 198 199]
[200 201 202 203]
[204 205 206 207]]]

[[208 209 210 211]
[212 213 214 215]
[216 217 218 219]
[220 221 222 223]]]

[[224 225 226 227]
[228 229 230 231]
[232 233 234 235]
[236 237 238 239]]]

[[240 241 242 243]

```
[244 245 246 247]
[248 249 250 251]
[252 253 254 255]]]]
```

After `.multiply.reduce()`:

```
[[[ 0 1618305 3329040 5134545]
 [ 7037184 9039345 11143440 13351905]
 [ 15667200 18091809 20628240 23279025]
 [ 26046720 28933905 31943184 35077185]]

 [[ 38338560 41729985 45254160 48913809]
 [ 52711680 56650545 60733200 64962465]
 [ 69341184 73872225 78558480 83402865]
 [ 88408320 93577809 98914320 104420865]]

 [[110100480 115956225 121991184 128208465]
 [134611200 141202545 147985680 154963809]
 [162140160 169517985 177100560 184891185]
 [192893184 201109905 209544720 218201025]]

 [[227082240 236191809 245533200 255109905]
 [264925440 274983345 285287184 295840545]
 [306647040 317710305 329034000 340621809]
 [352477440 364604625 377007120 389688705]]]
```

↑↑ `.multiply.reduce()` Like `add.reduce()`, this is just going straight to the source of what `np.prod()` calls. So it is a bit faster. My favorite part about this one is seeing everything condensed down into this nicely arranged array. It is like number art! And I can also see how this could be very useful data analysis as well!

```
# Example 3: causing an error with .minimum.reduce
```

```
error_me = np.arange(4).reshape(2, 2)
error_you = np.arange(8).reshape(4, 2)
```

```
print("error_me: \n", error_me, "\n")
print("error_you: \n", error_you, "\n")
```

```
what_a_mess = np.minimum.reduce(error_me, error_you)
```

```
# TypeError: only integer scalar arrays can be converted to a scalar index
```

error_me:

```
[[0 1]
 [2 3]]
```

error_you:

```
[[0 1]
 [2 3]
 [4 5]
 [6 7]]
```

```
-----
TypeError                                Traceback (most recent call last)
/tmp/ipykernel_38/920115831.py in <module>
      7 print("error_you: \n", error_you, "\n")
      8
----> 9 what_a_mess = np.minimum.reduce(error_me, error_you)
     10 # TypeError: only integer scalar arrays can be converted to a scalar index
```

TypeError: only integer scalar arrays can be converted to a scalar index

↑↑ **SUMMARY:** In this example with `.minimum.reduce()`: the data was not vectorizable, meaning that the program did not know how to deal with the shapes I gave it simultaneously, as it were.

An explanation from stackoverflow.com does the issue better justice than I (the situation is not identical, but the same issues apply):

"...the result of each individual operation has a different shape: (3, 1), (3, 2) and (3, 3). They can not form the output of a single vectorized operation, because the output has to be one contiguous array. Of course, it can contain (3, 1), (3, 2) and (3, 3) arrays inside of it (as views), but that's what your original array already does."

source = Maxim:

<https://stackoverflow.com/questions/46902367/numpy-array-typeerror-only-integer-scalar-arrays-can-be-converted-to-a-scalar-i>

```
jovian.commit()
```

► np.ndenumerate

- `np.ndenumerate` returns an iterator yielding pairs of array coordinates and values.
- The syntax is: `np.ndenumerate(a, flags=None)`
- `a` is the array to be iterated over.
- `flags` is an optional parameter that can be used to specify the memory layout of the array.
- The default is `None`.

```
# Example 1: Iterate over a 2D array

a_2D_array = np.arange(12).reshape(3, 4)
print("a_2D_array: \n", a_2D_array, "\n")

for index, value in np.ndenumerate(a_2D_array):
    print(index, value)
```

↑↑ I chose `np.ndenumerate()`: because of its usefulness. I know that I would feel completely lost without `enumerate()` in Python, and likewise, now that I am getting to know Numpy quite well, I can see how useful this

function will be. It definitely will be a big console function for me just to keep track of things. And I know it will come in incredibly handy in all sorts of functions and operations.

```
# Example 2: Iterate over a 3D array

a_3D_array = np.arange(24).reshape(2, 3, 4)
print("a_3D_array: \n", a_3D_array, "\n")

for index, value in np.ndenumerate(a_3D_array):
    print(index, value)
```

a_3D_array:

```
[[[ 0  1  2  3]
   [ 4  5  6  7]
   [ 8  9 10 11]]
```

```
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]]
```

```
(0, 0, 0) 0
(0, 0, 1) 1
(0, 0, 2) 2
(0, 0, 3) 3
(0, 1, 0) 4
(0, 1, 1) 5
(0, 1, 2) 6
(0, 1, 3) 7
(0, 2, 0) 8
(0, 2, 1) 9
(0, 2, 2) 10
(0, 2, 3) 11
(1, 0, 0) 12
(1, 0, 1) 13
(1, 0, 2) 14
(1, 0, 3) 15
(1, 1, 0) 16
(1, 1, 1) 17
(1, 1, 2) 18
(1, 1, 3) 19
(1, 2, 0) 20
(1, 2, 1) 21
(1, 2, 2) 22
(1, 2, 3) 23
```

↑↑ As arrays become more dense and involved, I think `.ndenumerate()` becomes increasingly useful just to grab a quick address of an element. It also is very useful in visualizing the data being worked with.

```
# Example 3: Cause an error with np.ndenumerate

i_am_an_array = np.arange(12).reshape(3, 4)
print("i_am_an_array: \n", i_am_an_array, "\n")

i_am_not_iterable = 32
print("i_am_not_iterable: \n", i_am_not_iterable, "\n")

for index, value in np.ndenumerate(i_am_not_iterable):
    print(value, "\n")

# Well, this is embarrassing! That was not supposed to work!
print("That was surprising...Let's try again...")

# There must be a way to break this...
print("I feel like such a lowlife having to stoop to this level,")
print("but denumerate leaves me no choice...\n")

print("Gods, forgive me...I will throw an error the only way I know how.")

x, y = np.ndenumerate()
print(x, y)

# TypeError: ndenumerate.__init__() missing 1 required positional argument: 'arr'
```

```
i_am_an_array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

```
i_am_not_iterable:
32
```

```
32
```

```
That was surprising...Let's try again...
```

```
I feel like such a lowlife having to stoop to this level,
but denumerate leaves me no choice...
```

```
Gods, forgive me...I will throw an error the only way I know how.
```

TypeError

Traceback (most recent call last)

```
/tmp/ipykernel_38/2092068071.py in <module>
    20 print("Gods, forgive me...I will throw an error the only way I know how.")
    21
--> 22 x, y = np.ndenumerate()
    23 print(x, y)
    24
```

TypeError: __init__() missing 1 required positional argument: 'arr'

↑↑ This was tricky! `.ndenumerate()` even iterates over integers variables! So it was hard to come up with a way to break it. I had to stoop so low as just to not pass it anything, otherwise, it will always come up with something to give me back, it seems. I guess that is the functional equivalent of dividing by zero.

↑↑ **SUMMARY:** I can say with much certainty that `.ndenumerate` is a function I will be using a great deal as I work more with Numpy and step further into the field of data analysis. I can see its importance in manipulating data, but I also know that for me personally, it will be an important tool when I want to print out certain information just to wrap my mind around the data I am working with in various complex scenarios.

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/numpy-array-operations" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/numpy-array-operations
'https://jovian.ai/evanmarie/numpy-array-operations'
```

We Come to the Numpy Conclusion - Evaluation Time!

I am so glad I spent so much time working with Numpy over the course of this lecture, the assignments, and the research I have done on my own out of curiosity associated with this assignment and the 100 Numpy Exercises assignment. I can see how it is an invaluable tool in the data sciences, and I look forward to more deeply understanding its vast functionality.

Reference Links (sites I find helpful):

Real Python's Numpy tutorial:

<https://realpython.com/numpy-tutorial/>

The Numpy Documentation, of course:

<https://numpy.org/doc/stable/index.html>

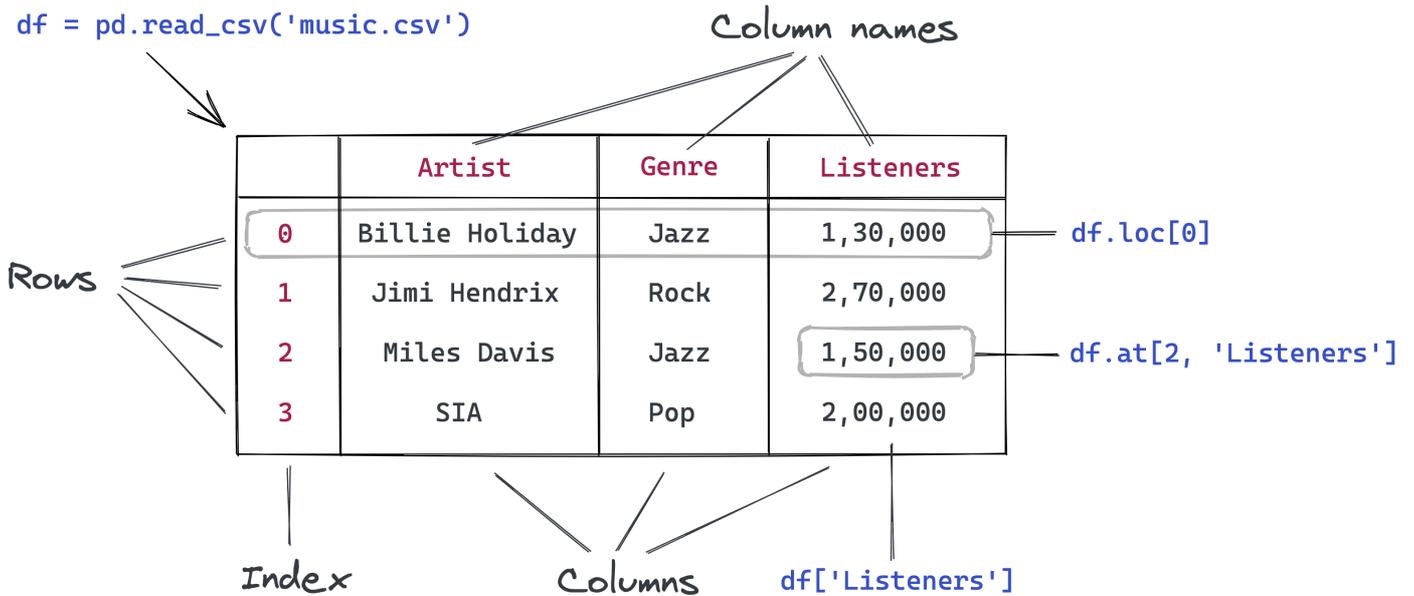
StackOverflow was useful!

<https://stackoverflow.com/>

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/numpy-array-operations" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/numpy-array-operations
'https://jovian.ai/evanmarie/numpy-array-operations'
```

Analyzing Tabular Data using Python and Pandas



This tutorial series is a beginner-friendly introduction to programming and data analysis using the Python programming language. These tutorials take a practical and coding-focused approach. The best way to learn the material is to execute the code and experiment with it yourself.

This tutorial covers the following topics:

- Reading a CSV file into a Pandas data frame
- Retrieving data from Pandas data frames
- Querying, sorting, and analyzing data
- Merging, grouping, and aggregation of data
- Extracting useful information from dates
- Basic plotting using line and bar charts
- Writing data frames to CSV files

How to run the code

This tutorial is an executable [Jupyter notebook](#) hosted on [Jovian](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Jupyter Notebooks: This tutorial is a [Jupyter notebook](#) - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc., instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Reading a CSV file using Pandas

[Pandas](#) is a popular Python library used for working in tabular data (similar to the data stored in a spreadsheet). Pandas provides helper functions to read data from various file formats like CSV, Excel spreadsheets, HTML tables, JSON, SQL, and more. Let's download a file `italy-covid-daywise.txt` which contains day-wise Covid-19 data for Italy in the following format:

```
date,new_cases,new_deaths,new_tests
2020-04-21,2256.0,454.0,28095.0
2020-04-22,2729.0,534.0,44248.0
2020-04-23,3370.0,437.0,37083.0
2020-04-24,2646.0,464.0,95273.0
2020-04-25,3021.0,420.0,38676.0
2020-04-26,2357.0,415.0,24113.0
2020-04-27,2324.0,260.0,26678.0
2020-04-28,1739.0,333.0,37554.0
...
```

This format of storing data is known as *comma-separated values* or CSV.

CSVs: A comma-separated values (CSV) file is a delimited text file that uses a comma to separate values. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. A CSV file typically stores tabular data (numbers and text) in plain text, in which case each line will have the same number of fields. (Wikipedia)

We'll download this file using the `urlretrieve` function from the `urllib.request` module.

```
from urllib.request import urlretrieve
```

```
italy_covid_url = 'https://gist.githubusercontent.com/aakashns/f6a004fa20c84fec53262f9a
urlretrieve(italy_covid_url, 'italy-covid-daywise.csv')
('italy-covid-daywise.csv', <http.client.HTTPMessage at 0x7fb7760b88b0>)
```

To read the file, we can use the `read_csv` method from Pandas. First, let's install the Pandas library.

```
!pip install pandas --upgrade --quiet
```

We can now import the `pandas` module. As a convention, it is imported with the alias `pd`.

```
import pandas as pd
```

```
covid_df = pd.read_csv('italy-covid-daywise.csv')
```

Data from the file is read and stored in a `DataFrame` object - one of the core data structures in Pandas for storing and working with tabular data. We typically use the `_df` suffix in the variable names for dataframes.

```
type(covid_df)
```

```
pandas.core.frame.DataFrame
```

```
covid_df.head()
```

	date	new_cases	new_deaths	new_tests
0	2019-12-31	0.0	0.0	NaN
1	2020-01-01	0.0	0.0	NaN
2	2020-01-02	0.0	0.0	NaN
3	2020-01-03	0.0	0.0	NaN
4	2020-01-04	0.0	0.0	NaN

```
covid_df
```

	date	new_cases	new_deaths	new_tests
0	2019-12-31	0.0	0.0	NaN
1	2020-01-01	0.0	0.0	NaN
2	2020-01-02	0.0	0.0	NaN
3	2020-01-03	0.0	0.0	NaN
4	2020-01-04	0.0	0.0	NaN
...
243	2020-08-30	1444.0	1.0	53541.0
244	2020-08-31	1365.0	4.0	42583.0
245	2020-09-01	996.0	6.0	54395.0
246	2020-09-02	975.0	8.0	NaN
247	2020-09-03	1326.0	6.0	NaN

248 rows × 4 columns

Here's what we can tell by looking at the dataframe:

- The file provides four day-wise counts for COVID-19 in Italy
- The metrics reported are new cases, deaths, and tests
- Data is provided for 248 days: from Dec 12, 2019, to Sep 3, 2020

Keep in mind that these are officially reported numbers. The actual number of cases & deaths may be higher, as not all cases are diagnosed.

We can view some basic information about the data frame using the `.info` method.

```
covid_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 248 entries, 0 to 247
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype
---  -
 0   date        248 non-null   object
 1   new_cases   248 non-null   float64
 2   new_deaths  248 non-null   float64
 3   new_tests   135 non-null   float64
dtypes: float64(3), object(1)
memory usage: 7.9+ KB
```

It appears that each column contains values of a specific data type. You can view statistical information for numerical columns (mean, standard deviation, minimum/maximum values, and the number of non-empty values) using the `.describe` method.

```
covid_df.describe()
```

	new_cases	new_deaths	new_tests
count	248.000000	248.000000	135.000000
mean	1094.818548	143.133065	31699.674074
std	1554.508002	227.105538	11622.209757
min	-148.000000	-31.000000	7841.000000
25%	123.000000	3.000000	25259.000000
50%	342.000000	17.000000	29545.000000
75%	1371.750000	175.250000	37711.000000
max	6557.000000	971.000000	95273.000000

The `columns` property contains the list of columns within the data frame.

```
covid_df.columns
```

```
Index(['date', 'new_cases', 'new_deaths', 'new_tests'], dtype='object')
```

You can also retrieve the number of rows and columns in the data frame using the `.shape` property

```
covid_df.shape
```

```
(248, 4)
```

Here's a summary of the functions & methods we've looked at so far:

- `pd.read_csv` - Read data from a CSV file into a Pandas DataFrame object
- `.info()` - View basic information about rows, columns & data types
- `.describe()` - View statistical information about numeric columns
- `.columns` - Get the list of column names
- `.shape` - Get the number of rows & columns as a tuple

Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. [Jovian](#) offers an easy way of saving and sharing your Jupyter notebooks online.

```
# Install the library  
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit(project='python-pandas-data-analysis')
```

[jovian] Error: Failed to read the Jupyter notebook. Please re-run this cell to try again. If the issue persists, provide the "filename" argument to "jovian.commit" e.g. "jovian.commit(filename='my-notebook.ipynb')"

The first time you run `jovian.commit`, you'll be asked to provide an API Key to securely upload the notebook to your Jovian account. You can get the API key from your [Jovian profile page](#) after logging in / signing up.

`jovian.commit` uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work.

Retrieving data from a data frame

The first thing you might want to do is retrieve data from this data frame, e.g., the counts of a specific day or the list of values in a particular column. To do this, it might help to understand the internal representation of data in a data frame. Conceptually, you can think of a dataframe as a dictionary of lists: keys are column names, and values are lists/arrays containing data for the respective columns.

```
# Pandas format is similar to this  
covid_data_dict = {  
    'date': ['2020-08-30', '2020-08-31', '2020-09-01', '2020-09-02', '2020-09-03'],  
    'new_cases': [1444, 1365, 996, 975, 1326],  
    'new_deaths': [1, 4, 6, 8, 6],  
    'new_tests': [53541, 42583, 54395, None, None]  
}
```

Representing data in the above format has a few benefits:

- All values in a column typically have the same type of value, so it's more efficient to store them in a single array.
- Retrieving the values for a particular row simply requires extracting the elements at a given index from each column array.
- The representation is more compact (column names are recorded only once) compared to other formats that use a dictionary for each row of data (see the example below).

```
# Pandas format is not similar to this
covid_data_list = [
    {'date': '2020-08-30', 'new_cases': 1444, 'new_deaths': 1, 'new_tests': 53541},
    {'date': '2020-08-31', 'new_cases': 1365, 'new_deaths': 4, 'new_tests': 42583},
    {'date': '2020-09-01', 'new_cases': 996, 'new_deaths': 6, 'new_tests': 54395},
    {'date': '2020-09-02', 'new_cases': 975, 'new_deaths': 8 },
    {'date': '2020-09-03', 'new_cases': 1326, 'new_deaths': 6},
]
```

With the dictionary of lists analogy in mind, you can now guess how to retrieve data from a data frame. For example, we can get a list of values from a specific column using the `[]` indexing notation.

```
covid_data_dict['new_cases']
```

```
[1444, 1365, 996, 975, 1326]
```

```
covid_df['new_cases']
```

```
0      0.0
1      0.0
2      0.0
3      0.0
4      0.0
```

```
...
```

```
243    1444.0
244    1365.0
245     996.0
246     975.0
247    1326.0
```

```
Name: new_cases, Length: 248, dtype: float64
```

Each column is represented using a data structure called `Series`, which is essentially a numpy array with some extra methods and properties.

```
type(covid_df['new_cases'])
```

```
pandas.core.series.Series
```

Like arrays, you can retrieve a specific value with a series using the indexing notation `[]`.

```
covid_df['new_cases'][246]
```

```
975.0
```

```
covid_df['new_tests'][240]
```

```
57640.0
```

Pandas also provides the `.at` method to retrieve the element at a specific row & column directly.

```
covid_df.at[246, 'new_cases']
```

```
975.0
```

```
covid_df.at[240, 'new_tests']
```

```
57640.0
```

Instead of using the indexing notation `[]`, Pandas also allows accessing columns as properties of the dataframe using the `.` notation. However, this method only works for columns whose names do not contain spaces or special characters.

```
covid_df.new_cases
```

```
0      0.0
1      0.0
2      0.0
3      0.0
4      0.0
```

```
...
```

```
243    1444.0
244    1365.0
245     996.0
246     975.0
247    1326.0
```

```
Name: new_cases, Length: 248, dtype: float64
```

Further, you can also pass a list of columns within the indexing notation `[]` to access a subset of the data frame with just the given columns.

```
cases_df = covid_df[['date', 'new_cases']]
cases_df
```

	date	new_cases
0	2019-12-31	0.0
1	2020-01-01	0.0
2	2020-01-02	0.0
3	2020-01-03	0.0

	date	new_cases
4	2020-01-04	0.0
...
243	2020-08-30	1444.0
244	2020-08-31	1365.0
245	2020-09-01	996.0
246	2020-09-02	975.0
247	2020-09-03	1326.0

248 rows × 2 columns

The new data frame `cases_df` is simply a "view" of the original data frame `covid_df`. Both point to the same data in the computer's memory. Changing any values inside one of them will also change the respective values in the other. Sharing data between data frames makes data manipulation in Pandas blazing fast. You needn't worry about the overhead of copying thousands or millions of rows every time you want to create a new data frame by operating on an existing one.

Sometimes you might need a full copy of the data frame, in which case you can use the `copy` method.

```
covid_df_copy = covid_df.copy()
```

The data within `covid_df_copy` is completely separate from `covid_df`, and changing values inside one of them will not affect the other.

To access a specific row of data, Pandas provides the `.loc` method.

```
covid_df
```

	date	new_cases	new_deaths	new_tests
0	2019-12-31	0.0	0.0	NaN
1	2020-01-01	0.0	0.0	NaN
2	2020-01-02	0.0	0.0	NaN
3	2020-01-03	0.0	0.0	NaN
4	2020-01-04	0.0	0.0	NaN
...
243	2020-08-30	1444.0	1.0	53541.0
244	2020-08-31	1365.0	4.0	42583.0
245	2020-09-01	996.0	6.0	54395.0
246	2020-09-02	975.0	8.0	NaN
247	2020-09-03	1326.0	6.0	NaN

248 rows × 4 columns

```
covid_df.loc[243]
```

```
date          2020-08-30
```

```
new_cases      1444.0
new_deaths      1.0
new_tests      53541.0
Name: 243, dtype: object
```

Each retrieved row is also a `Series` object.

```
type(covid_df.loc[243])
```

```
pandas.core.series.Series
```

We can use the `.head` and `.tail` methods to view the first or last few rows of data.

```
covid_df.head(5)
```

	date	new_cases	new_deaths	new_tests
0	2019-12-31	0.0	0.0	NaN
1	2020-01-01	0.0	0.0	NaN
2	2020-01-02	0.0	0.0	NaN
3	2020-01-03	0.0	0.0	NaN
4	2020-01-04	0.0	0.0	NaN

```
covid_df.tail(4)
```

	date	new_cases	new_deaths	new_tests
244	2020-08-31	1365.0	4.0	42583.0
245	2020-09-01	996.0	6.0	54395.0
246	2020-09-02	975.0	8.0	NaN
247	2020-09-03	1326.0	6.0	NaN

Notice above that while the first few values in the `new_cases` and `new_deaths` columns are `0`, the corresponding values within the `new_tests` column are `NaN`. That is because the CSV file does not contain any data for the `new_tests` column for specific dates (you can verify this by looking into the file). These values may be missing or unknown.

```
covid_df.at[0, 'new_tests']
```

```
nan
```

```
type(covid_df.at[0, 'new_tests'])
```

```
numpy.float64
```

The distinction between `0` and `NaN` is subtle but important. In this dataset, it represents that daily test numbers were not reported on specific dates. Italy started reporting daily tests on Apr 19, 2020. 93,5310 tests had already been conducted before Apr 19.

We can find the first index that doesn't contain a `NaN` value using a column's `first_valid_index` method.

```
covid_df.new_tests.first_valid_index()
```

111

Let's look at a few rows before and after this index to verify that the values change from NaN to actual numbers. We can do this by passing a range to `loc`.

```
covid_df.loc[108:113]
```

	date	new_cases	new_deaths	new_tests
108	2020-04-17	3786.0	525.0	NaN
109	2020-04-18	3493.0	575.0	NaN
110	2020-04-19	3491.0	480.0	NaN
111	2020-04-20	3047.0	433.0	7841.0
112	2020-04-21	2256.0	454.0	28095.0
113	2020-04-22	2729.0	534.0	44248.0

We can use the `.sample` method to retrieve a random sample of rows from the data frame.

```
covid_df.sample(10)
```

	date	new_cases	new_deaths	new_tests
238	2020-08-25	953.0	4.0	45798.0
98	2020-04-07	3599.0	636.0	NaN
5	2020-01-05	0.0	0.0	NaN
94	2020-04-03	4668.0	760.0	NaN
8	2020-01-08	0.0	0.0	NaN
18	2020-01-18	0.0	0.0	NaN
39	2020-02-08	0.0	0.0	NaN
200	2020-07-18	231.0	11.0	27569.0
161	2020-06-09	280.0	65.0	32200.0
221	2020-08-08	552.0	3.0	26631.0

Notice that even though we have taken a random sample, each row's original index is preserved - this is a useful property of data frames.

Here's a summary of the functions & methods we looked at in this section:

- `covid_df['new_cases']` - Retrieving columns as a Series using the column name
- `new_cases[243]` - Retrieving values from a Series using an index
- `covid_df.at[243, 'new_cases']` - Retrieving a single value from a data frame
- `covid_df.copy()` - Creating a deep copy of a data frame
- `covid_df.loc[243]` - Retrieving a row or range of rows of data from the data frame

- `head`, `tail`, and `sample` - Retrieving multiple rows of data from the data frame
- `covid_df.new_tests.first_valid_index` - Finding the first non-empty index in a series

Let's save a snapshot of our notebook before continuing.

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-pandas-data-analysis" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-pandas-data-analysis
```

```
'https://jovian.ai/evanmarie/python-pandas-data-analysis'
```

Analyzing data from data frames

Let's try to answer some questions about our data.

Q: What are the total number of reported cases and deaths related to Covid-19 in Italy?

Similar to Numpy arrays, a Pandas series supports the `sum` method to answer these questions.

```
total_cases = covid_df.new_cases.sum()
total_deaths = covid_df.new_deaths.sum()
```

```
print('The number of reported cases is {} and the number of reported deaths is {}'.format(total_cases, total_deaths))
```

The number of reported cases is 271515 and the number of reported deaths is 35497.

Q: What is the overall death rate (ratio of reported deaths to reported cases)?

```
death_rate = covid_df.new_deaths.sum() / covid_df.new_cases.sum()
```

```
print("The overall reported death rate in Italy is {:.2f} %".format(death_rate*100))
```

The overall reported death rate in Italy is 13.07 %.

Q: What is the overall number of tests conducted? A total of 935310 tests were conducted before daily test numbers were reported.

```
initial_tests = 935310
total_tests = initial_tests + covid_df.new_tests.sum()
```

```
total_tests
```

5214766.0

Q: What fraction of tests returned a positive result?

```
positive_rate = total_cases / total_tests
```

```
print('{:.2f}% of tests in Italy led to a positive diagnosis.'.format(positive_rate*100))
```

5.21% of tests in Italy led to a positive diagnosis.

Try asking and answering some more questions about the data using the empty cells below.

Let's save and commit our work before continuing.

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-pandas-data-analysis" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-pandas-data-analysis>

```
'https://jovian.ai/evanmarie/python-pandas-data-analysis'
```

Querying and sorting rows

Let's say we want only want to look at the days which had more than 1000 reported cases. We can use a boolean expression to check which rows satisfy this criterion.

```
high_new_cases = covid_df.new_cases > 1000
```

```
high_new_cases
```

```
0      False
1      False
2      False
3      False
4      False
...
243    True
244    True
245    False
246    False
247    True
```

```
Name: new_cases, Length: 248, dtype: bool
```

The boolean expression returns a series containing `True` and `False` boolean values. You can use this series to select a subset of rows from the original dataframe, corresponding to the `True` values in the series.

```
covid_df[high_new_cases]
```

	date	new_cases	new_deaths	new_tests
68	2020-03-08	1247.0	36.0	NaN
69	2020-03-09	1492.0	133.0	NaN
70	2020-03-10	1797.0	98.0	NaN
72	2020-03-12	2313.0	196.0	NaN
73	2020-03-13	2651.0	189.0	NaN
...
241	2020-08-28	1409.0	5.0	65135.0
242	2020-08-29	1460.0	9.0	64294.0
243	2020-08-30	1444.0	1.0	53541.0
244	2020-08-31	1365.0	4.0	42583.0
247	2020-09-03	1326.0	6.0	NaN

72 rows × 4 columns

We can write this succinctly on a single line by passing the boolean expression as an index to the data frame.

```
high_cases_df = covid_df[covid_df.new_cases > 1000]
```

```
high_cases_df
```

	date	new_cases	new_deaths	new_tests
68	2020-03-08	1247.0	36.0	NaN
69	2020-03-09	1492.0	133.0	NaN
70	2020-03-10	1797.0	98.0	NaN
72	2020-03-12	2313.0	196.0	NaN
73	2020-03-13	2651.0	189.0	NaN
...
241	2020-08-28	1409.0	5.0	65135.0
242	2020-08-29	1460.0	9.0	64294.0
243	2020-08-30	1444.0	1.0	53541.0
244	2020-08-31	1365.0	4.0	42583.0
247	2020-09-03	1326.0	6.0	NaN

72 rows × 4 columns

The data frame contains 72 rows, but only the first & last five rows are displayed by default with Jupyter for brevity. We can change some display options to view all the rows.

```

from IPython.display import display
with pd.option_context('display.max_rows', 100):
    display(covid_df[covid_df.new_cases > 1000])

```

	date	new_cases	new_deaths	new_tests
68	2020-03-08	1247.0	36.0	NaN
69	2020-03-09	1492.0	133.0	NaN
70	2020-03-10	1797.0	98.0	NaN
72	2020-03-12	2313.0	196.0	NaN
73	2020-03-13	2651.0	189.0	NaN
74	2020-03-14	2547.0	252.0	NaN
75	2020-03-15	3497.0	173.0	NaN
76	2020-03-16	2823.0	370.0	NaN
77	2020-03-17	4000.0	347.0	NaN
78	2020-03-18	3526.0	347.0	NaN
79	2020-03-19	4207.0	473.0	NaN
80	2020-03-20	5322.0	429.0	NaN
81	2020-03-21	5986.0	625.0	NaN
82	2020-03-22	6557.0	795.0	NaN
83	2020-03-23	5560.0	649.0	NaN
84	2020-03-24	4789.0	601.0	NaN
85	2020-03-25	5249.0	743.0	NaN
86	2020-03-26	5210.0	685.0	NaN
87	2020-03-27	6153.0	660.0	NaN
88	2020-03-28	5959.0	971.0	NaN
89	2020-03-29	5974.0	887.0	NaN
90	2020-03-30	5217.0	758.0	NaN
91	2020-03-31	4050.0	810.0	NaN
92	2020-04-01	4053.0	839.0	NaN
93	2020-04-02	4782.0	727.0	NaN
94	2020-04-03	4668.0	760.0	NaN
95	2020-04-04	4585.0	764.0	NaN
96	2020-04-05	4805.0	681.0	NaN
97	2020-04-06	4316.0	527.0	NaN
98	2020-04-07	3599.0	636.0	NaN
99	2020-04-08	3039.0	604.0	NaN
100	2020-04-09	3836.0	540.0	NaN
101	2020-04-10	4204.0	612.0	NaN
102	2020-04-11	3951.0	570.0	NaN
103	2020-04-12	4694.0	619.0	NaN

	date	new_cases	new_deaths	new_tests
104	2020-04-13	4092.0	431.0	NaN
105	2020-04-14	3153.0	564.0	NaN
106	2020-04-15	2972.0	604.0	NaN
107	2020-04-16	2667.0	578.0	NaN
108	2020-04-17	3786.0	525.0	NaN
109	2020-04-18	3493.0	575.0	NaN
110	2020-04-19	3491.0	480.0	NaN
111	2020-04-20	3047.0	433.0	7841.0
112	2020-04-21	2256.0	454.0	28095.0
113	2020-04-22	2729.0	534.0	44248.0
114	2020-04-23	3370.0	437.0	37083.0
115	2020-04-24	2646.0	464.0	95273.0
116	2020-04-25	3021.0	420.0	38676.0
117	2020-04-26	2357.0	415.0	24113.0
118	2020-04-27	2324.0	260.0	26678.0
119	2020-04-28	1739.0	333.0	37554.0
120	2020-04-29	2091.0	382.0	38589.0
121	2020-04-30	2086.0	323.0	41441.0
122	2020-05-01	1872.0	285.0	43732.0
123	2020-05-02	1965.0	269.0	31231.0
124	2020-05-03	1900.0	474.0	27047.0
125	2020-05-04	1389.0	174.0	22999.0
126	2020-05-05	1221.0	195.0	32211.0
127	2020-05-06	1075.0	236.0	37771.0
128	2020-05-07	1444.0	369.0	13665.0
129	2020-05-08	1401.0	274.0	45428.0
130	2020-05-09	1327.0	243.0	36091.0
131	2020-05-10	1083.0	194.0	31384.0
134	2020-05-13	1402.0	172.0	37049.0
236	2020-08-23	1071.0	3.0	47463.0
237	2020-08-24	1209.0	7.0	33358.0
240	2020-08-27	1366.0	13.0	57640.0
241	2020-08-28	1409.0	5.0	65135.0
242	2020-08-29	1460.0	9.0	64294.0
243	2020-08-30	1444.0	1.0	53541.0
244	2020-08-31	1365.0	4.0	42583.0
247	2020-09-03	1326.0	6.0	NaN

We can also formulate more complex queries that involve multiple columns. As an example, let's try to determine the days when the ratio of cases reported to tests conducted is higher than the overall `positive_rate` .

```
positive_rate
```

```
0.05206657403227681
```

```
high_ratio_df = covid_df[covid_df.new_cases / covid_df.new_tests > positive_rate]
```

```
high_ratio_df
```

	date	new_cases	new_deaths	new_tests
111	2020-04-20	3047.0	433.0	7841.0
112	2020-04-21	2256.0	454.0	28095.0
113	2020-04-22	2729.0	534.0	44248.0
114	2020-04-23	3370.0	437.0	37083.0
116	2020-04-25	3021.0	420.0	38676.0
117	2020-04-26	2357.0	415.0	24113.0
118	2020-04-27	2324.0	260.0	26678.0
120	2020-04-29	2091.0	382.0	38589.0
123	2020-05-02	1965.0	269.0	31231.0
124	2020-05-03	1900.0	474.0	27047.0
125	2020-05-04	1389.0	174.0	22999.0
128	2020-05-07	1444.0	369.0	13665.0

The result of performing an operation on two columns is a new series.

```
covid_df.new_cases / covid_df.new_tests
```

```
0      NaN
1      NaN
2      NaN
3      NaN
4      NaN
```

```
...
```

```
243    0.026970
```

```
244    0.032055
```

```
245    0.018311
```

```
246      NaN
```

```
247      NaN
```

```
Length: 248, dtype: float64
```

We can use this series to add a new column to the data frame.

```
covid_df['positive_rate'] = covid_df.new_cases / covid_df.new_tests
```

```
covid_df
```

	date	new_cases	new_deaths	new_tests	positive_rate
0	2019-12-31	0.0	0.0	NaN	NaN
1	2020-01-01	0.0	0.0	NaN	NaN
2	2020-01-02	0.0	0.0	NaN	NaN
3	2020-01-03	0.0	0.0	NaN	NaN
4	2020-01-04	0.0	0.0	NaN	NaN
...
243	2020-08-30	1444.0	1.0	53541.0	0.026970
244	2020-08-31	1365.0	4.0	42583.0	0.032055
245	2020-09-01	996.0	6.0	54395.0	0.018311
246	2020-09-02	975.0	8.0	NaN	NaN
247	2020-09-03	1326.0	6.0	NaN	NaN

248 rows × 5 columns

However, keep in mind that sometimes it takes a few days to get the results for a test, so we can't compare the number of new cases with the number of tests conducted on the same day. Any inference based on this `positive_rate` column is likely to be incorrect. It's essential to watch out for such subtle relationships that are often not conveyed within the CSV file and require some external context. It's always a good idea to read through the documentation provided with the dataset or ask for more information.

For now, let's remove the `positive_rate` column using the `drop` method.

```
covid_df.drop(columns=['positive_rate'], inplace=True)
```

Can you figure the purpose of the `inplace` argument?

Sorting rows using column values

The rows can also be sorted by a specific column using `.sort_values`. Let's sort to identify the days with the highest number of cases, then chain it with the `head` method to list just the first ten results.

```
covid_df.sort_values('new_cases', ascending=False).head(10)
```

	date	new_cases	new_deaths	new_tests
82	2020-03-22	6557.0	795.0	NaN
87	2020-03-27	6153.0	660.0	NaN
81	2020-03-21	5986.0	625.0	NaN
89	2020-03-29	5974.0	887.0	NaN
88	2020-03-28	5959.0	971.0	NaN
83	2020-03-23	5560.0	649.0	NaN
80	2020-03-20	5322.0	429.0	NaN
85	2020-03-25	5249.0	743.0	NaN
90	2020-03-30	5217.0	758.0	NaN
86	2020-03-26	5210.0	685.0	NaN

It looks like the last two weeks of March had the highest number of daily cases. Let's compare this to the days where the highest number of deaths were recorded.

```
covid_df.sort_values('new_deaths', ascending=False).head(10)
```

	date	new_cases	new_deaths	new_tests
88	2020-03-28	5959.0	971.0	NaN
89	2020-03-29	5974.0	887.0	NaN
92	2020-04-01	4053.0	839.0	NaN
91	2020-03-31	4050.0	810.0	NaN
82	2020-03-22	6557.0	795.0	NaN
95	2020-04-04	4585.0	764.0	NaN
94	2020-04-03	4668.0	760.0	NaN
90	2020-03-30	5217.0	758.0	NaN
85	2020-03-25	5249.0	743.0	NaN
93	2020-04-02	4782.0	727.0	NaN

It appears that daily deaths hit a peak just about a week after the peak in daily new cases.

Let's also look at the days with the least number of cases. We might expect to see the first few days of the year on this list.

```
covid_df.sort_values('new_cases').head(10)
```

	date	new_cases	new_deaths	new_tests
172	2020-06-20	-148.0	47.0	29875.0
0	2019-12-31	0.0	0.0	NaN
29	2020-01-29	0.0	0.0	NaN
30	2020-01-30	0.0	0.0	NaN
32	2020-02-01	0.0	0.0	NaN
33	2020-02-02	0.0	0.0	NaN
34	2020-02-03	0.0	0.0	NaN
36	2020-02-05	0.0	0.0	NaN
37	2020-02-06	0.0	0.0	NaN
38	2020-02-07	0.0	0.0	NaN

It seems like the count of new cases on Jun 20, 2020, was -148 , a negative number! Not something we might have expected, but that's the nature of real-world data. It could be a data entry error, or the government may have issued a correction to account for miscounting in the past. Can you dig through news articles online and figure out why the number was negative?

Let's look at some days before and after Jun 20, 2020.

```
covid_df.loc[169:175]
```

	date	new_cases	new_deaths	new_tests
169	2020-06-17	210.0	34.0	33957.0
170	2020-06-18	328.0	43.0	32921.0
171	2020-06-19	331.0	66.0	28570.0
172	2020-06-20	-148.0	47.0	29875.0
173	2020-06-21	264.0	49.0	24581.0
174	2020-06-22	224.0	24.0	16152.0
175	2020-06-23	221.0	23.0	23225.0

For now, let's assume this was indeed a data entry error. We can use one of the following approaches for dealing with the missing or faulty value:

1. Replace it with 0.
2. Replace it with the average of the entire column
3. Replace it with the average of the values on the previous & next date
4. Discard the row entirely

Which approach you pick requires some context about the data and the problem. In this case, since we are dealing with data ordered by date, we can go ahead with the third approach.

You can use the `.at` method to modify a specific value within the dataframe.

```
covid_df.at[172, 'new_cases'] = (covid_df.at[171, 'new_cases'] + covid_df.at[173, 'new_
```

Here's a summary of the functions & methods we looked at in this section:

- `covid_df.new_cases.sum()` - Computing the sum of values in a column or series
- `covid_df[covid_df.new_cases > 1000]` - Querying a subset of rows satisfying the chosen criteria using boolean expressions
- `df['pos_rate'] = df.new_cases/df.new_tests` - Adding new columns by combining data from existing columns
- `covid_df.drop('positive_rate')` - Removing one or more columns from the data frame
- `sort_values` - Sorting the rows of a data frame using column values
- `covid_df.at[172, 'new_cases'] = ...` - Replacing a value within the data frame

Let's save and commit our work before continuing.

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-pandas-data-analysis" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-pandas-data-analysis>

'<https://jovian.ai/evanmarie/python-pandas-data-analysis>'

Working with dates

While we've looked at overall numbers for the cases, tests, positive rate, etc., it would also be useful to study these numbers on a month-by-month basis. The `date` column might come in handy here, as Pandas provides many utilities for working with dates.

```
covid_df.date
```

```
0      2019-12-31
1      2020-01-01
2      2020-01-02
3      2020-01-03
4      2020-01-04
...
243    2020-08-30
244    2020-08-31
245    2020-09-01
246    2020-09-02
247    2020-09-03
```

```
Name: date, Length: 248, dtype: object
```

The data type of `date` is currently `object`, so Pandas does not know that this column is a date. We can convert it into a `datetime` column using the `pd.to_datetime` method.

```
covid_df['date'] = pd.to_datetime(covid_df.date)
```

```
covid_df['date']
```

```
0      2019-12-31
1      2020-01-01
2      2020-01-02
3      2020-01-03
4      2020-01-04
...
243    2020-08-30
244    2020-08-31
245    2020-09-01
246    2020-09-02
247    2020-09-03
```

```
Name: date, Length: 248, dtype: datetime64[ns]
```

You can see that it now has the datatype `datetime64`. We can now extract different parts of the data into separate columns, using the `DatetimeIndex` class ([view docs](#)).

```
covid_df['year'] = pd.DatetimeIndex(covid_df.date).year
covid_df['month'] = pd.DatetimeIndex(covid_df.date).month
```

```
covid_df['day'] = pd.DatetimeIndex(covid_df.date).day
covid_df['weekday'] = pd.DatetimeIndex(covid_df.date).weekday
```

```
covid_df
```

	date	new_cases	new_deaths	new_tests	year	month	day	weekday
0	2019-12-31	0.0	0.0	NaN	2019	12	31	1
1	2020-01-01	0.0	0.0	NaN	2020	1	1	2
2	2020-01-02	0.0	0.0	NaN	2020	1	2	3
3	2020-01-03	0.0	0.0	NaN	2020	1	3	4
4	2020-01-04	0.0	0.0	NaN	2020	1	4	5
...
243	2020-08-30	1444.0	1.0	53541.0	2020	8	30	6
244	2020-08-31	1365.0	4.0	42583.0	2020	8	31	0
245	2020-09-01	996.0	6.0	54395.0	2020	9	1	1
246	2020-09-02	975.0	8.0	NaN	2020	9	2	2
247	2020-09-03	1326.0	6.0	NaN	2020	9	3	3

248 rows × 8 columns

Let's check the overall metrics for May. We can query the rows for May, choose a subset of columns, and use the `sum` method to aggregate each selected column's values.

```
# Query the rows for May
covid_df_may = covid_df[covid_df.month == 5]

# Extract the subset of columns to be aggregated
covid_df_may_metrics = covid_df_may[['new_cases', 'new_deaths', 'new_tests']]

# Get the column-wise sum
covid_may_totals = covid_df_may_metrics.sum()
```

```
covid_may_totals
```

```
new_cases      29073.0
new_deaths      5658.0
new_tests     1078720.0
dtype: float64
```

```
type(covid_may_totals)
```

```
pandas.core.series.Series
```

We can also combine the above operations into a single statement.

```
covid_df[covid_df.month == 5][['new_cases', 'new_deaths', 'new_tests']].sum()
```

```
new_cases      29073.0
new_deaths     5658.0
new_tests     1078720.0
dtype: float64
```

As another example, let's check if the number of cases reported on Sundays is higher than the average number of cases reported every day. This time, we might want to aggregate columns using the `.mean` method.

```
# Overall average
covid_df.new_cases.mean()
```

```
1096.6149193548388
```

```
# Average for Sundays
covid_df[covid_df.weekday == 6].new_cases.mean()
```

```
1247.2571428571428
```

It seems like more cases were reported on Sundays compared to other days.

Try asking and answering some more date-related questions about the data using the cells below.

Let's save and commit our work before continuing.

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-pandas-data-analysis" on https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-pandas-data-analysis
```

```
'https://jovian.ai/evanmarie/python-pandas-data-analysis'
```

Grouping and aggregation

As a next step, we might want to summarize the day-wise data and create a new dataframe with month-wise data. We can use the `groupby` function to create a group for each month, select the columns we wish to aggregate, and aggregate them using the `sum` method.

```
covid_month_df = covid_df.groupby('month')[['new_cases', 'new_deaths', 'new_tests']].su
```

```
covid_month_df
```

	new_cases	new_deaths	new_tests
month			
1	3.0	0.0	0.0
2	885.0	21.0	0.0
3	100851.0	11570.0	0.0
4	101852.0	16091.0	419591.0
5	29073.0	5658.0	1078720.0
6	8217.5	1404.0	830354.0
7	6722.0	388.0	797692.0
8	21060.0	345.0	1098704.0
9	3297.0	20.0	54395.0
12	0.0	0.0	0.0

The result is a new data frame that uses unique values from the column passed to `groupby` as the index. Grouping and aggregation is a powerful method for progressively summarizing data into smaller data frames.

Instead of aggregating by sum, you can also aggregate by other measures like mean. Let's compute the average number of daily new cases, deaths, and tests for each month.

```
covid_month_mean_df = covid_df.groupby('month')[['new_cases', 'new_deaths', 'new_tests']
```

```
monthly_groups = covid_df.groupby('month')
```

```
monthly_groups[['new_cases', 'new_deaths', 'new_tests']].sum()
```

	new_cases	new_deaths	new_tests
month			
1	3.0	0.0	0.0
2	885.0	21.0	0.0
3	100851.0	11570.0	0.0
4	101852.0	16091.0	419591.0
5	29073.0	5658.0	1078720.0
6	8217.5	1404.0	830354.0
7	6722.0	388.0	797692.0
8	21060.0	345.0	1098704.0
9	3297.0	20.0	54395.0
12	0.0	0.0	0.0

```
covid_month_mean_df
```

	new_cases	new_deaths	new_tests
month			
1	0.096774	0.000000	NaN
2	30.517241	0.724138	NaN
3	3253.258065	373.225806	NaN
4	3395.066667	536.366667	38144.636364
5	937.838710	182.516129	34797.419355
6	273.916667	46.800000	27678.466667
7	216.838710	12.516129	25732.000000
8	679.354839	11.129032	35442.064516
9	1099.000000	6.666667	54395.000000
12	0.000000	0.000000	NaN

Apart from grouping, another form of aggregation is the running or cumulative sum of cases, tests, or death up to each row's date. We can use the `cumsum` method to compute the cumulative sum of a column as a new series. Let's add three new columns: `total_cases`, `total_deaths`, and `total_tests`.

```
covid_df['total_cases'] = covid_df.new_cases.cumsum()
```

```
covid_df['total_deaths'] = covid_df.new_deaths.cumsum()
```

```
covid_df['total_tests'] = covid_df.new_tests.cumsum() + initial_tests
```

We've also included the initial test count in `total_test` to account for tests conducted before daily reporting was started.

```
covid_df
```

	date	new_cases	new_deaths	new_tests	year	month	day	weekday	total_cases	total_deaths	total_tests
0	2019-12-31	0.0	0.0	NaN	2019	12	31	1	0.0	0.0	NaN
1	2020-01-01	0.0	0.0	NaN	2020	1	1	2	0.0	0.0	NaN
2	2020-01-02	0.0	0.0	NaN	2020	1	2	3	0.0	0.0	NaN
3	2020-01-03	0.0	0.0	NaN	2020	1	3	4	0.0	0.0	NaN

	date	new_cases	new_deaths	new_tests	year	month	day	weekday	total_cases	total_deaths	total_tests
4	2020-01-04	0.0	0.0	NaN	2020	1	4	5	0.0	0.0	NaN
...
243	2020-08-30	1444.0	1.0	53541.0	2020	8	30	6	267298.5	35473.0	5117788.0
244	2020-08-31	1365.0	4.0	42583.0	2020	8	31	0	268663.5	35477.0	5160371.0
245	2020-09-01	996.0	6.0	54395.0	2020	9	1	1	269659.5	35483.0	5214766.0
246	2020-09-02	975.0	8.0	NaN	2020	9	2	2	270634.5	35491.0	NaN
247	2020-09-03	1326.0	6.0	NaN	2020	9	3	3	271960.5	35497.0	NaN

248 rows × 11 columns

Notice how the NaN values in the total_tests column remain unaffected.

Merging data from multiple sources

To determine other metrics like test per million, cases per million, etc., we require some more information about the country, viz. its population. Let's download another file `locations.csv` that contains health-related information for many countries, including Italy.

```
urlretrieve('https://gist.githubusercontent.com/aakashns/8684589ef4f266116cdce023377fc9
            'locations.csv')
```

```
('locations.csv', <http.client.HTTPMessage at 0x7fb77609e910>)
```

```
locations_df = pd.read_csv('locations.csv')
```

```
locations_df
```

	location	continent	population	life_expectancy	hospital_beds_per_thousand	gdp_per_capita
0	Afghanistan	Asia	3.892834e+07	64.83	0.500	1803.987
1	Albania	Europe	2.877800e+06	78.57	2.890	11803.431
2	Algeria	Africa	4.385104e+07	76.88	1.900	13913.839
3	Andorra	Europe	7.726500e+04	83.73	NaN	NaN
4	Angola	Africa	3.286627e+07	61.15	NaN	5819.495
...
207	Yemen	Asia	2.982597e+07	66.12	0.700	1479.147
208	Zambia	Africa	1.838396e+07	63.89	2.000	3689.251
209	Zimbabwe	Africa	1.486293e+07	61.49	1.700	1899.775
210	World	NaN	7.794799e+09	72.58	2.705	15469.207
211	International	NaN	NaN	NaN	NaN	NaN

212 rows × 6 columns

```
locations_df[locations_df.location == "Italy"]
```

	location	continent	population	life_expectancy	hospital_beds_per_thousand	gdp_per_capita
97	Italy	Europe	60461828.0	83.51	3.18	35220.084

We can merge this data into our existing data frame by adding more columns. However, to merge two data frames, we need at least one common column. Let's insert a `location` column in the `covid_df` dataframe with all values set to "Italy" .

```
covid_df['location'] = "Italy"
```

```
covid_df
```

	date	new_cases	new_deaths	new_tests	year	month	day	weekday	total_cases	total_deaths	total_tests	l
0	2019-12-31	0.0	0.0	NaN	2019	12	31	1	0.0	0.0	NaN	
1	2020-01-01	0.0	0.0	NaN	2020	1	1	2	0.0	0.0	NaN	
2	2020-01-02	0.0	0.0	NaN	2020	1	2	3	0.0	0.0	NaN	
3	2020-01-03	0.0	0.0	NaN	2020	1	3	4	0.0	0.0	NaN	
4	2020-01-04	0.0	0.0	NaN	2020	1	4	5	0.0	0.0	NaN	
...	
243	2020-08-30	1444.0	1.0	53541.0	2020	8	30	6	267298.5	35473.0	5117788.0	
244	2020-08-31	1365.0	4.0	42583.0	2020	8	31	0	268663.5	35477.0	5160371.0	
245	2020-09-01	996.0	6.0	54395.0	2020	9	1	1	269659.5	35483.0	5214766.0	
246	2020-09-02	975.0	8.0	NaN	2020	9	2	2	270634.5	35491.0	NaN	
247	2020-09-03	1326.0	6.0	NaN	2020	9	3	3	271960.5	35497.0	NaN	

248 rows × 12 columns

We can now add the columns from `locations_df` into `covid_df` using the `.merge` method.

```
merged_df = covid_df.merge(locations_df, on="location")
```

```
merged_df
```

	date	new_cases	new_deaths	new_tests	year	month	day	weekday	total_cases	total_deaths	total_tests	l
0	2019-12-31	0.0	0.0	NaN	2019	12	31	1	0.0	0.0	NaN	

	date	new_cases	new_deaths	new_tests	year	month	day	weekday	total_cases	total_deaths	total_tests
243	2020-08-30	1444.0	1.0	53541.0	2020	8	30	6	267298.5	35473.0	5117788.0
244	2020-08-31	1365.0	4.0	42583.0	2020	8	31	0	268663.5	35477.0	5160371.0
245	2020-09-01	996.0	6.0	54395.0	2020	9	1	1	269659.5	35483.0	5214766.0
246	2020-09-02	975.0	8.0	NaN	2020	9	2	2	270634.5	35491.0	NaN
247	2020-09-03	1326.0	6.0	NaN	2020	9	3	3	271960.5	35497.0	NaN

248 rows × 20 columns

Let's save and commit our work before continuing.

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-pandas-data-analysis" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-pandas-data-analysis>

'<https://jovian.ai/evanmarie/python-pandas-data-analysis>'

Writing data back to files

After completing your analysis and adding new columns, you should write the results back to a file. Otherwise, the data will be lost when the Jupyter notebook shuts down. Before writing to file, let us first create a data frame containing just the columns we wish to record.

```
result_df = merged_df[['date',
                        'new_cases',
                        'total_cases',
                        'new_deaths',
                        'total_deaths',
                        'new_tests',
                        'total_tests',
                        'cases_per_million',
                        'deaths_per_million',
                        'tests_per_million']]
```

```
result_df
```

	date	new_cases	total_cases	new_deaths	total_deaths	new_tests	total_tests	cases_per_million	deaths_per_m
0	2019-12-31	0.0	0.0	0.0	0.0	NaN	NaN	0.000000	0.000000
1	2020-01-01	0.0	0.0	0.0	0.0	NaN	NaN	0.000000	0.000000

	date	new_cases	total_cases	new_deaths	total_deaths	new_tests	total_tests	cases_per_million	deaths_per_m
2	2020-01-02	0.0	0.0	0.0	0.0	NaN	NaN	0.000000	0.000
3	2020-01-03	0.0	0.0	0.0	0.0	NaN	NaN	0.000000	0.000
4	2020-01-04	0.0	0.0	0.0	0.0	NaN	NaN	0.000000	0.000
...
243	2020-08-30	1444.0	267298.5	1.0	35473.0	53541.0	5117788.0	4420.946386	586.700
244	2020-08-31	1365.0	268663.5	4.0	35477.0	42583.0	5160371.0	4443.522614	586.760
245	2020-09-01	996.0	269659.5	6.0	35483.0	54395.0	5214766.0	4459.995818	586.860
246	2020-09-02	975.0	270634.5	8.0	35491.0	NaN	NaN	4476.121695	586.990
247	2020-09-03	1326.0	271960.5	6.0	35497.0	NaN	NaN	4498.052887	587.090

248 rows × 10 columns

To write the data from the data frame into a file, we can use the `to_csv` function.

```
result_df.to_csv('results.csv', index=None)
```

The `to_csv` function also includes an additional column for storing the index of the dataframe by default. We pass `index=None` to turn off this behavior. You can now verify that the `results.csv` is created and contains data from the data frame in CSV format:

```
date,new_cases,total_cases,new_deaths,total_deaths,new_tests,total_tests,cases_per_mi
2020-02-27,78.0,400.0,1.0,12.0,,,6.61574439992122,0.1984723319976366,
2020-02-28,250.0,650.0,5.0,17.0,,,10.750584649871982,0.28116913699665186,
2020-02-29,238.0,888.0,4.0,21.0,,,14.686952567825108,0.34732658099586405,
2020-03-01,240.0,1128.0,8.0,29.0,,,18.656399207777838,0.47964146899428844,
2020-03-02,561.0,1689.0,6.0,35.0,,,27.93498072866735,0.5788776349931067,
2020-03-03,347.0,2036.0,17.0,52.0,,,33.67413899559901,0.8600467719897585,
...
```

You can attach the `results.csv` file to our notebook while uploading it to [Jovian](https://jovian.ai) using the `outputs` argument to `jovian.commit`.

```
import jovian
```

```
jovian.commit(outputs=['results.csv'])
```

```
[jovian] Updating notebook "evanmarie/python-pandas-data-analysis" on https://jovian.ai
[jovian] Uploading additional outputs...
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-pandas-data-analysis
```

'<https://jovian.ai/evanmarie/python-pandas-data-analysis>'

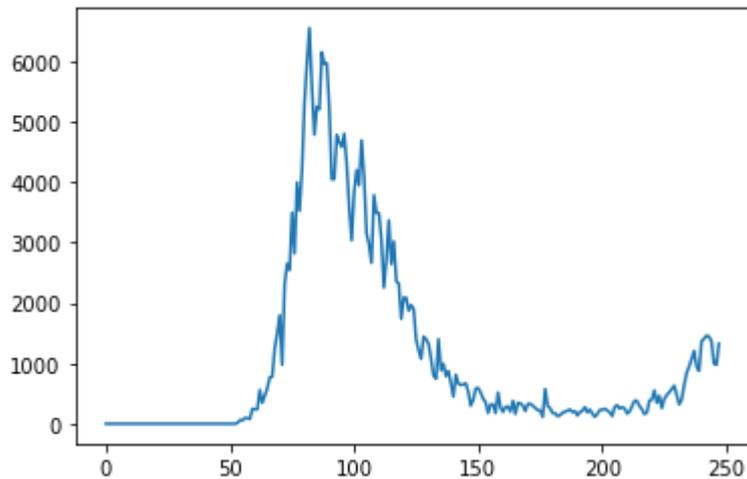
You can find the CSV file in the "Files" tab on the project page.

Bonus: Basic Plotting with Pandas

We generally use a library like `matplotlib` or `seaborn` plot graphs within a Jupyter notebook. However, Pandas dataframes & series provide a handy `.plot` method for quick and easy plotting.

Let's plot a line graph showing how the number of daily cases varies over time.

```
result_df.new_cases.plot();
```



While this plot shows the overall trend, it's hard to tell where the peak occurred, as there are no dates on the X-axis. We can use the `date` column as the index for the data frame to address this issue.

```
result_df.set_index('date', inplace=True)
```

```
result_df
```

	new_cases	total_cases	new_deaths	total_deaths	new_tests	total_tests	cases_per_million	deaths_per_million
date								
2019-12-31	0.0	0.0	0.0	0.0	NaN	NaN	0.000000	0.000000
2020-01-01	0.0	0.0	0.0	0.0	NaN	NaN	0.000000	0.000000
2020-01-02	0.0	0.0	0.0	0.0	NaN	NaN	0.000000	0.000000
2020-01-03	0.0	0.0	0.0	0.0	NaN	NaN	0.000000	0.000000
2020-01-04	0.0	0.0	0.0	0.0	NaN	NaN	0.000000	0.000000
...
2020-08-30	1444.0	267298.5	1.0	35473.0	53541.0	5117788.0	4420.946386	586.700753
2020-08-31	1365.0	268663.5	4.0	35477.0	42583.0	5160371.0	4443.522614	586.766910

	new_cases	total_cases	new_deaths	total_deaths	new_tests	total_tests	cases_per_million	deaths_per_million
date								
2020-09-01	996.0	269659.5	6.0	35483.0	54395.0	5214766.0	4459.995818	586.866146
2020-09-02	975.0	270634.5	8.0	35491.0	NaN	NaN	4476.121695	586.998461
2020-09-03	1326.0	271960.5	6.0	35497.0	NaN	NaN	4498.052887	587.097697

248 rows × 9 columns

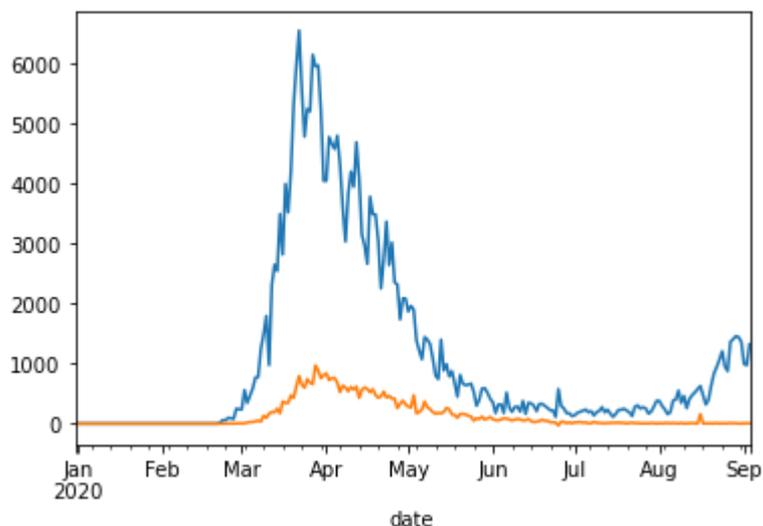
Notice that the index of a data frame doesn't have to be numeric. Using the date as the index also allows us to get the data for a specific data using `.loc`.

```
result_df.loc['2020-09-01']
```

```
new_cases          9.960000e+02
total_cases        2.696595e+05
new_deaths         6.000000e+00
total_deaths       3.548300e+04
new_tests          5.439500e+04
total_tests        5.214766e+06
cases_per_million  4.459996e+03
deaths_per_million 5.868661e+02
tests_per_million  8.624890e+04
Name: 2020-09-01 00:00:00, dtype: float64
```

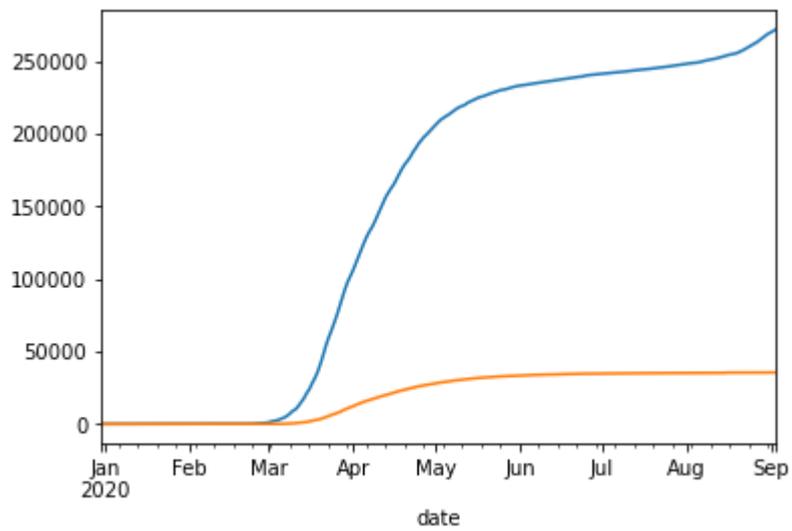
Let's plot the new cases & new deaths per day as line graphs.

```
result_df.new_cases.plot()
result_df.new_deaths.plot();
```



We can also compare the total cases vs. total deaths.

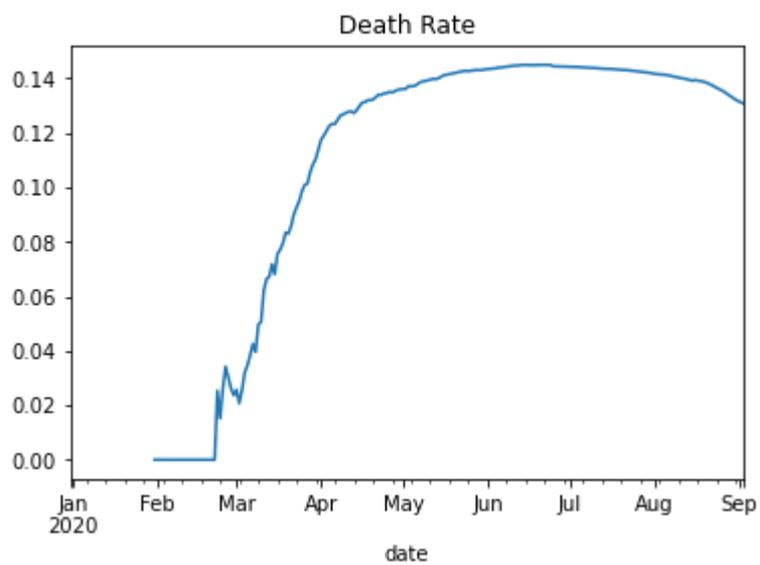
```
result_df.total_cases.plot()
result_df.total_deaths.plot();
```



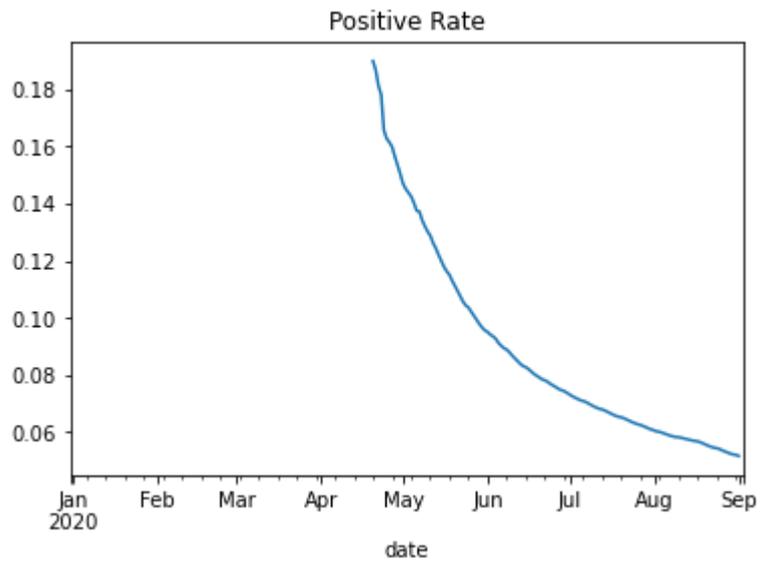
Let's see how the death rate and positive testing rates vary over time.

```
death_rate = result_df.total_deaths / result_df.total_cases
```

```
death_rate.plot(title='Death Rate');
```

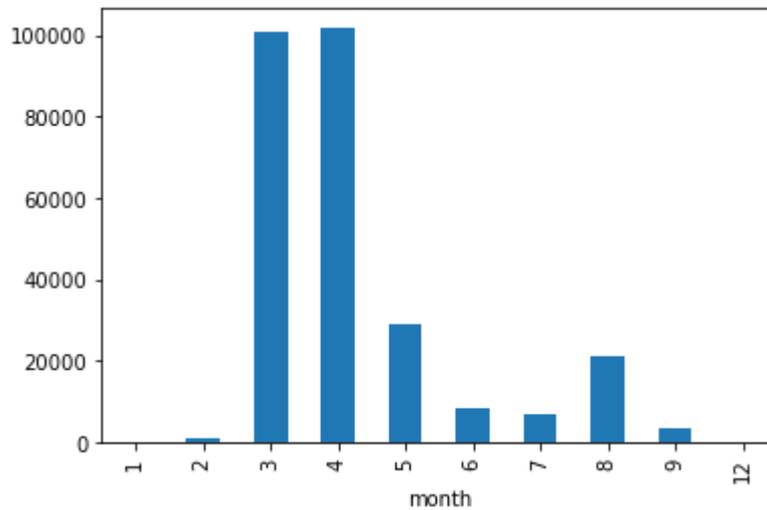


```
positive_rates = result_df.total_cases / result_df.total_tests
positive_rates.plot(title='Positive Rate');
```



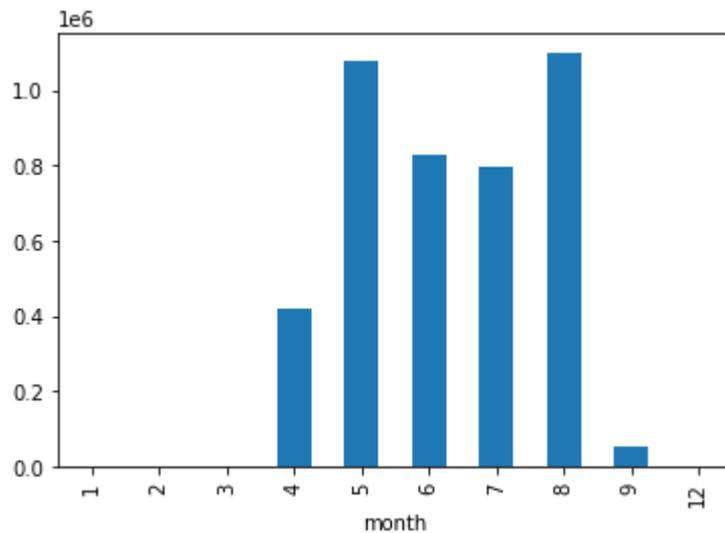
Finally, let's plot some month-wise data using a bar chart to visualize the trend at a higher level.

```
covid_month_df.new_cases.plot(kind='bar');
```



```
covid_month_df.new_tests.plot(kind='bar')
```

<AxesSubplot: xlabel='month'>



Let's save and commit our work to Jovian.

```
import jovian
```

```
jovian.commit()
```

Exercises

Try the following exercises to become familiar with Pandas dataframe and practice your skills:

- Assignment on Pandas dataframes: <https://jovian.ml/aakashns/pandas-practice-assignment>
- Additional exercises on Pandas: https://github.com/guipsamora/pandas_exercises
- Try downloading and analyzing some data from Kaggle: <https://www.kaggle.com/datasets>

Summary and Further Reading

We've covered the following topics in this tutorial:

- Reading a CSV file into a Pandas data frame
- Retrieving data from Pandas data frames
- Querying, sorting, and analyzing data
- Merging, grouping, and aggregation of data
- Extracting useful information from dates
- Basic plotting using line and bar charts
- Writing data frames to CSV files

Check out the following resources to learn more about Pandas:

- User guide for Pandas: https://pandas.pydata.org/docs/user_guide/index.html
- Python for Data Analysis (book by Wes McKinney - creator of Pandas): <https://www.oreilly.com/library/view/python-for-data/9781491957653/>

You are ready to move on to the next tutorial: [Data Visualization using Matplotlib & Seaborn](#).

Questions for Revision

Try answering the following questions to test your understanding of the topics covered in this notebook:

1. What is Pandas? What makes it useful?
2. How do you install the Pandas library?
3. How do you import the pandas module?
4. What is the common alias used while importing the pandas module?
5. How do you read a CSV file using Pandas? Give an example?
6. What are some other file formats you can read using Pandas? Illustrate with examples.
7. What are Pandas dataframes?

8. How are Pandas dataframes different from Numpy arrays?
9. How do you find the number of rows and columns in a dataframe?
10. How do you get the list of columns in a dataframe?
11. What is the purpose of the describe method of a dataframe?
12. How are the info and describe dataframe methods different?
13. Is a Pandas dataframe conceptually similar to a list of dictionaries or a dictionary of lists? Explain with an example.
14. What is a Pandas Series? How is it different from a Numpy array?
15. How do you access a column from a dataframe?
16. How do you access a row from a dataframe?
17. How do you access an element at a specific row & column of a dataframe?
18. How do you create a subset of a dataframe with a specific set of columns?
19. How do you create a subset of a dataframe with a specific range of rows?
20. Does changing a value within a dataframe affect other dataframes created using a subset of the rows or columns? Why is it so?
21. How do you create a copy of a dataframe?
22. Why should you avoid creating too many copies of a dataframe?
23. How do you view the first few rows of a dataframe?
24. How do you view the last few rows of a dataframe?
25. How do you view a random selection of rows of a dataframe?
26. What is the "index" in a dataframe? How is it useful?
27. What does a NaN value in a Pandas dataframe represent?
28. How is Nan different from 0?
29. How do you identify the first non-empty row in a Pandas series or column?
30. What is the difference between df.loc and df.at?
31. Where can you find a full list of methods supported by Pandas DataFrame and Series objects?
32. How do you find the sum of numbers in a column of dataframe?
33. How do you find the mean of numbers in a column of a dataframe?
34. How do you find the number of non-empty numbers in a column of a dataframe?
35. What is the result obtained by using a Pandas column in a boolean expression? Illustrate with an example.
36. How do you select a subset of rows where a specific column's value meets a given condition? Illustrate with an example.
37. What is the result of the expression df[df.new_cases > 100] ?
38. How do you display all the rows of a pandas dataframe in a Jupyter cell output?
39. What is the result obtained when you perform an arithmetic operation between two columns of a dataframe? Illustrate with an example.
40. How do you add a new column to a dataframe by combining values from two existing columns? Illustrate with an example.

41. How do you remove a column from a dataframe? Illustrate with an example.
42. What is the purpose of the `inplace` argument in dataframe methods?
43. How do you sort the rows of a dataframe based on the values in a particular column?
44. How do you sort a pandas dataframe using values from multiple columns?
45. How do you specify whether to sort by ascending or descending order while sorting a Pandas dataframe?
46. How do you change a specific value within a dataframe?
47. How do you convert a dataframe column to the `datetime` data type?
48. What are the benefits of using the `datetime` data type instead of `object`?
49. How do you extract different parts of a date column like the month, year, month, weekday, etc., into separate columns? Illustrate with an example.
50. How do you aggregate multiple columns of a dataframe together?
51. What is the purpose of the `groupby` method of a dataframe? Illustrate with an example.
52. What are the different ways in which you can aggregate the groups created by `groupby`?
53. What do you mean by a running or cumulative sum?
54. How do you create a new column containing the running or cumulative sum of another column?
55. What are other cumulative measures supported by Pandas dataframes?
56. What does it mean to merge two dataframes? Give an example.
57. How do you specify the columns that should be used for merging two dataframes?
58. How do you write data from a Pandas dataframe into a CSV file? Give an example.
59. What are some other file formats you can write to from a Pandas dataframe? Illustrate with examples.
60. How do you create a line plot showing the values within a column of dataframe?
61. How do you convert a column of a dataframe into its index?
62. Can the index of a dataframe be non-numeric?
63. What are the benefits of using a non-numeric dataframe? Illustrate with an example.
64. How you create a bar plot showing the values within a column of a dataframe?
65. What are some other types of plots supported by Pandas dataframes and series?

Assignment 3 - Pandas Data Analysis Practice

This assignment is a part of the course ["Data Analysis with Python: Zero to Pandas"](#)

In this assignment, you'll get to practice some of the concepts and skills covered in this tutorial:

<https://jovian.ai/aakashns/python-pandas-data-analysis>

As you go through this notebook, you will find a ??? in certain places. To complete this assignment, you must replace all the ??? with appropriate values, expressions or statements to ensure that the notebook runs properly end-to-end.

Some things to keep in mind:

- Make sure to run all the code cells, otherwise you may get errors like `NameError` for undefined variables.
- Do not change variable names, delete cells or disturb other existing code. It may cause problems during evaluation.
- In some cases, you may need to add some code cells or new statements before or after the line of code containing the ???.
- Since you'll be using a temporary online service for code execution, save your work by running `jovian.commit` at regular intervals.
- Questions marked (**Optional**) will not be considered for evaluation, and can be skipped. They are for your learning.

You can make submissions on this page: <https://jovian.ai/learn/data-analysis-with-python-zero-to-pandas/assignment/assignment-3-pandas-practice>

If you are stuck, you can ask for help on the community forum: <https://jovian.ai/forum/t/assignment-3-pandas-practice/11225/3>. You can get help with errors or ask for hints, describe your approach in simple words, link to documentation, but **please don't ask for or share the full working answer code** on the forum.

How to run the code and save your work

The recommended way to run this notebook is to click the "Run" button at the top of this page, and select "Run on Binder". This will run the notebook on mybinder.org, a free online service for running Jupyter notebooks.

Before starting the assignment, let's save a snapshot of the assignment to your [Jovian.ai](https://jovian.ai) profile, so that you can access it later, and continue your work.

```
import jovian
```

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

```
[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/pandas-practice-assignment
'https://jovian.ai/evanmarie/pandas-practice-assignment'
```

```
# Run the next line to install Pandas
!pip install pandas --upgrade
```


	location	continent	population	life_expectancy	hospital_beds_per_thousand	gdp_per_capita
3	Andorra	Europe	77265.0	83.73	NaN	NaN
4	Angola	Africa	32866268.0	61.15	NaN	5819.495
...
205	Vietnam	Asia	97338583.0	75.40	2.60	6171.884
206	Western Sahara	Africa	597330.0	70.26	NaN	NaN
207	Yemen	Asia	29825968.0	66.12	0.70	1479.147
208	Zambia	Africa	18383956.0	63.89	2.00	3689.251
209	Zimbabwe	Africa	14862927.0	61.49	1.70	1899.775

210 rows × 6 columns

Q1: How many countries does the dataframe contain?

Hint: Use the `.shape` method.

```
num_countries = countries_df.shape
print(f"This file shows {num_countries[0]} countries.")
```

This file shows 210 countries.

```
print('There are {} countries in the dataset'.format(num_countries[0]))
```

There are 210 countries in the dataset

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

```
[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/pandas-practice-assignment
'https://jovian.ai/evanmarie/pandas-practice-assignment'
```

```
countries_df.columns
```

```
Index(['location', 'continent', 'population', 'life_expectancy',
       'hospital_beds_per_thousand', 'gdp_per_capita'],
      dtype='object')
```

Q2: Retrieve a list of continents from the dataframe?

Hint: Use the `.unique` method of a series.

```
continents = countries_df.continent.unique()
```

```
continents
```

```
array(['Asia', 'Europe', 'Africa', 'North America', 'South America',
       'Oceania'], dtype=object)
```

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

```
[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on https://jovian.ai  
[jovian] Committed successfully! https://jovian.ai/evanmarie/pandas-practice-assignment  
'https://jovian.ai/evanmarie/pandas-practice-assignment'
```

Q3: What is the total population of all the countries listed in this dataset?

```
total_population = countries_df.population.sum()
```

```
print('The total population is {:,}'.format(int(total_population)))
```

The total population is 7,757,980,095.

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

```
[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on https://jovian.ai  
[jovian] Committed successfully! https://jovian.ai/evanmarie/pandas-practice-assignment  
'https://jovian.ai/evanmarie/pandas-practice-assignment'
```

Q: (Optional) What is the overall life expectancy across in the world?

Hint: You'll need to take a weighted average of life expectancy using populations as weights.

- $\text{overall_life_expectancy} = \frac{\text{sum}(\text{population of country} * \text{life expectancy of country})}{\text{total_population}}$
- use `np.average()` to take a weighted average
- ignore countries with missing life expectancy data
- use the `.dropna()` method to remove rows with missing values
- store the result of `.dropna()` in a new dataframe as not to lose the original dataframe
- `.dropna()` returns a new dataframe with missing values removed

```
countries_cleaned_df = countries_df.dropna()
```

```
overall_life_expectancy = np.average(countries_cleaned_df.life_expectancy, weights = cc  
print('The overall life expectancy across the world is {:.2f} years.'.format(overall_li
```

The overall life expectancy across the world is 73.63 years.

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

```
[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on https://jovian.ai  
[jovian] Committed successfully! https://jovian.ai/evanmarie/pandas-practice-assignment  
'https://jovian.ai/evanmarie/pandas-practice-assignment'
```

Q4: Create a dataframe containing 10 countries with the highest population.

Hint: Chain the `sort_values` and `head` methods.

```

most_populous_df = countries_df.sort_values(by='population',
                                             ascending=False).head(10)

print("The top 10 countries by population are: \n",
      most_populous_df, "\n")

```

The top 10 countries by population are:

	location	continent	population	life_expectancy \
41	China	Asia	1.439324e+09	76.91
90	India	Asia	1.380004e+09	69.66
199	United States	North America	3.310026e+08	78.86
91	Indonesia	Asia	2.735236e+08	71.72
145	Pakistan	Asia	2.208923e+08	67.27
27	Brazil	South America	2.125594e+08	75.88
141	Nigeria	Africa	2.061396e+08	54.69
15	Bangladesh	Asia	1.646894e+08	72.59
157	Russia	Europe	1.459345e+08	72.58
125	Mexico	North America	1.289328e+08	75.05

	hospital_beds_per_thousand	gdp_per_capita
41	4.34	15308.712
90	0.53	6426.674
199	2.77	54225.446
91	1.04	11188.744
145	0.60	5034.708
27	2.20	14103.452
141	NaN	5338.454
15	0.80	3523.984
157	8.05	24765.954
125	1.38	17336.469

```
most_populous_df
```

	location	continent	population	life_expectancy	hospital_beds_per_thousand	gdp_per_capita
41	China	Asia	1.439324e+09	76.91	4.34	15308.712
90	India	Asia	1.380004e+09	69.66	0.53	6426.674
199	United States	North America	3.310026e+08	78.86	2.77	54225.446
91	Indonesia	Asia	2.735236e+08	71.72	1.04	11188.744
145	Pakistan	Asia	2.208923e+08	67.27	0.60	5034.708
27	Brazil	South America	2.125594e+08	75.88	2.20	14103.452
141	Nigeria	Africa	2.061396e+08	54.69	NaN	5338.454

	location	continent	population	life_expectancy	hospital_beds_per_thousand	gdp_per_capita
15	Bangladesh	Asia	1.646894e+08	72.59	0.80	3523.984
157	Russia	Europe	1.459345e+08	72.58	8.05	24765.954
125	Mexico	North America	1.289328e+08	75.05	1.38	17336.469

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on <https://jovian.ai>
[jovian] Committed successfully! <https://jovian.ai/evanmarie/pandas-practice-assignment>
'<https://jovian.ai/evanmarie/pandas-practice-assignment>'

Q5: Add a new column in `countries_df` to record the overall GDP per country (product of population & per capita GDP).

```
gdp = countries_df.population * countries_df.gdp_per_capita
countries_df['gdp'] = gdp
```

```
countries_df
```

	location	continent	population	life_expectancy	hospital_beds_per_thousand	gdp_per_capita	gdp
0	Afghanistan	Asia	38928341.0	64.83	0.50	1803.987	7.022622e+10
1	Albania	Europe	2877800.0	78.57	2.89	11803.431	3.396791e+10
2	Algeria	Africa	43851043.0	76.88	1.90	13913.839	6.101364e+11
3	Andorra	Europe	77265.0	83.73	NaN	NaN	NaN
4	Angola	Africa	32866268.0	61.15	NaN	5819.495	1.912651e+11
...
205	Vietnam	Asia	97338583.0	75.40	2.60	6171.884	6.007624e+11
206	Western Sahara	Africa	597330.0	70.26	NaN	NaN	NaN
207	Yemen	Asia	29825968.0	66.12	0.70	1479.147	4.411699e+10
208	Zambia	Africa	18383956.0	63.89	2.00	3689.251	6.782303e+10
209	Zimbabwe	Africa	14862927.0	61.49	1.70	1899.775	2.823622e+10

210 rows × 7 columns

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on <https://jovian.ai>
[jovian] Committed successfully! <https://jovian.ai/evanmarie/pandas-practice-assignment>
'<https://jovian.ai/evanmarie/pandas-practice-assignment>'

Q: (Optional) Create a dataframe containing 10 countries with the lowest GDP per capita, among the counties with population greater than 100 million.

```
countries_low_gdp_df = countries_df[countries_df.population > 100000000].sort_values(by
print("The 10 countries > 100 mil pop with lowest GDP per capita: \n", countries_low_gc
```

The 10 countries > 100 mil pop with lowest GDP per capita:

	location	continent	population	life_expectancy	\
63	Ethiopia	Africa	1.149636e+08	66.60	
15	Bangladesh	Asia	1.646894e+08	72.59	
145	Pakistan	Asia	2.208923e+08	67.27	
141	Nigeria	Africa	2.061396e+08	54.69	
90	India	Asia	1.380004e+09	69.66	
151	Philippines	Asia	1.095811e+08	71.23	
58	Egypt	Africa	1.023344e+08	71.99	
91	Indonesia	Asia	2.735236e+08	71.72	
27	Brazil	South America	2.125594e+08	75.88	
41	China	Asia	1.439324e+09	76.91	

	hospital_beds_per_thousand	gdp_per_capita	gdp
63	0.30	1729.927	1.988786e+11
15	0.80	3523.984	5.803628e+11
145	0.60	5034.708	1.112128e+12
141	NaN	5338.454	1.100467e+12
90	0.53	6426.674	8.868838e+12
151	1.00	7599.188	8.327273e+11
58	1.60	10550.206	1.079649e+12
91	1.04	11188.744	3.060386e+12
27	2.20	14103.452	2.997821e+12
41	4.34	15308.712	2.203419e+13

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

```
[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/pandas-practice-assignment
'https://jovian.ai/evanmarie/pandas-practice-assignment'
```

Q6: Create a data frame that counts the number countries in each continent?

Hint: Use groupby, select the location column and aggregate using count.

```
country_counts_df = countries_df.groupby('continent').location.count()
print("The number of countries in each continent are: \n", country_counts_df, "\n")
```

The number of countries in each continent are:

```
continent
Africa      55
Asia        47
Europe      51
North America 36
Oceania      8
South America 13
Name: location, dtype: int64
```

```
country_counts_df
```

```
continent
Africa      55
Asia        47
Europe      51
North America 36
Oceania      8
South America 13
Name: location, dtype: int64
```

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

```
[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/pandas-practice-assignment
'https://jovian.ai/evanmarie/pandas-practice-assignment'
```

Q7: Create a data frame showing the total population of each continent.

Hint: Use `groupby`, select the population column and aggregate using `sum`.

```
continent_populations_df = countries_df.groupby('continent').population.sum()
print("The total population of each continent are: \n", continent_populations_df, "\n")
```

The total population of each continent are:

```
continent
Africa      1.339424e+09
Asia        4.607388e+09
Europe      7.485062e+08
North America 5.912425e+08
Oceania      4.095832e+07
South America 4.304611e+08
Name: population, dtype: float64
```

```
continent_populations_df
```

```
continent
Africa      1.339424e+09
Asia        4.607388e+09
Europe      7.485062e+08
North America 5.912425e+08
Oceania     4.095832e+07
South America 4.304611e+08
Name: population, dtype: float64
```

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

```
[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/pandas-practice-assignment
'https://jovian.ai/evanmarie/pandas-practice-assignment'
```

Let's download another CSV file containing overall Covid-19 stats for various countries, and read the data into another Pandas data frame.

```
urlretrieve('https://gist.githubusercontent.com/aakashns/b2a968a6cfd9fbbb0ff3d6bd0f2626
'covid-countries-data.csv')
```

```
('covid-countries-data.csv', <http.client.HTTPMessage at 0x7f5f1c98b4c0>)
```

```
covid_data_df = pd.read_csv('covid-countries-data.csv')
```

```
covid_data_df
```

	location	total_cases	total_deaths	total_tests
0	Afghanistan	38243.0	1409.0	NaN
1	Albania	9728.0	296.0	NaN
2	Algeria	45158.0	1525.0	NaN
3	Andorra	1199.0	53.0	NaN
4	Angola	2729.0	109.0	NaN
...
207	Western Sahara	766.0	1.0	NaN
208	World	26059065.0	863535.0	NaN
209	Yemen	1976.0	571.0	NaN
210	Zambia	12415.0	292.0	NaN
211	Zimbabwe	6638.0	206.0	97272.0

212 rows x 4 columns

Q8: Count the number of countries for which the total_tests data is missing.

Hint: Use the .isna method.

```
total_tests_missing = total_tests_missing = covid_data_df.total_tests.isna().sum()
print(f'The number of countries for which the total_tests data is missing is {total_tests_missing}')
```

The number of countries for which the total_tests data is missing is 122.

```
print("The data for total tests is missing for {} countries.".format(int(total_tests_missing)))
```

The data for total tests is missing for 122 countries.

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

```
[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/pandas-practice-assignment
'https://jovian.ai/evanmarie/pandas-practice-assignment'
```

Let's merge the two data frames, and compute some more metrics.

Q9: Merge countries_df with covid_data_df on the location column.

*Hint: Use the .merge method on countries_df.

```
combined_df = covid_data_df.merge(countries_df, on="location")
```

```
combined_df
```

	location	total_cases	total_deaths	total_tests	continent	population	life_expectancy	hospital_beds_per_thou
0	Afghanistan	38243.0	1409.0	NaN	Asia	38928341.0	64.83	
1	Albania	9728.0	296.0	NaN	Europe	2877800.0	78.57	
2	Algeria	45158.0	1525.0	NaN	Africa	43851043.0	76.88	
3	Andorra	1199.0	53.0	NaN	Europe	77265.0	83.73	
4	Angola	2729.0	109.0	NaN	Africa	32866268.0	61.15	
...
205	Vietnam	1046.0	35.0	261004.0	Asia	97338583.0	75.40	
206	Western Sahara	766.0	1.0	NaN	Africa	597330.0	70.26	
207	Yemen	1976.0	571.0	NaN	Asia	29825968.0	66.12	
208	Zambia	12415.0	292.0	NaN	Africa	18383956.0	63.89	
209	Zimbabwe	6638.0	206.0	97272.0	Africa	14862927.0	61.49	

210 rows × 10 columns

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

```
[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/pandas-practice-assignment
```

'<https://jovian.ai/evanmarie/pandas-practice-assignment>'

Q10: Add columns `tests_per_million`, `cases_per_million` and `deaths_per_million` into `combined_df`.

```
combined_df['tests_per_million'] = combined_df['total_tests'] * 1e6 / combined_df['popu
```

```
combined_df['cases_per_million'] = combined_df.total_cases / combined_df.population * 1
```

```
combined_df['deaths_per_million'] = combined_df.total_deaths / combined_df.population *
```

combined_df

	location	total_cases	total_deaths	total_tests	continent	population	life_expectancy	hospital_beds_per_thou
0	Afghanistan	38243.0	1409.0	NaN	Asia	38928341.0	64.83	
1	Albania	9728.0	296.0	NaN	Europe	2877800.0	78.57	
2	Algeria	45158.0	1525.0	NaN	Africa	43851043.0	76.88	
3	Andorra	1199.0	53.0	NaN	Europe	77265.0	83.73	
4	Angola	2729.0	109.0	NaN	Africa	32866268.0	61.15	
...
205	Vietnam	1046.0	35.0	261004.0	Asia	97338583.0	75.40	
206	Western Sahara	766.0	1.0	NaN	Africa	597330.0	70.26	
207	Yemen	1976.0	571.0	NaN	Asia	29825968.0	66.12	
208	Zambia	12415.0	292.0	NaN	Africa	18383956.0	63.89	
209	Zimbabwe	6638.0	206.0	97272.0	Africa	14862927.0	61.49	

210 rows × 13 columns

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on <https://jovian.ai>
[jovian] Committed successfully! <https://jovian.ai/evanmarie/pandas-practice-assignment>

'<https://jovian.ai/evanmarie/pandas-practice-assignment>'

Q11: Create a dataframe with 10 countries that have highest number of tests per million people.

```
highest_tests_df = combined_df.sort_values(by='tests_per_million', ascending=False).head(10)
```

highest_tests_df

	location	total_cases	total_deaths	total_tests	continent	population	life_expectancy	hospital_beds_per_t
197	United Arab Emirates	71540.0	387.0	7177430.0	Asia	9890400.0	77.97	
14	Bahrain	52440.0	190.0	1118837.0	Asia	1701583.0	77.29	
115	Luxembourg	7928.0	124.0	385820.0	Europe	625976.0	82.25	
122	Malta	1931.0	13.0	188539.0	Europe	441539.0	82.53	
53	Denmark	17195.0	626.0	2447911.0	Europe	5792203.0	80.90	
96	Israel	122539.0	969.0	2353984.0	Asia	8655541.0	82.97	
89	Iceland	2121.0	10.0	88829.0	Europe	341250.0	82.99	
157	Russia	1005000.0	17414.0	37176827.0	Europe	145934460.0	72.58	
199	United States	6114406.0	185744.0	83898416.0	North America	331002647.0	78.86	
10	Australia	25923.0	663.0	6255797.0	Oceania	25499881.0	83.44	

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on <https://jovian.ai>
[jovian] Committed successfully! <https://jovian.ai/evanmarie/pandas-practice-assignment>
'<https://jovian.ai/evanmarie/pandas-practice-assignment>'

Q12: Create a dataframe with 10 countries that have highest number of positive cases per million people.

```
highest_cases_df = combined_df.sort_values(by='cases_per_million', ascending=False).head(10)
```

```
highest_cases_df
```

	location	total_cases	total_deaths	total_tests	continent	population	life_expectancy	hospital_beds_per_thou
155	Qatar	119206.0	199.0	634745.0	Asia	2881060.0	80.23	
14	Bahrain	52440.0	190.0	1118837.0	Asia	1701583.0	77.29	
147	Panama	94084.0	2030.0	336345.0	North America	4314768.0	78.51	
40	Chile	414739.0	11344.0	2458762.0	South America	19116209.0	80.18	
162	San Marino	735.0	42.0	NaN	Europe	33938.0	84.97	
9	Aruba	2211.0	12.0	NaN	North America	106766.0	76.29	
105	Kuwait	86478.0	535.0	621616.0	Asia	4270563.0	75.49	
150	Peru	663437.0	29259.0	584232.0	South America	32971846.0	76.74	
27	Brazil	3997865.0	123780.0	4797948.0	South America	212559409.0	75.88	
199	United States	6114406.0	185744.0	83898416.0	North America	331002647.0	78.86	

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

```
[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on https://jovian.ai  
[jovian] Committed successfully! https://jovian.ai/evanmarie/pandas-practice-assignment  
'https://jovian.ai/evanmarie/pandas-practice-assignment'
```

Q13: Create a dataframe with 10 countries that have highest number of deaths cases per million people?

```
highest_deaths_df = combined_df.sort_values(by='deaths_per_million', ascending = False)
```

```
highest_deaths_df
```

	location	total_cases	total_deaths	total_tests	continent	population	life_expectancy	hospital_beds_per_thou
162	San Marino	735.0	42.0	NaN	Europe	33938.0	84.97	
150	Peru	663437.0	29259.0	584232.0	South America	32971846.0	76.74	
18	Belgium	85817.0	9898.0	2281853.0	Europe	11589616.0	81.63	
3	Andorra	1199.0	53.0	NaN	Europe	77265.0	83.73	
177	Spain	479554.0	29194.0	6416533.0	Europe	46754783.0	83.56	
198	United Kingdom	338676.0	41514.0	13447568.0	Europe	67886004.0	81.32	
40	Chile	414739.0	11344.0	2458762.0	South America	19116209.0	80.18	
97	Italy	271515.0	35497.0	5214766.0	Europe	60461828.0	83.51	
27	Brazil	3997865.0	123780.0	4797948.0	South America	212559409.0	75.88	
182	Sweden	84532.0	5820.0	NaN	Europe	10099270.0	82.80	

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

```
[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on https://jovian.ai  
[jovian] Committed successfully! https://jovian.ai/evanmarie/pandas-practice-assignment  
'https://jovian.ai/evanmarie/pandas-practice-assignment'
```

(Optional) Q: Count number of countries that feature in both the lists of "highest number of tests per million" and "highest number of cases per million".

```
too_many_highs = highest_tests_df.location.isin(highest_cases_df.location).sum()  
  
print(f'The number of countries that feature in both the lists of \n'  
      f'"highest number of tests per million" and "highest number \n'  
      f'of cases per million" is {too_many_highs}.')
```

The number of countries that feature in both the lists of

"highest number of tests per million" and "highest number of cases per million" is 2.

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

(Optional) Q: Count number of countries that feature in both the lists "20 countries with lowest GDP per capita" and "20 countries with the lowest number of hospital beds per thousand population". Only consider countries with a population higher than 10 million while creating the list.

```
lowest_20_beds_df = combined_df[combined_df.population > 10000000].sort_values(by='hosp
```

```
lowest_gdp_df = combined_df[combined_df.population>10000000].sort_values(by='gdp_per_ca
```

```
unacceptable = lowest_20_beds_df.location.isin(lowest_gdp_df.location).sum()
```

```
print(f'The number of countries that feature in both the lists \n'  
      f'"20 countries with lowest GDP per capita" and "20 countries \n'  
      f'with the lowest number of hospital beds per thousand population" \n'  
      f'is {unacceptable}.')
```

The number of countries that feature in both the lists "20 countries with lowest GDP per capita" and "20 countries with the lowest number of hospital beds per thousand population" is 14.

```
import jovian
```

```
jovian.commit(project='pandas-practice-assignment', environment=None)
```

```
[jovian] Updating notebook "evanmarie/pandas-practice-assignment" on https://jovian.ai  
[jovian] Committed successfully! https://jovian.ai/evanmarie/pandas-practice-assignment  
'https://jovian.ai/evanmarie/pandas-practice-assignment'
```

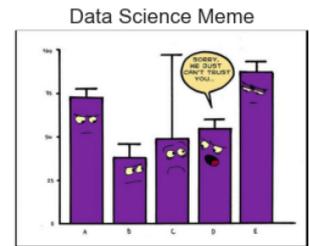
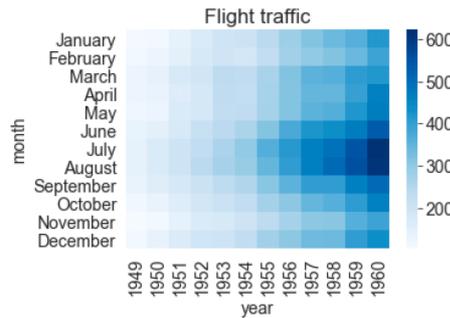
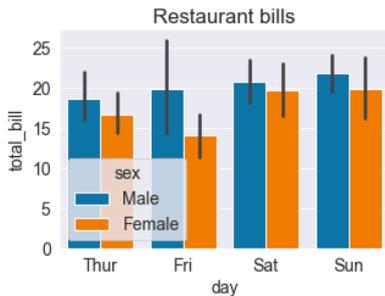
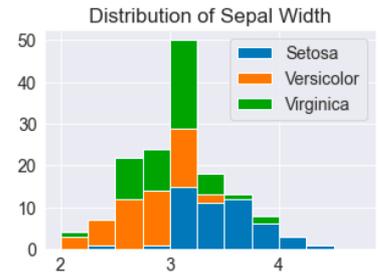
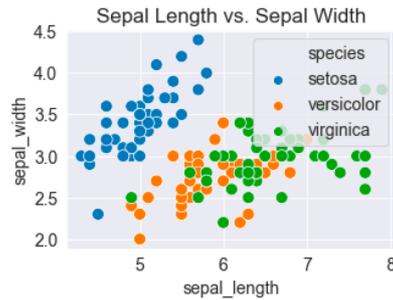
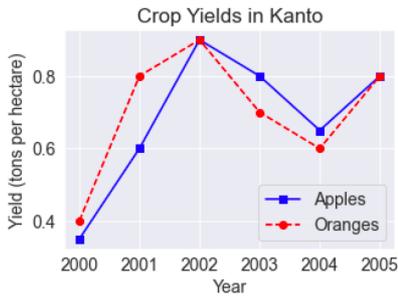
Submission

Congratulations on making it this far! You've reached the end of this assignment, and you just completed your first real-world data analysis problem. It's time to record one final version of your notebook for submission.

Make a submission here by filling the submission form: <https://jovian.ai/learn/data-analysis-with-python-zero-to-pandas/assignment/assignment-3-pandas-practice>

Also make sure to help others on the forum: <https://jovian.ai/forum/t/assignment-3-pandas-practice/11225/2>

Data Visualization using Python, Matplotlib and Seaborn



Part 8 of "Data Analysis with Python: Zero to Pandas"

This tutorial series is a beginner-friendly introduction to programming and data analysis using the Python programming language. These tutorials take a practical and coding-focused approach. The best way to learn the material is to execute the code and experiment with it yourself. Check out the full series here:

- [1. First Steps with Python and Jupyter](#)
- [2. A Quick Tour of Variables and Data Types](#)
- [3. Branching using Conditional Statements and Loops](#)
- [4. Writing Reusable Code Using Functions](#)
- [5. Reading from and Writing to Files](#)
- [6. Numerical Computing with Python and Numpy](#)
- [7. Analyzing Tabular Data using Pandas](#)
- [8. Data Visualization using Matplotlib & Seaborn](#)
- [9. Exploratory Data Analysis - A Case Study](#)

This tutorial covers the following topics:

- Creating and customizing line charts using Matplotlib
- Visualizing relationships between two or more variables using scatter plots
- Studying distributions of variables using histograms & bar charts to
- Visualizing two-dimensional data using heatmaps
- Displaying images using Matplotlib's `plt.imshow`
- Plotting multiple Matplotlib and Seaborn charts in a grid

How to run the code

This tutorial is an executable [Jupyter notebook](#) hosted on [Jovian](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Jupyter Notebooks: This tutorial is a [Jupyter notebook](#) - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc., instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Introduction

Data visualization is the graphic representation of data. It involves producing images that communicate relationships among the represented data to viewers. Visualizing data is an essential part of data analysis and machine learning. We'll use Python libraries [Matplotlib](#) and [Seaborn](#) to learn and apply some popular data visualization techniques. We'll use the words *chart*, *plot*, and *graph* interchangeably in this tutorial.

To begin, let's install and import the libraries. We'll use the `matplotlib.pyplot` module for basic plots like line & bar charts. It is often imported with the alias `plt`. We'll use the `seaborn` module for more advanced plots. It is commonly imported with the alias `sns`.

```
!pip install matplotlib seaborn --upgrade --quiet
```

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

Notice this we also include the special command `%matplotlib inline` to ensure that our plots are shown and embedded within the Jupyter notebook itself. Without this command, sometimes plots may show up in pop-up windows.

Line Chart

The line chart is one of the simplest and most widely used data visualization techniques. A line chart displays information as a series of data points or markers connected by straight lines. You can customize the shape, size, color, and other aesthetic elements of the lines and markers for better visual clarity.

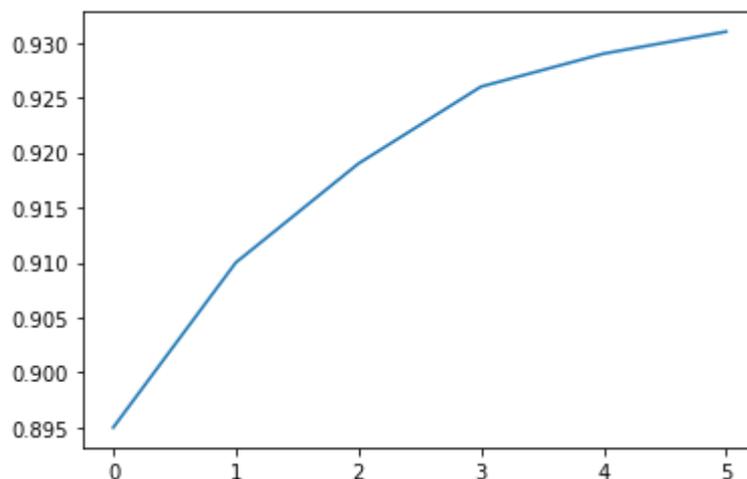
Here's a Python list showing the yield of apples (tons per hectare) over six years in an imaginary country called Kanto.

```
yield_apples = [0.895, 0.91, 0.919, 0.926, 0.929, 0.931]
```

We can visualize how the yield of apples changes over time using a line chart. To draw a line chart, we can use the `plt.plot` function.

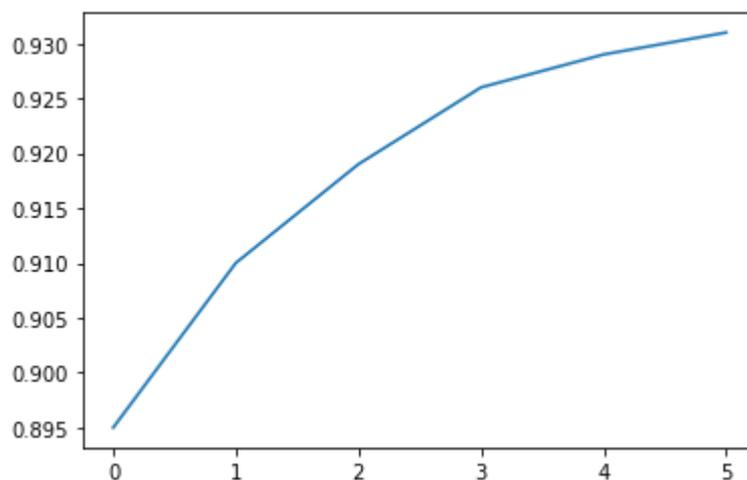
```
plt.plot(yield_apples)
```

```
[<matplotlib.lines.Line2D at 0x7f3c270d8af0>]
```



Calling the `plt.plot` function draws the line chart as expected. It also returns a list of plots drawn [`<matplotlib.lines.Line2D at 0x7ff70aa20760>`], shown within the output. We can include a semicolon (`;`) at the end of the last statement in the cell to avoid showing the output and display just the graph.

```
plt.plot(yield_apples);
```

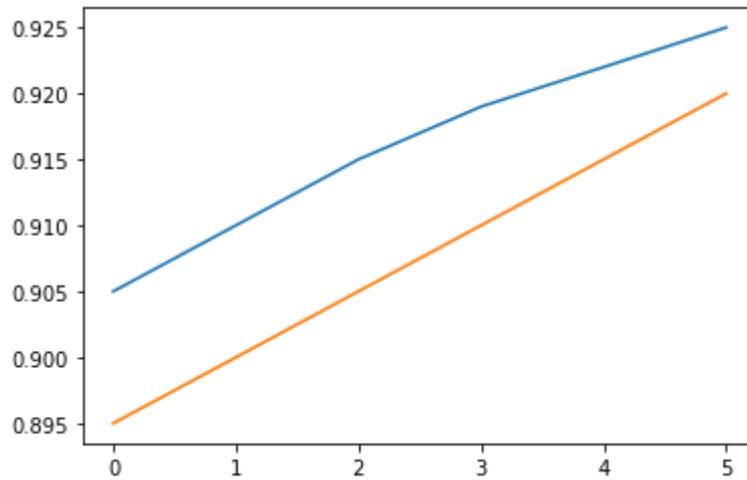


Let's enhance this plot step-by-step to make it more informative and beautiful.

```
yield_oranges = [0.905, 0.91, 0.915, 0.919, 0.922, 0.925]  
yield_bananas = [0.895, 0.9, 0.905, 0.91, 0.915, 0.92]
```

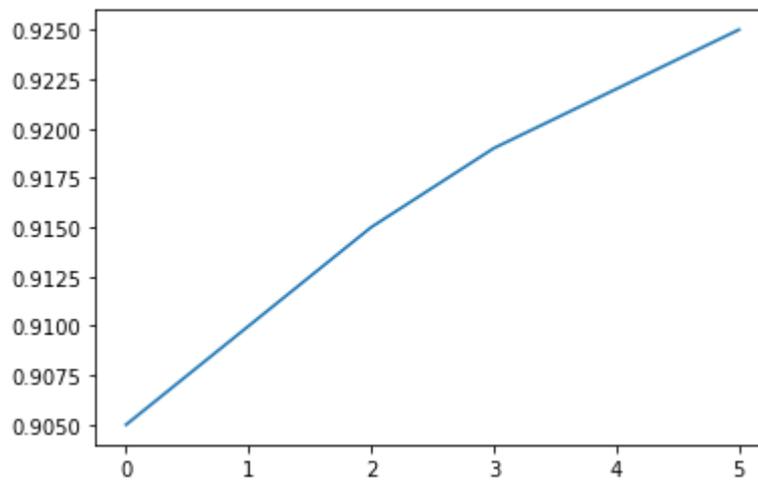
```
# Oranges and bananas plotted together:
```

```
plt.plot(yield_oranges);  
plt.plot(yield_bananas);
```



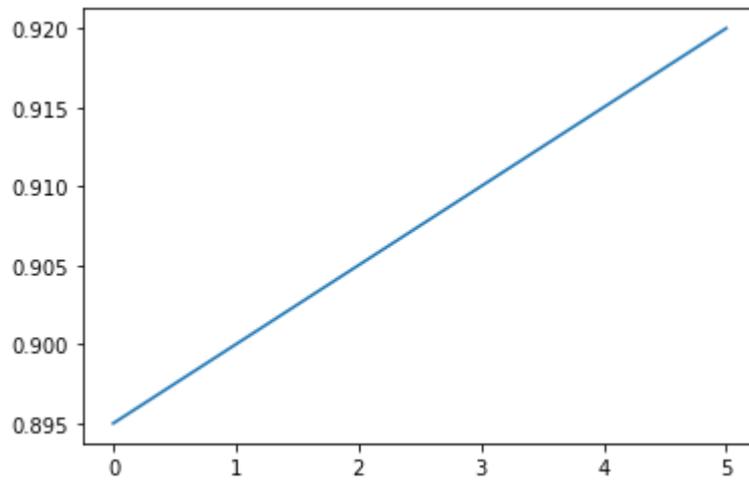
```
# Oranges plotted alone
```

```
plt.plot(yield_oranges);
```



```
# Bananas plotted alone
```

```
plt.plot(yield_bananas);
```

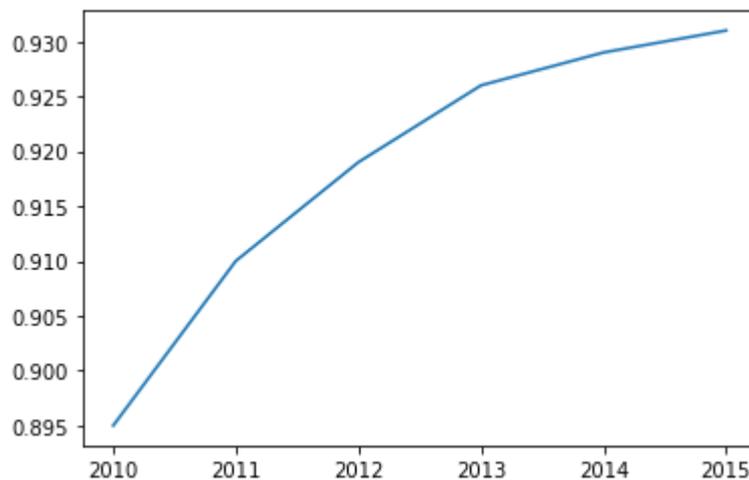


Customizing the X-axis

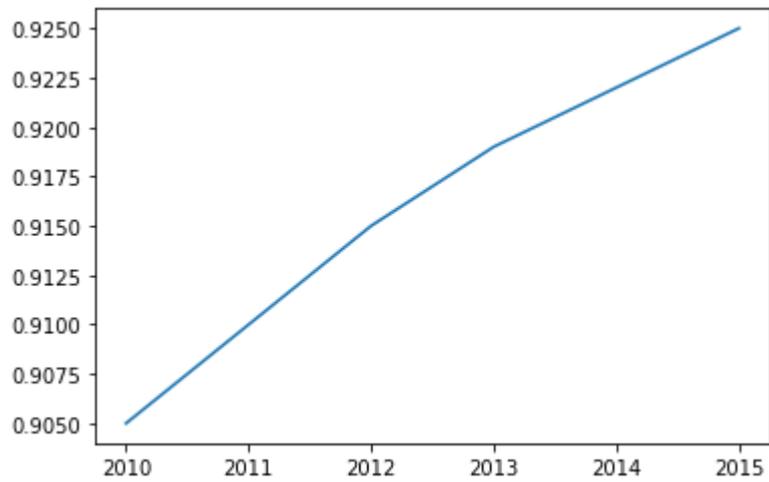
The X-axis of the plot currently shows list element indexes 0 to 5. The plot would be more informative if we could display the year for which we're plotting the data. We can do this by two arguments `plt.plot` .

```
years = [2010, 2011, 2012, 2013, 2014, 2015]
yield_apples = [0.895, 0.91, 0.919, 0.926, 0.929, 0.931]
yield_oranges = [0.905, 0.91, 0.915, 0.919, 0.922, 0.925]
yield_bananas = [0.895, 0.9, 0.905, 0.91, 0.915, 0.92]
```

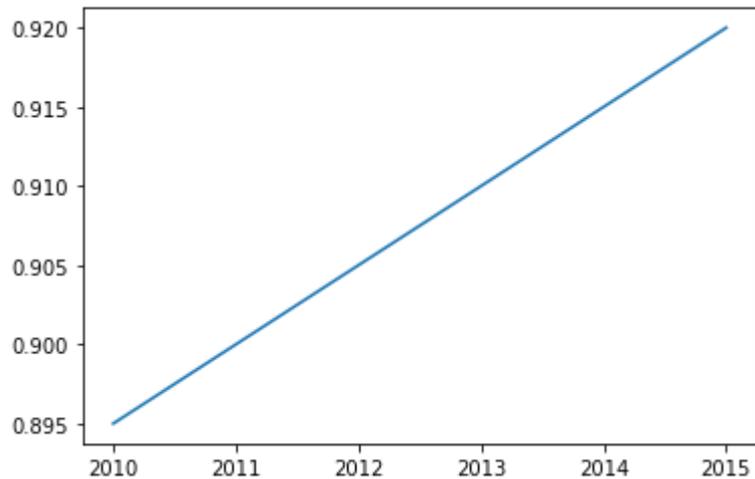
```
plt.plot(years, yield_apples);
```



```
plt.plot(years, yield_oranges);
```

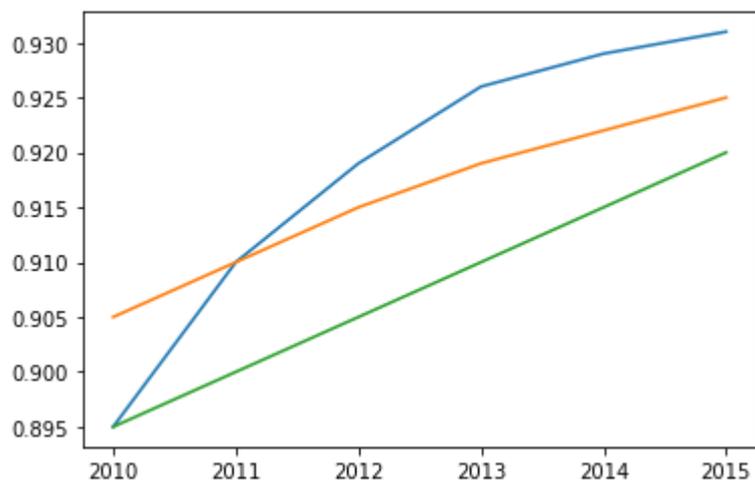


```
plt.plot(years, yield_bananas);
```



All Three!

```
plt.plot(years, yield_apples);  
plt.plot(years, yield_oranges);  
plt.plot(years, yield_bananas);
```



Axis Labels

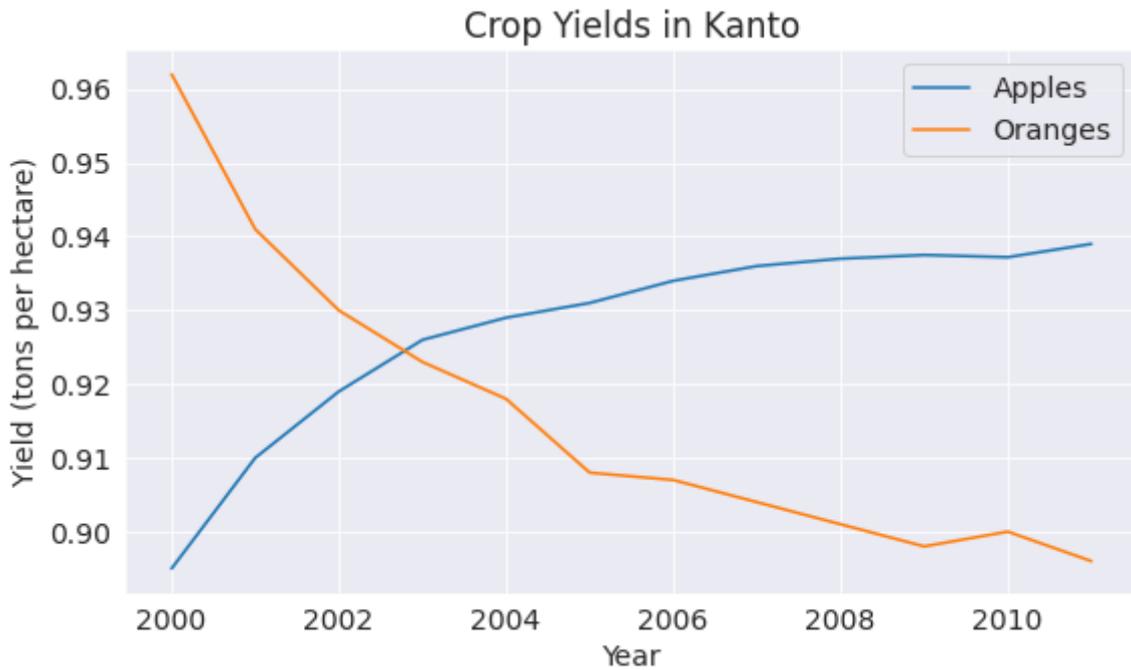
We can add labels to the axes to show what each axis represents using the `plt.xlabel` and `plt.ylabel` methods.

To differentiate between multiple lines, we can include a legend within the graph using the `plt.legend` function. We can also set a title for the chart using the `plt.title` function.

```
plt.plot(years, apples)
plt.plot(years, oranges)

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```



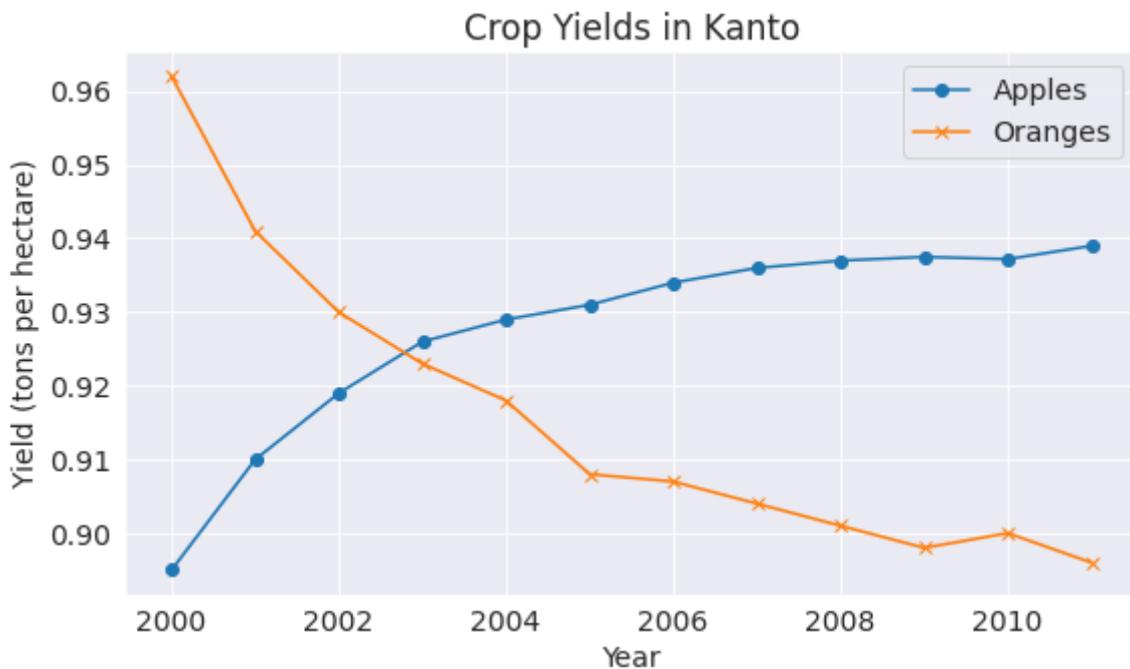
Line Markers

We can also show markers for the data points on each line using the `marker` argument of `plt.plot`. Matplotlib provides many different markers, like a circle, cross, square, diamond, etc. You can find the full list of marker types here: https://matplotlib.org/3.1.1/api/markers_api.html.

```
plt.plot(years, apples, marker='o')
plt.plot(years, oranges, marker='x')

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```



Styling Lines and Markers

The `plt.plot` function supports many arguments for styling lines and markers:

- `color` or `c`: Set the color of the line ([supported colors](#))
- `linestyle` or `ls`: Choose between a solid or dashed line
- `linewidth` or `lw`: Set the width of a line
- `markersize` or `ms`: Set the size of markers
- `markeredgecolor` or `mec`: Set the edge color for markers
- `markeredgewidth` or `mew`: Set the edge width for markers
- `markerfacecolor` or `mfc`: Set the fill color for markers
- `alpha`: Opacity of the plot

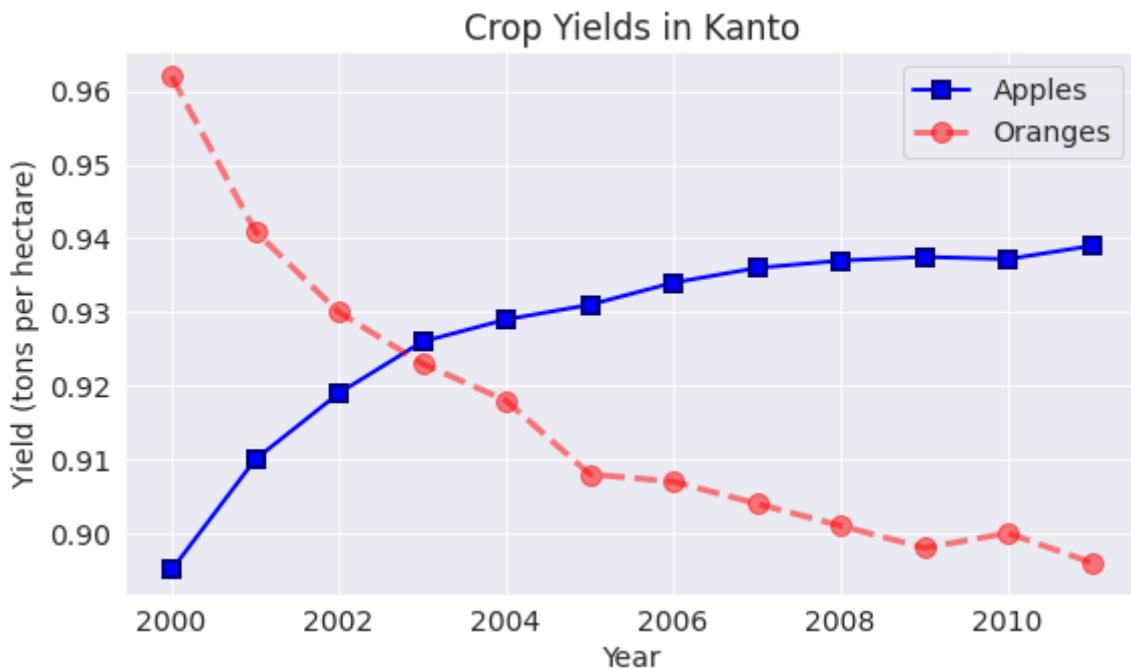
Check out the documentation for `plt.plot` to learn more:

https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot .

```
plt.plot(years, apples, marker='s', c='b', ls='-', lw=2, ms=8, mew=2, mec='navy')
plt.plot(years, oranges, marker='o', c='r', ls='--', lw=3, ms=10, alpha=.5)

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```



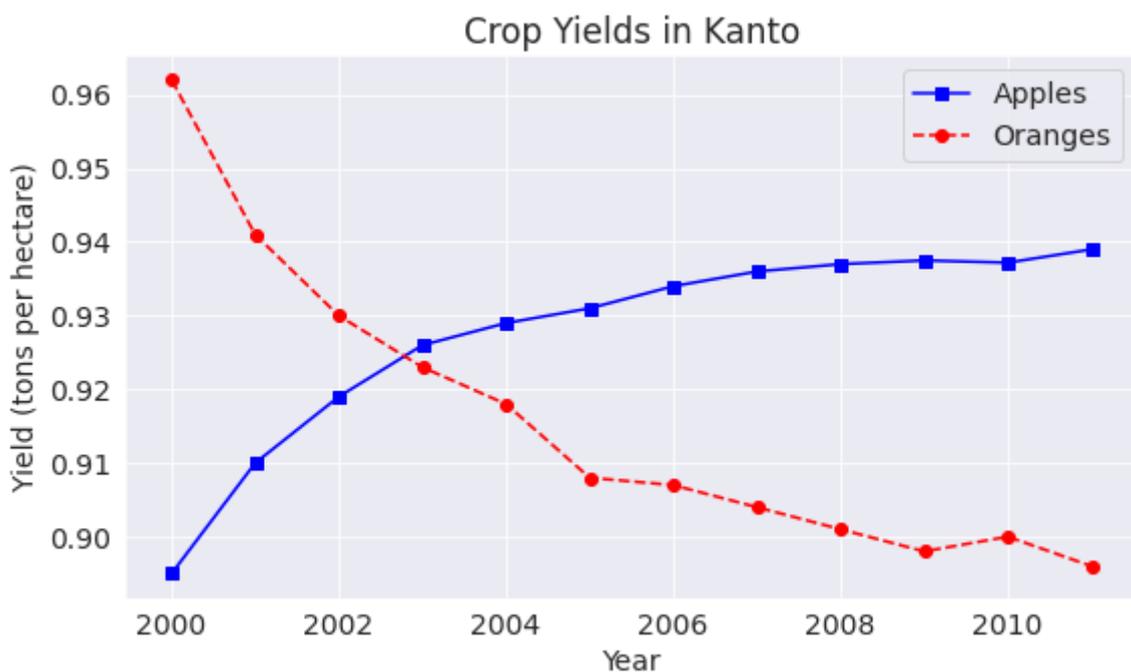
The `fmt` argument provides a shorthand for specifying the marker shape, line style, and line color. It can be provided as the third argument to `plt.plot`.

```
fmt = '[marker][line][color]'
```

```
plt.plot(years, apples, 's-b')
plt.plot(years, oranges, 'o--r')

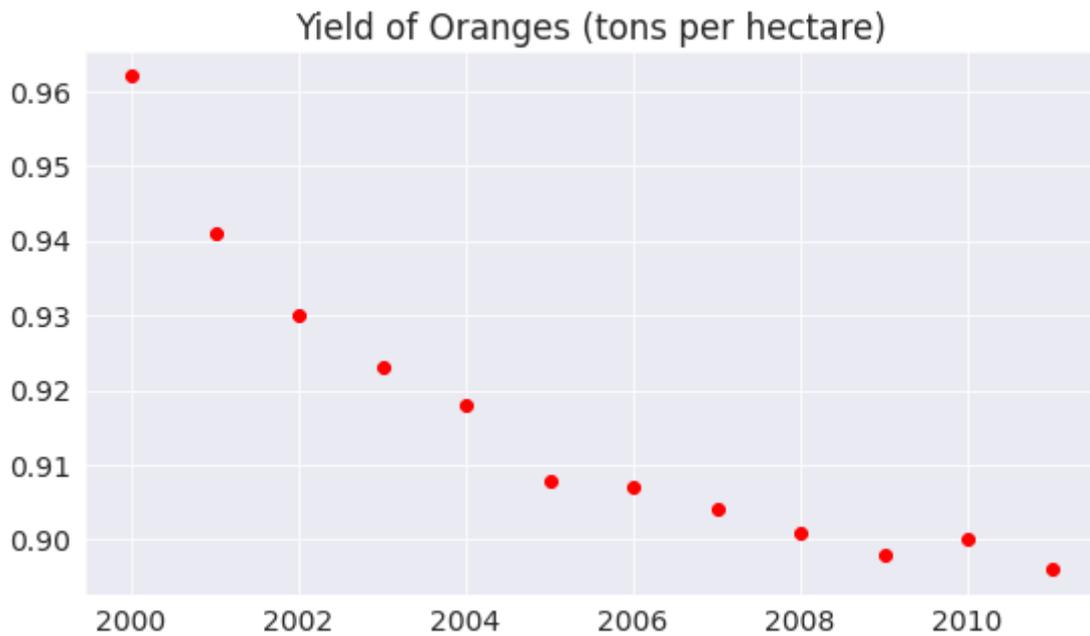
plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```



If you don't specify a line style in `fmt`, only markers are drawn.

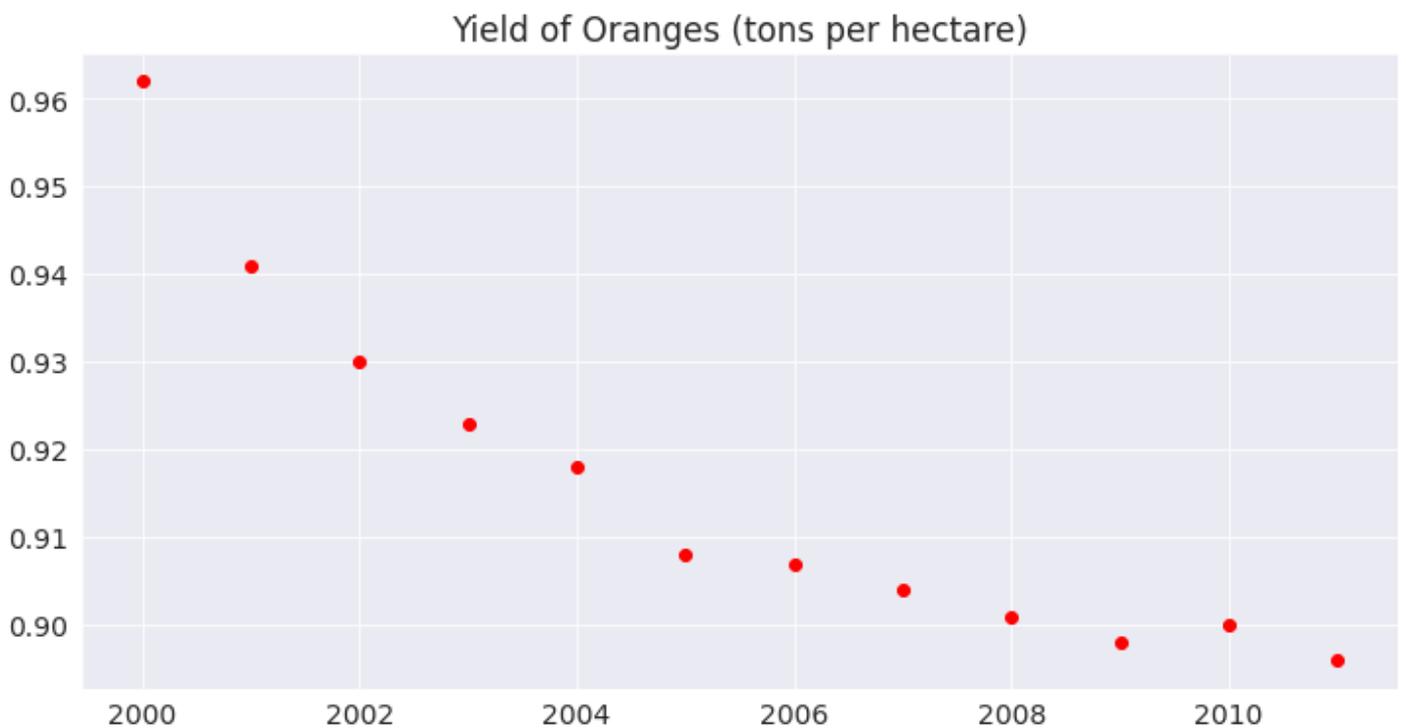
```
plt.plot(years, oranges, 'or')
plt.title("Yield of Oranges (tons per hectare)");
```



Changing the Figure Size

You can use the `plt.figure` function to change the size of the figure.

```
plt.figure(figsize=(12, 6))
plt.plot(years, oranges, 'or')
plt.title("Yield of Oranges (tons per hectare)");
```



Improving Default Styles using Seaborn

An easy way to make your charts look beautiful is to use some default styles from the Seaborn library. These can be applied globally using the `sns.set_style` function. You can see a full list of predefined styles here:

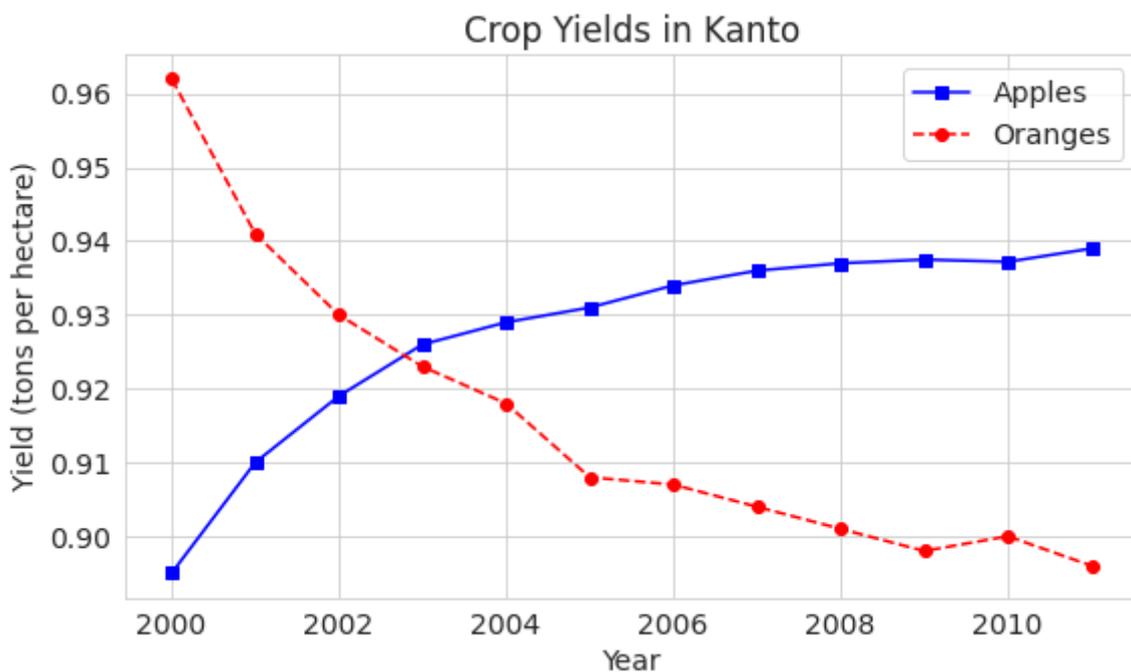
https://seaborn.pydata.org/generated/seaborn.set_style.html .

```
sns.set_style("whitegrid")
```

```
plt.plot(years, apples, 's-b')
plt.plot(years, oranges, 'o--r')

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```

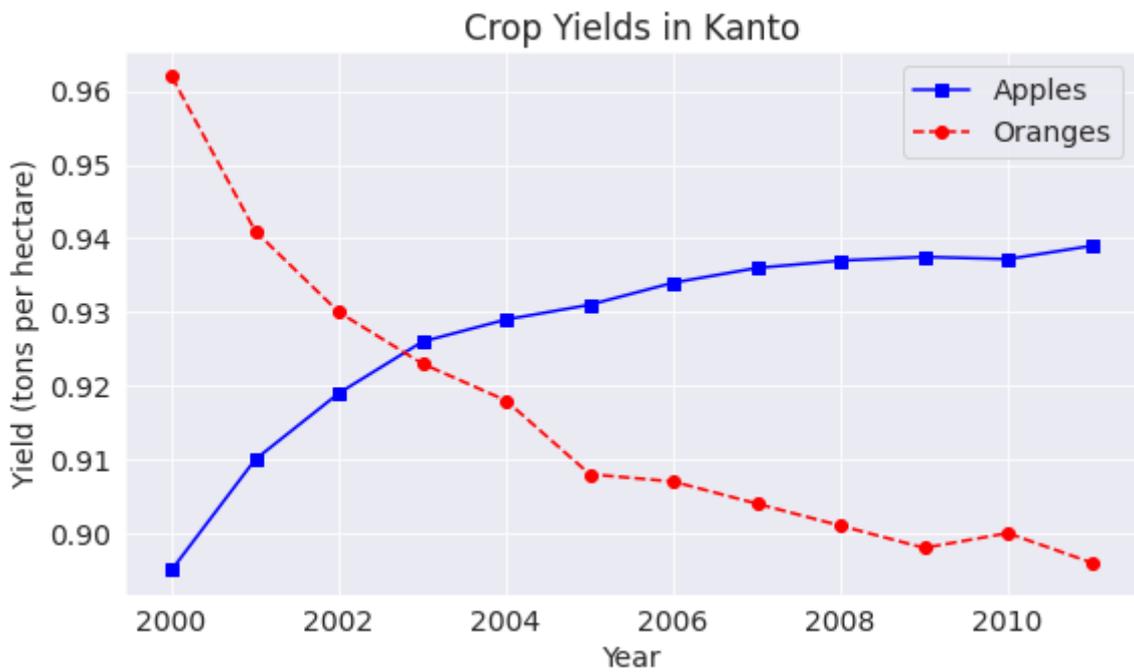


```
sns.set_style("darkgrid")
```

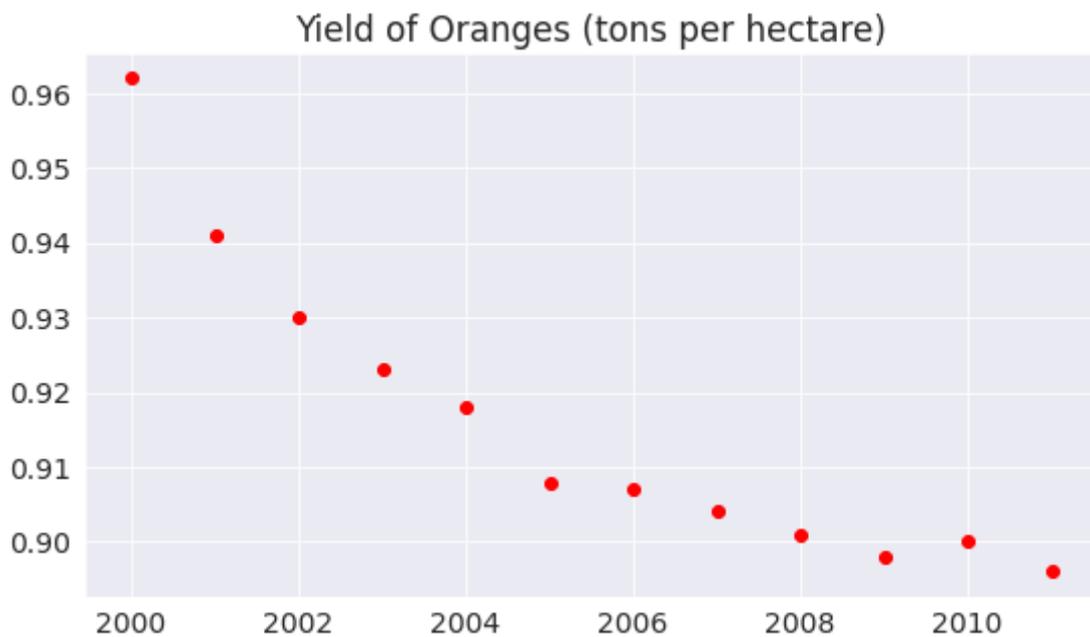
```
plt.plot(years, apples, 's-b')
plt.plot(years, oranges, 'o--r')

plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```



```
plt.plot(years, oranges, 'or')
plt.title("Yield of Oranges (tons per hectare)");
```



You can also edit default styles directly by modifying the `matplotlib.rcParams` dictionary. Learn more: <https://matplotlib.org/3.2.1/tutorials/introductory/customizing.html#matplotlib-rcparams>.

```
import matplotlib
```

```
matplotlib.rcParams['font.size'] = 14
matplotlib.rcParams['figure.figsize'] = (9, 5)
matplotlib.rcParams['figure.facecolor'] = '#00000000'
```

Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. [Jovian](#) offers an easy way of saving and sharing your Jupyter notebooks online.

```
# Install the library  
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit(project='python-matplotlib-data-visualization')
```

```
[jovian] Updating notebook "evanmarie/python-matplotlib-data-visualization" on  
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-matplotlib-data-visualization
```

```
'https://jovian.ai/evanmarie/python-matplotlib-data-visualization'
```

The first time you run `jovian.commit`, you'll be asked to provide an API Key to securely upload the notebook to your Jovian account. You can get the API key from your [Jovian profile page](#) after logging in / signing up.

`jovian.commit` uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work.

Scatter Plot

In a scatter plot, the values of 2 variables are plotted as points on a 2-dimensional grid. Additionally, you can also use a third variable to determine the size or color of the points. Let's try out an example.

The [Iris flower dataset](#) provides sample measurements of sepals and petals for three species of flowers. The Iris dataset is included with the Seaborn library and can be loaded as a Pandas data frame.

```
# Load data into a Pandas dataframe  
flowers_df = sns.load_dataset("iris")
```

```
flowers_df
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica

	sepal_length	sepal_width	petal_length	petal_width	species
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

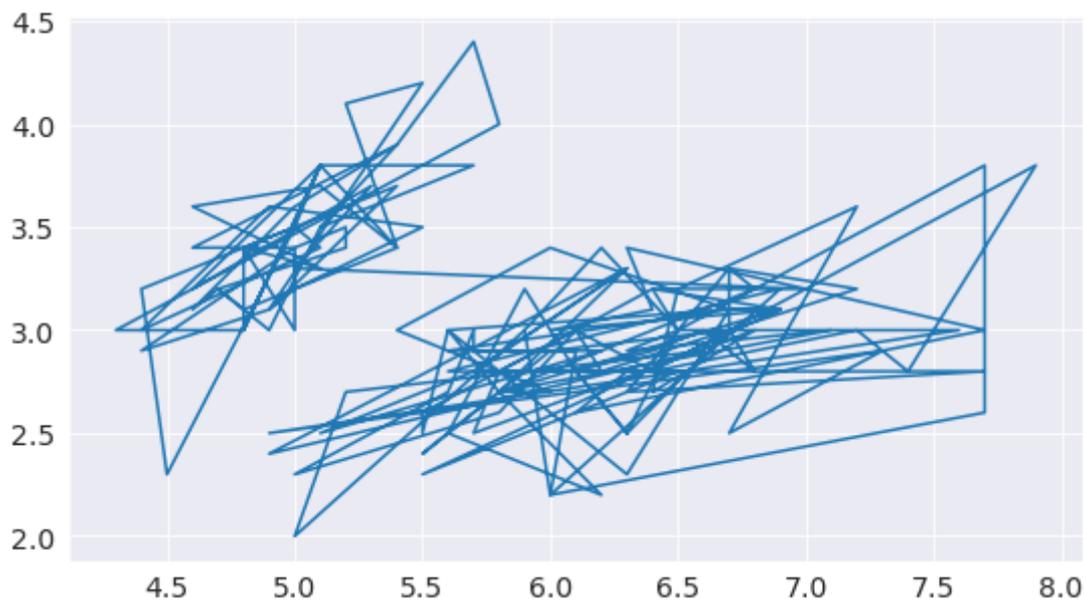
150 rows × 5 columns

```
flowers_df.species.unique()
```

```
array(['setosa', 'versicolor', 'virginica'], dtype=object)
```

Let's try to visualize the relationship between sepal length and sepal width. Our first instinct might be to create a line chart using `plt.plot`.

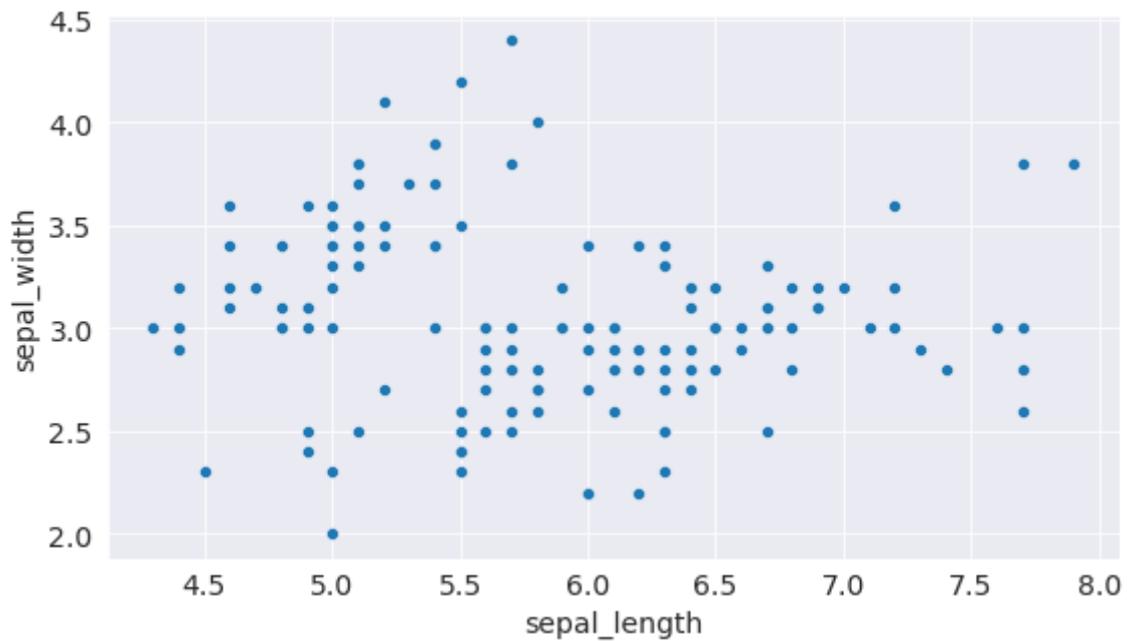
```
plt.plot(flowers_df.sepal_length, flowers_df.sepal_width);
```



The output is not very informative as there are too many combinations of the two properties within the dataset. There doesn't seem to be simple relationship between them.

We can use a scatter plot to visualize how sepal length & sepal width vary using the `scatterplot` function from the `seaborn` module (imported as `sns`).

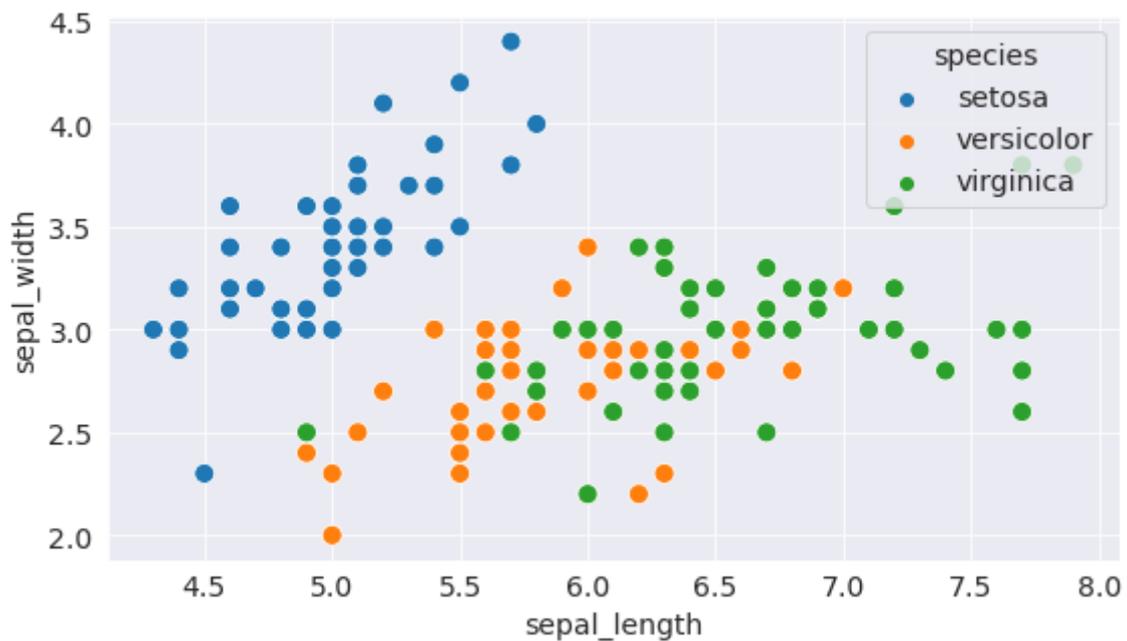
```
sns.scatterplot(x=flowers_df.sepal_length, y=flowers_df.sepal_width);
```



Adding Hues

Notice how the points in the above plot seem to form distinct clusters with some outliers. We can color the dots using the flower species as a `hue`. We can also make the points larger using the `s` argument.

```
sns.scatterplot(x=flowers_df.sepal_length, y=flowers_df.sepal_width, hue=flowers_df.species, s=100)
```



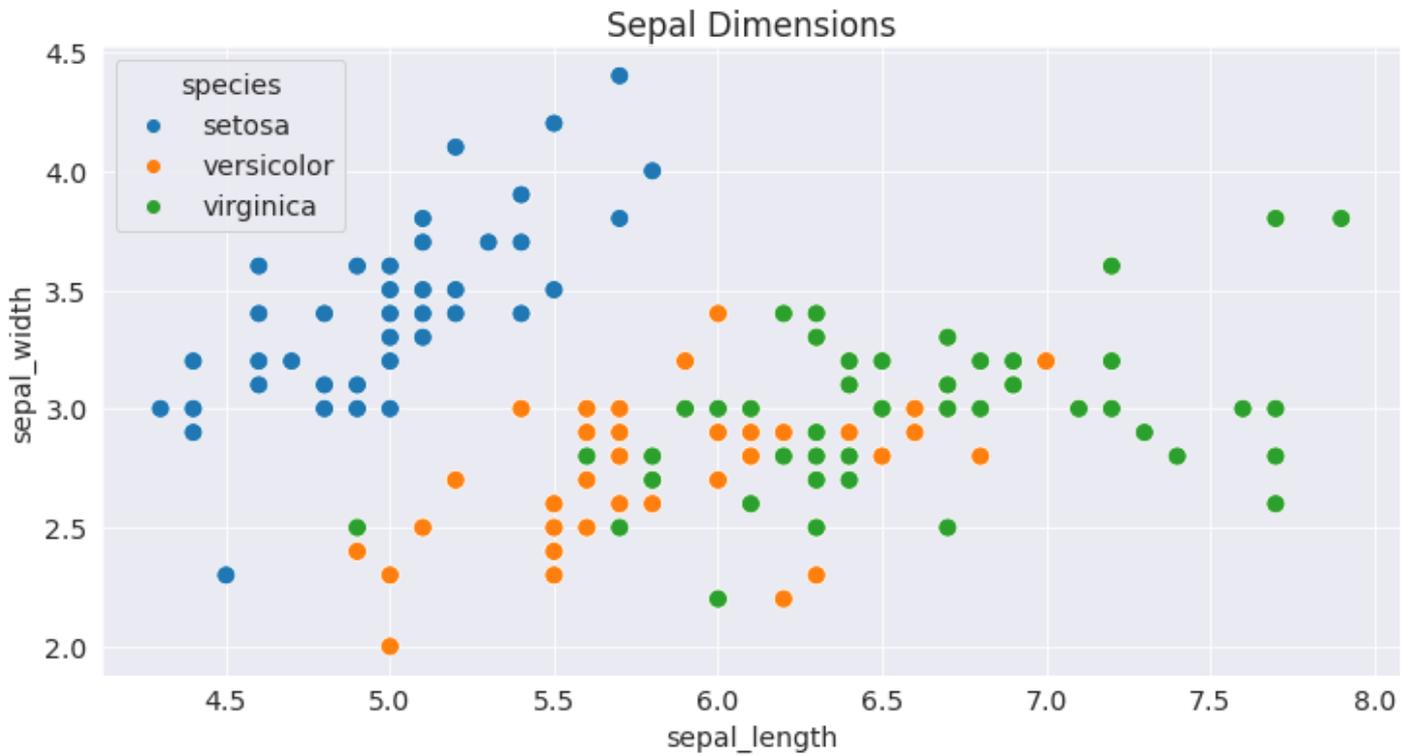
Adding hues makes the plot more informative. We can immediately tell that Setosa flowers have a smaller sepal length but higher sepal widths. In contrast, the opposite is true for Virginica flowers.

Customizing Seaborn Figures

Since Seaborn uses Matplotlib's plotting functions internally, we can use functions like `plt.figure` and `plt.title` to modify the figure.

```
plt.figure(figsize=(12, 6))
plt.title('Sepal Dimensions')
```

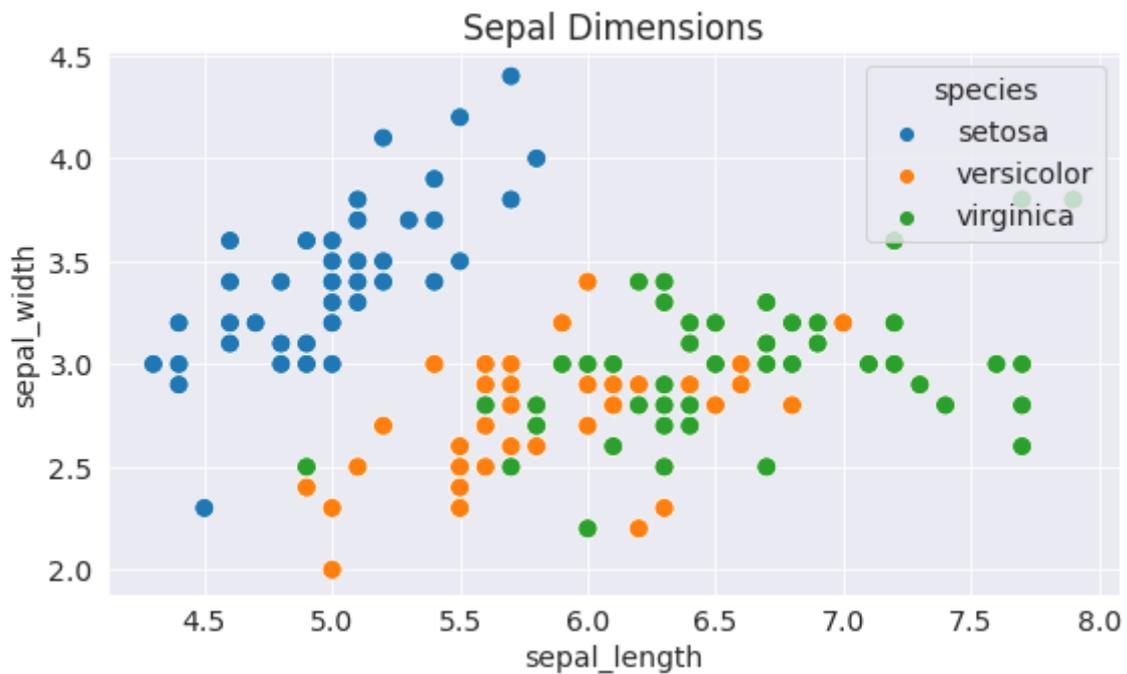
```
sns.scatterplot(x=flowers_df.sepal_length,  
               y=flowers_df.sepal_width,  
               hue=flowers_df.species,  
               s=100);
```



Plotting using Pandas Data Frames

Seaborn has in-built support for Pandas data frames. Instead of passing each column as a series, you can provide column names and use the `data` argument to specify a data frame.

```
plt.title('Sepal Dimensions')  
sns.scatterplot(x='sepal_length',  
               y='sepal_width',  
               hue='species',  
               s=100,  
               data=flowers_df);
```



Let's save and upload our work before continuing.

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-matplotlib-data-visualization" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-matplotlib-data-visualization>

'<https://jovian.ai/evanmarie/python-matplotlib-data-visualization>'

Histogram

A histogram represents the distribution of a variable by creating bins (interval) along the range of values and showing vertical bars to indicate the number of observations in each bin.

For example, let's visualize the distribution of values of sepal width in the flowers dataset. We can use the `plt.hist` function to create a histogram.

```
# Load data into a Pandas dataframe
flowers_df = sns.load_dataset("iris")
```

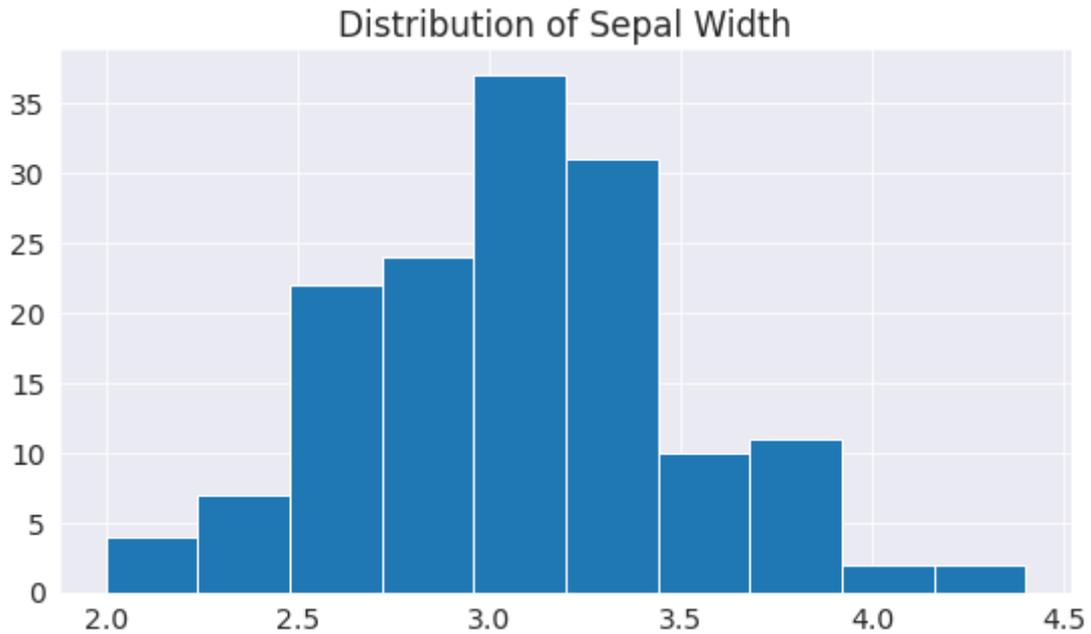
```
flowers_df.sepal_width
```

```
0    3.5
1    3.0
2    3.2
3    3.1
4    3.6
...
```

```
145    3.0
146    2.5
147    3.0
148    3.4
149    3.0
```

```
Name: sepal_width, Length: 150, dtype: float64
```

```
plt.title("Distribution of Sepal Width")
plt.hist(flowers_df.sepal_width);
```

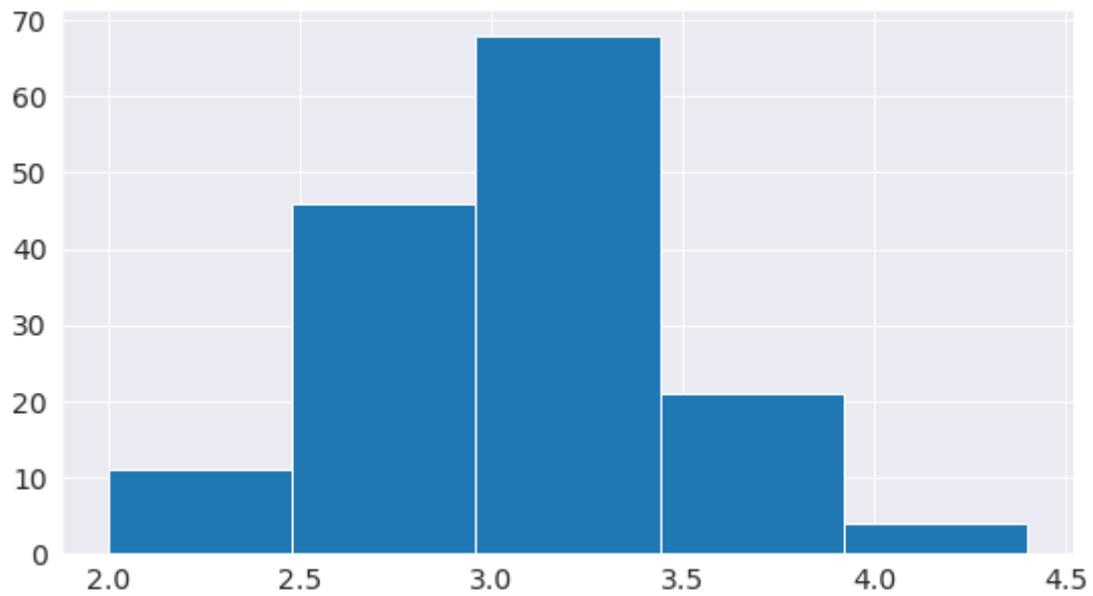


We can immediately see that the sepal widths lie in the range 2.0 - 4.5, and around 35 values are in the range 2.9 - 3.1, which seems to be the most populous bin.

Controlling the size and number of bins

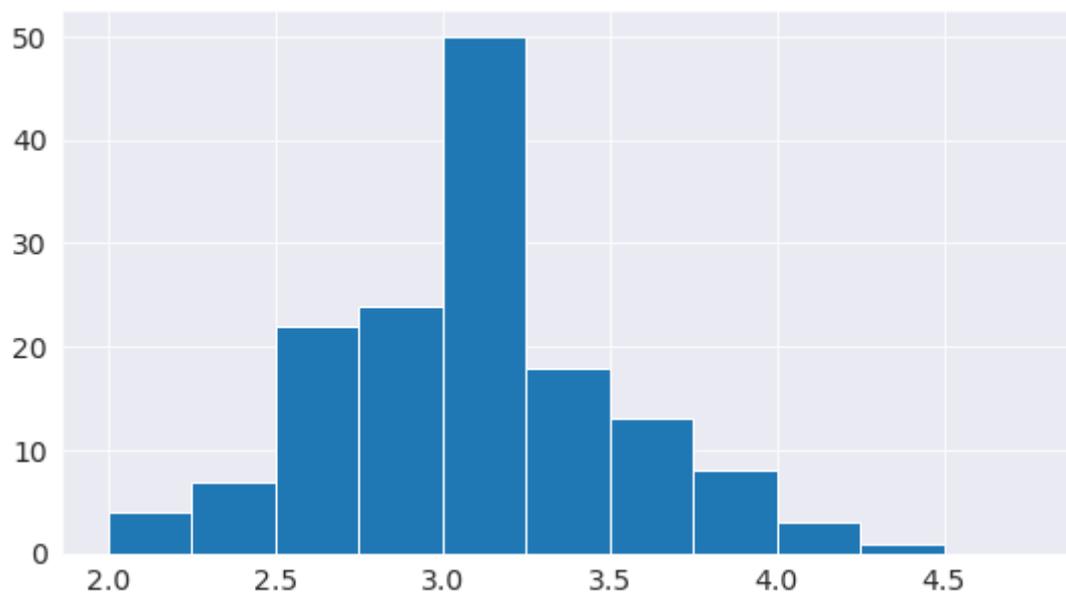
We can control the number of bins or the size of each one using the bins argument.

```
# Specifying the number of bins
plt.hist(flowers_df.sepal_width, bins=5);
```

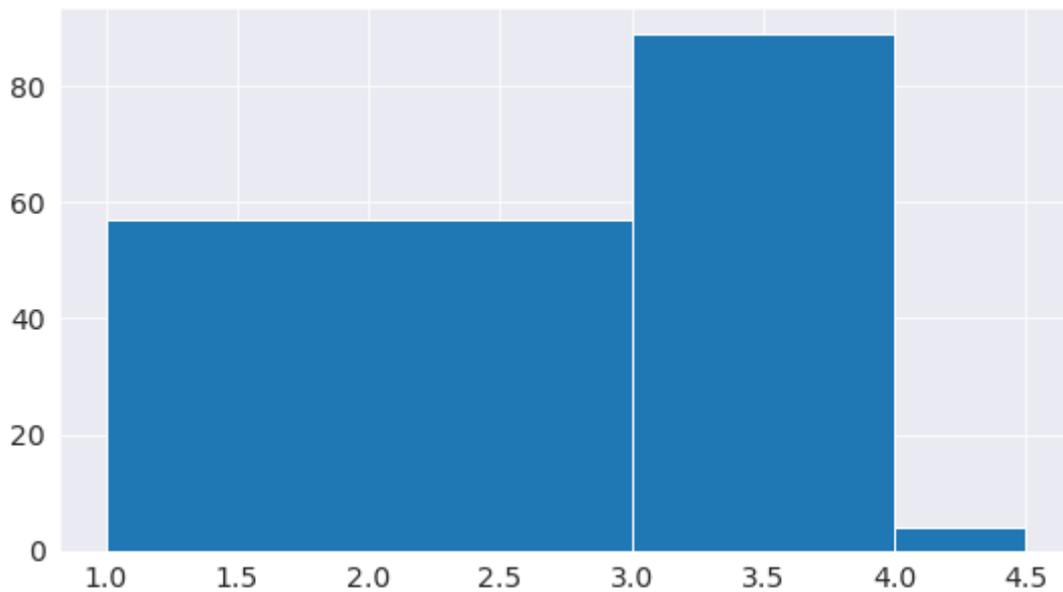


```
import numpy as np

# Specifying the boundaries of each bin
plt.hist(flowers_df.sepal_width, bins=np.arange(2, 5, 0.25));
```



```
# Bins of unequal sizes
plt.hist(flowers_df.sepal_width, bins=[1, 3, 4, 4.5]);
```



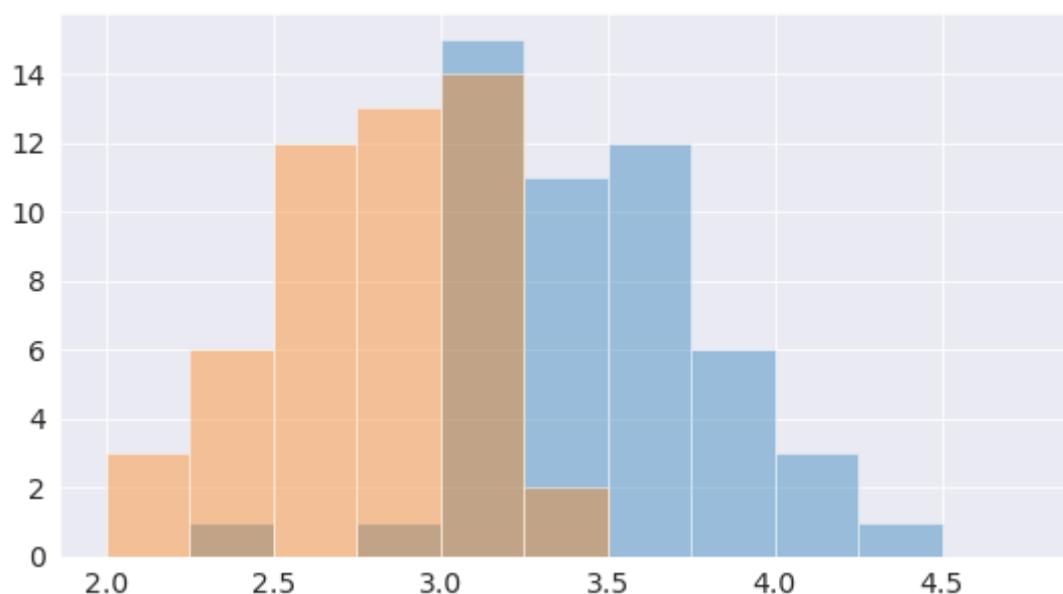
Multiple Histograms

Similar to line charts, we can draw multiple histograms in a single chart. We can reduce each histogram's opacity so that one histogram's bars don't hide the others'.

Let's draw separate histograms for each species of flowers.

```
setosa_df = flowers_df[flowers_df.species == 'setosa']
versicolor_df = flowers_df[flowers_df.species == 'versicolor']
virginica_df = flowers_df[flowers_df.species == 'virginica']
```

```
plt.hist(setosa_df.sepal_width, alpha=0.4, bins=np.arange(2, 5, 0.25));
plt.hist(versicolor_df.sepal_width, alpha=0.4, bins=np.arange(2, 5, 0.25));
```

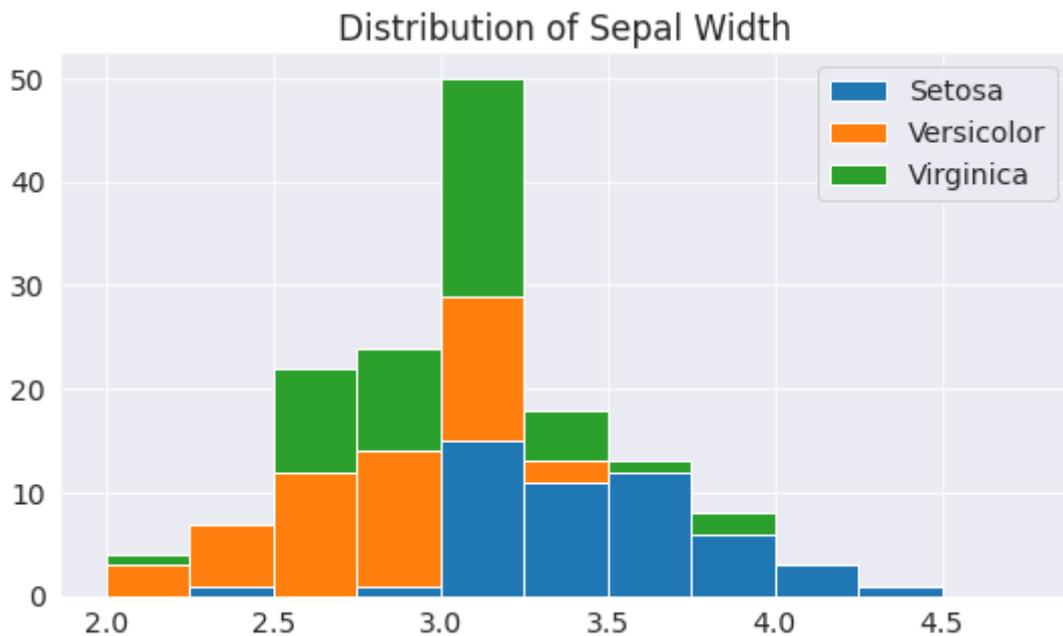


We can also stack multiple histograms on top of one another.

```
plt.title('Distribution of Sepal Width')
plt.hist([setosa_df.sepal_width, versicolor_df.sepal_width, virginica_df.sepal_width],
```

```
bins=np.arange(2, 5, 0.25),
stacked=True);
```

```
plt.legend(['Setosa', 'Versicolor', 'Virginica']);
```



Let's save and commit our work before continuing

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-matplotlib-data-visualization" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-matplotlib-data-visualization>

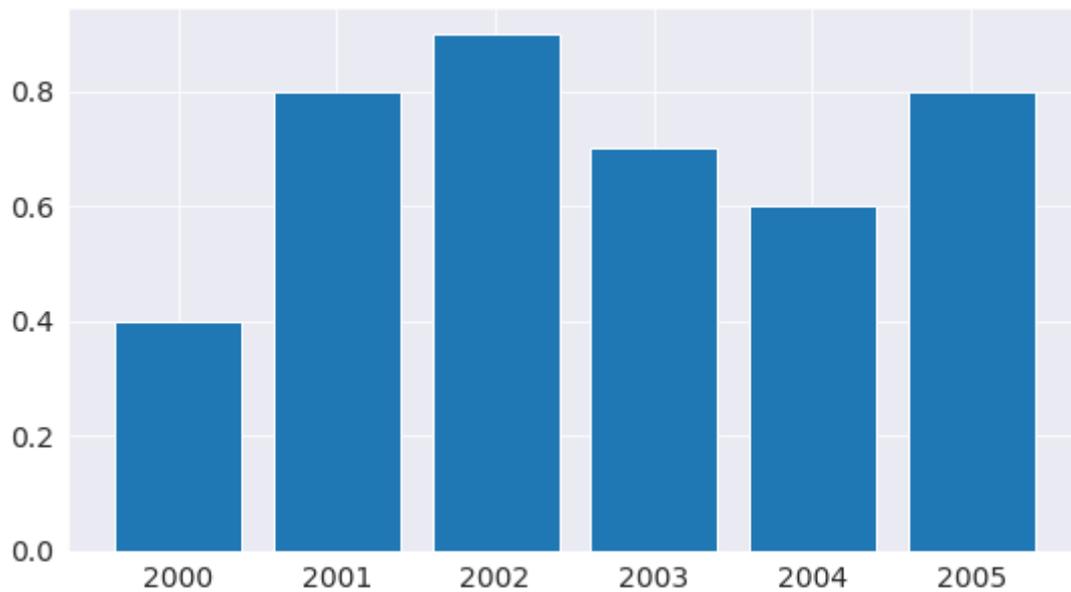
'<https://jovian.ai/evanmarie/python-matplotlib-data-visualization>'

Bar Chart

Bar charts are quite similar to line charts, i.e., they show a sequence of values. However, a bar is shown for each value, rather than points connected by lines. We can use the `plt.bar` function to draw a bar chart.

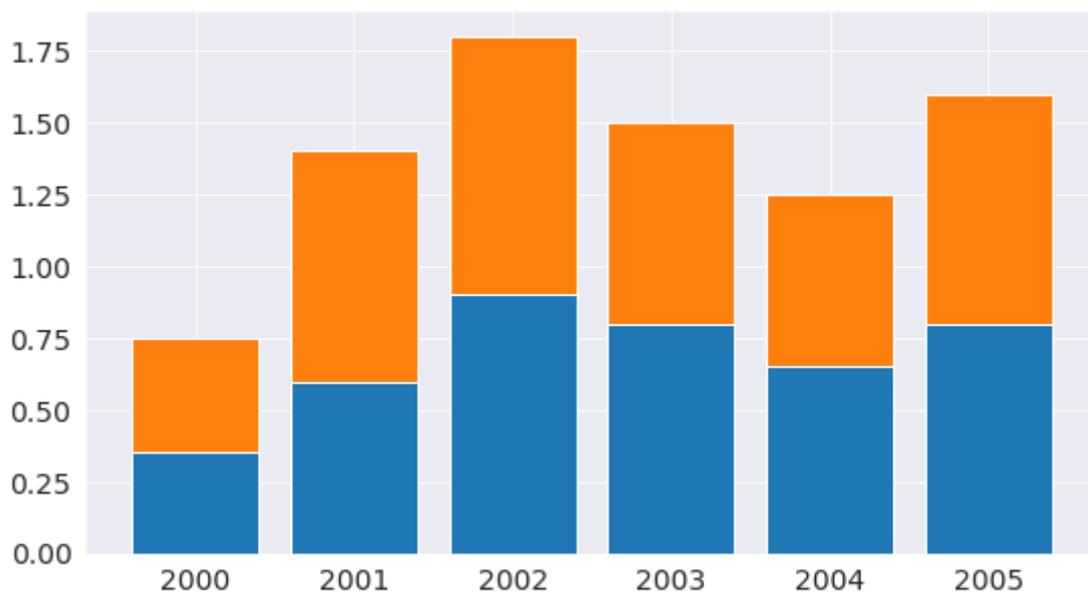
```
years = range(2000, 2006)
apples = [0.35, 0.6, 0.9, 0.8, 0.65, 0.8]
oranges = [0.4, 0.8, 0.9, 0.7, 0.6, 0.8]
```

```
plt.bar(years, oranges);
```



Like histograms, we can stack bars on top of one another. We use the `bottom` argument of `plt.bar` to achieve this.

```
plt.bar(years, apples)
plt.bar(years, oranges, bottom=apples);
```



Bar Plots with Averages

Let's look at another sample dataset included with Seaborn, called `tips`. The dataset contains information about the sex, time of day, total bill, and tip amount for customers visiting a restaurant over a week.

```
tips_df = sns.load_dataset("tips");
```

```
tips_df
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3

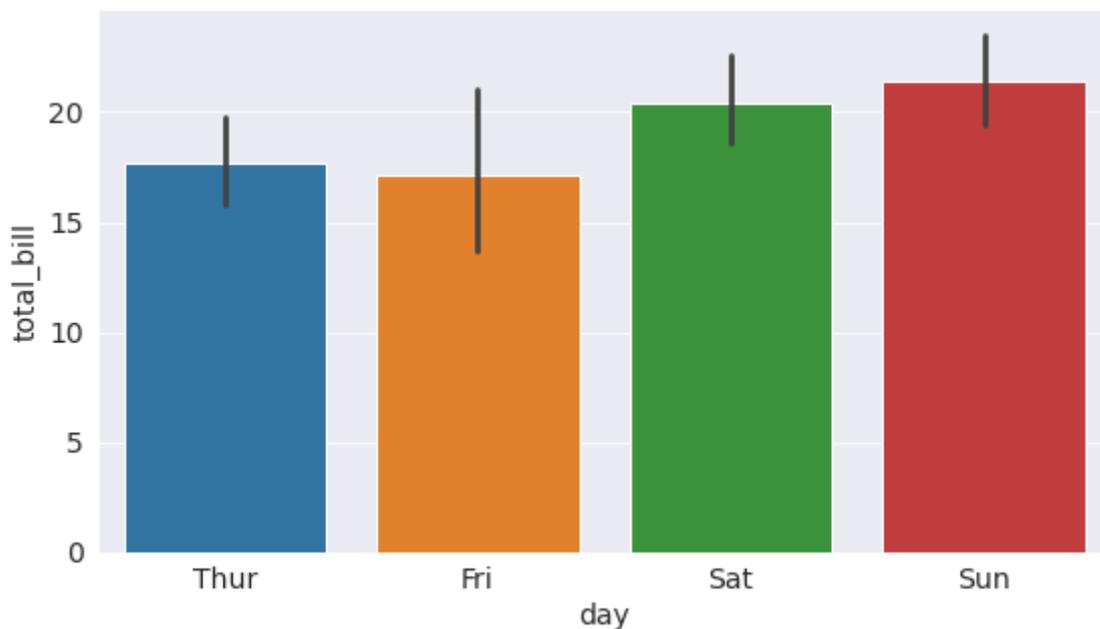
	total_bill	tip	sex	smoker	day	time	size
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
...
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

244 rows × 7 columns

We might want to draw a bar chart to visualize how the average bill amount varies across different days of the week. One way to do this would be to compute the day-wise averages and then use `plt.bar` (try it as an exercise).

However, since this is a very common use case, the Seaborn library provides a `barplot` function which can automatically compute averages.

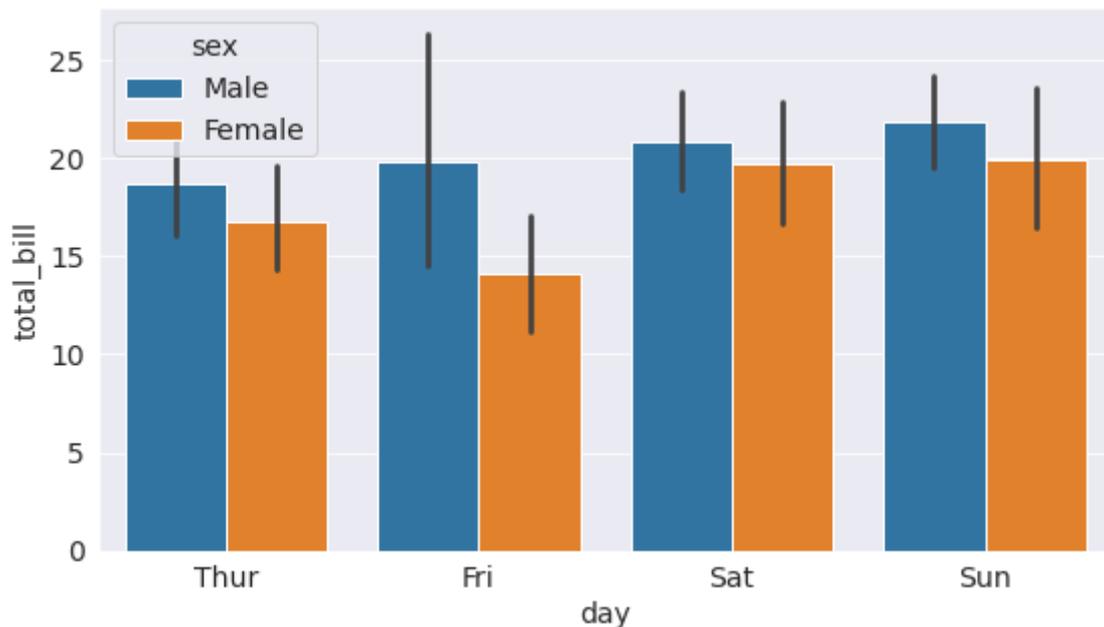
```
sns.barplot(x='day', y='total_bill', data=tips_df);
```



The lines cutting each bar represent the amount of variation in the values. For instance, it seems like the variation in the total bill is relatively high on Fridays and low on Saturday.

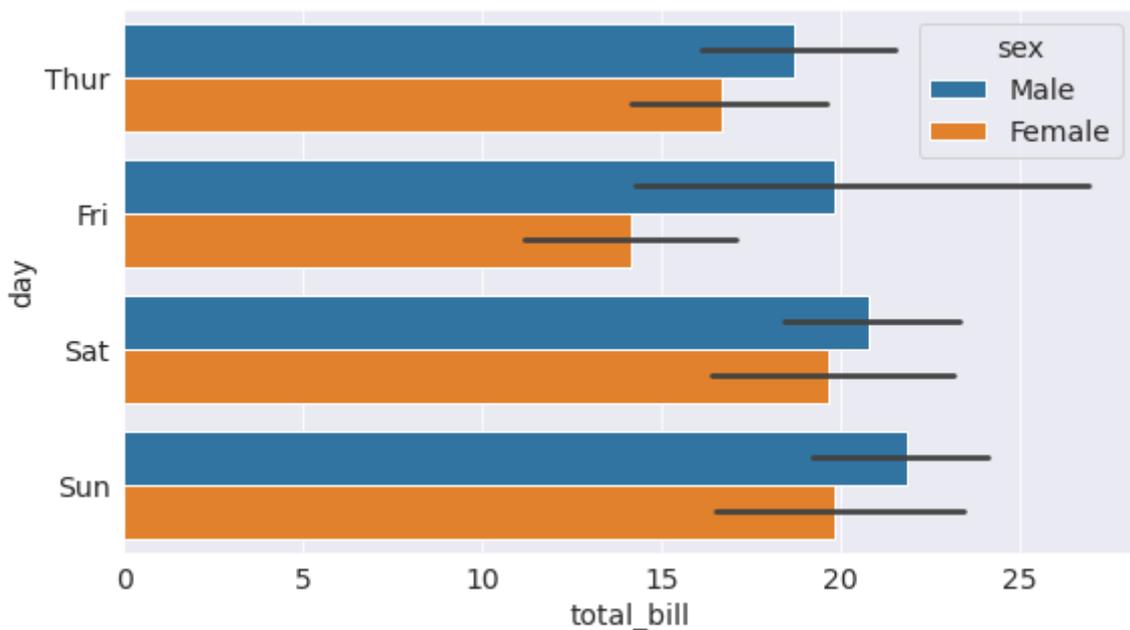
We can also specify a `hue` argument to compare bar plots side-by-side based on a third feature, e.g., `sex`.

```
sns.barplot(x='day', y='total_bill', hue='sex', data=tips_df);
```



You can make the bars horizontal simply by switching the axes.

```
sns.barplot(x='total_bill', y='day', hue='sex', data=tips_df);
```



Let's save and commit our work before continuing.

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-matplotlib-data-visualization" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-matplotlib-data-visualization>

'<https://jovian.ai/evanmarie/python-matplotlib-data-visualization>'

Heatmap

A heatmap is used to visualize 2-dimensional data like a matrix or a table using colors. The best way to understand it is by looking at an example. We'll use another sample dataset from Seaborn, called `flights`, to visualize monthly passenger footfall at an airport over 12 years.

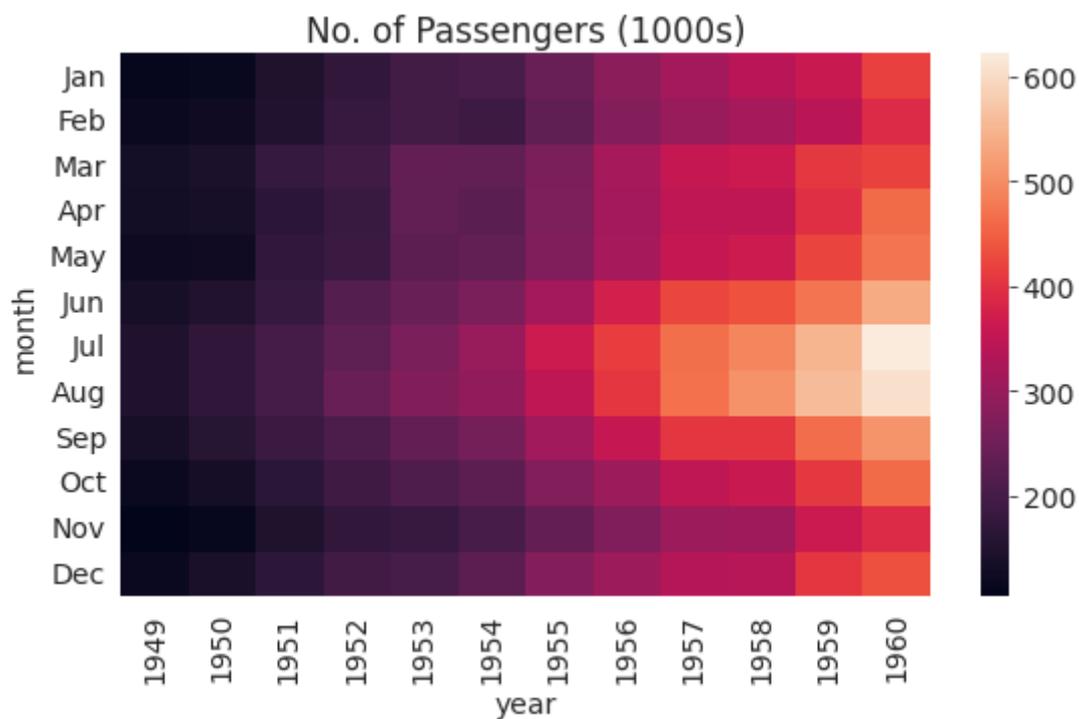
```
flights_df = sns.load_dataset("flights").pivot("month", "year", "passengers")
```

```
flights_df
```

year	1949	1950	1951	1952	1953	1954	1955	1956	1957	1958	1959	1960
month												
Jan	112	115	145	171	196	204	242	284	315	340	360	417
Feb	118	126	150	180	196	188	233	277	301	318	342	391
Mar	132	141	178	193	236	235	267	317	356	362	406	419
Apr	129	135	163	181	235	227	269	313	348	348	396	461
May	121	125	172	183	229	234	270	318	355	363	420	472
Jun	135	149	178	218	243	264	315	374	422	435	472	535
Jul	148	170	199	230	264	302	364	413	465	491	548	622
Aug	148	170	199	242	272	293	347	405	467	505	559	606
Sep	136	158	184	209	237	259	312	355	404	404	463	508
Oct	119	133	162	191	211	229	274	306	347	359	407	461
Nov	104	114	146	172	180	203	237	271	305	310	362	390
Dec	118	140	166	194	201	229	278	306	336	337	405	432

`flights_df` is a matrix with one row for each month and one column for each year. The values show the number of passengers (in thousands) that visited the airport in a specific month of a year. We can use the `sns.heatmap` function to visualize the footfall at the airport.

```
plt.title("No. of Passengers (1000s)")  
sns.heatmap(flights_df);
```

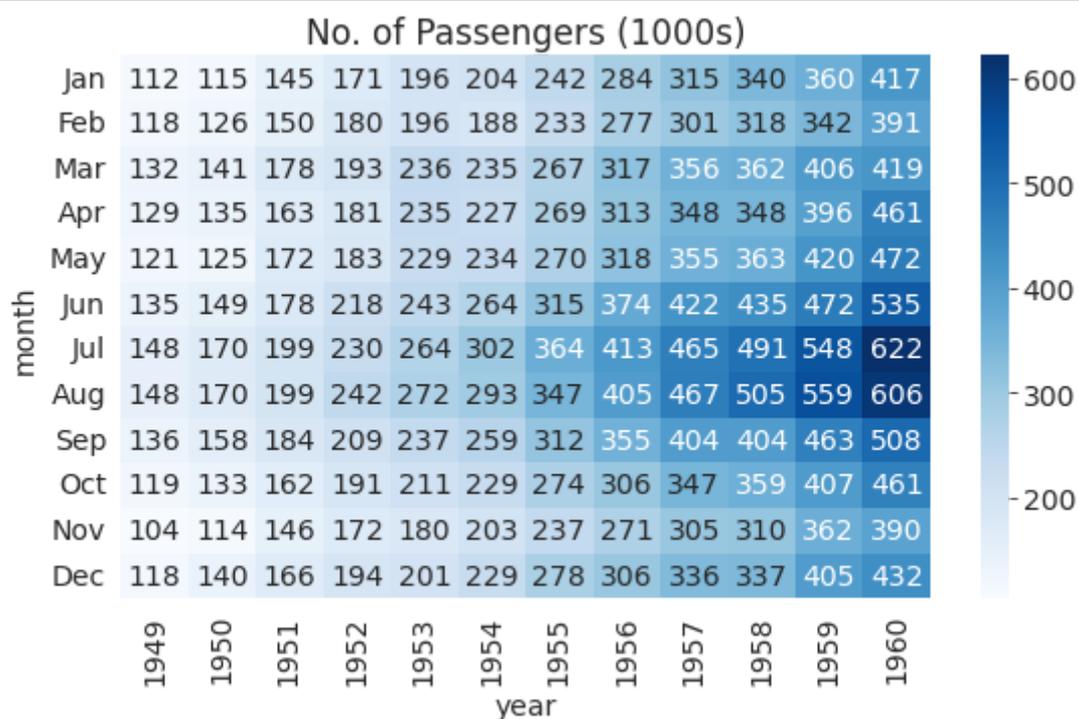


The brighter colors indicate a higher footfall at the airport. By looking at the graph, we can infer two things:

- The footfall at the airport in any given year tends to be the highest around July & August.
- The footfall at the airport in any given month tends to grow year by year.

We can also display the actual values in each block by specifying `annot=True` and using the `cmap` argument to change the color palette.

```
plt.title("No. of Passengers (1000s)")
sns.heatmap(flights_df, fmt="d", annot=True, cmap='Blues');
```



Let's save and upload our work before continuing.

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-matplotlib-data-visualization" on  
https://jovian.ai
```

```
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-matplotlib-data-visualization
```

```
'https://jovian.ai/evanmarie/python-matplotlib-data-visualization'
```

Images

We can also use Matplotlib to display images. Let's download an image from the internet.

```
from urllib.request import urlretrieve
```

```
urlretrieve('https://i.imgur.com/SkPbq.jpg', 'chart.jpg');
```

Before displaying an image, it has to be read into memory using the PIL module.

```
from PIL import Image
```

```
img = Image.open('chart.jpg')
```

An image loaded using PIL is simply a 3-dimensional numpy array containing pixel intensities for the red, green & blue (RGB) channels of the image. We can convert the image into an array using `np.array`.

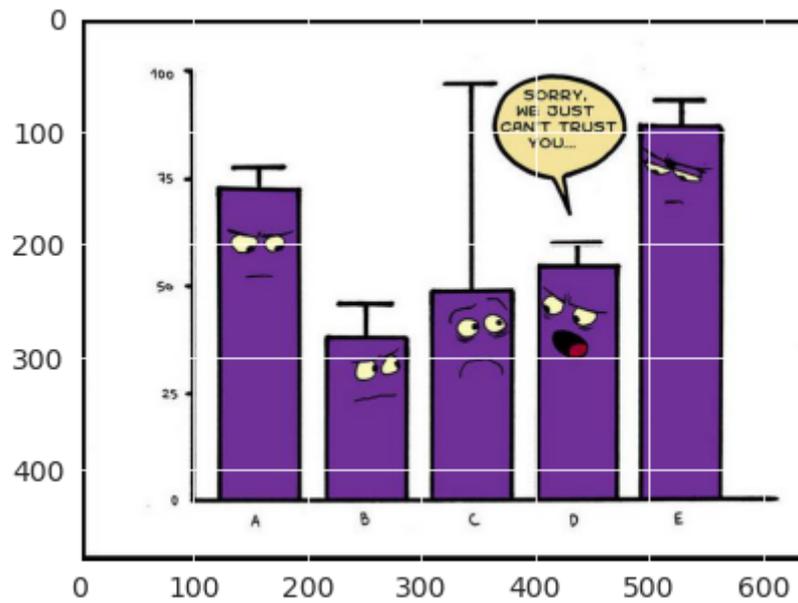
```
img_array = np.array(img)
```

```
img_array.shape
```

```
(481, 640, 3)
```

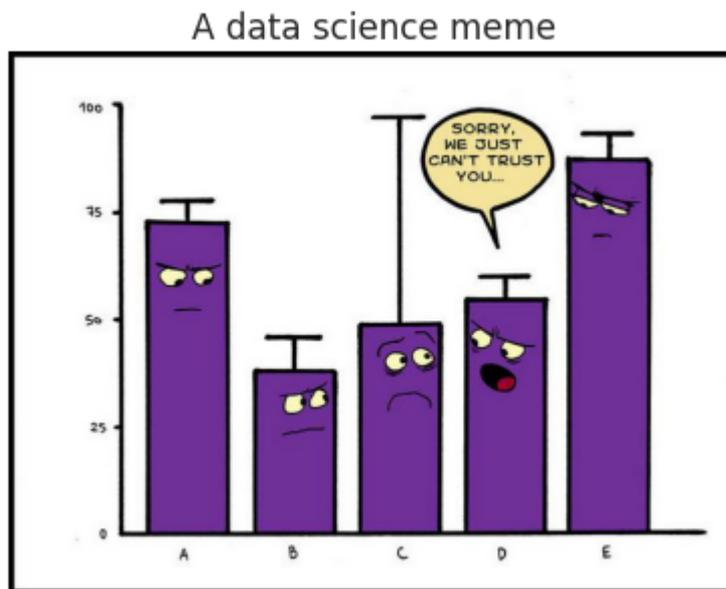
We can display the PIL image using `plt.imshow`.

```
plt.imshow(img);
```



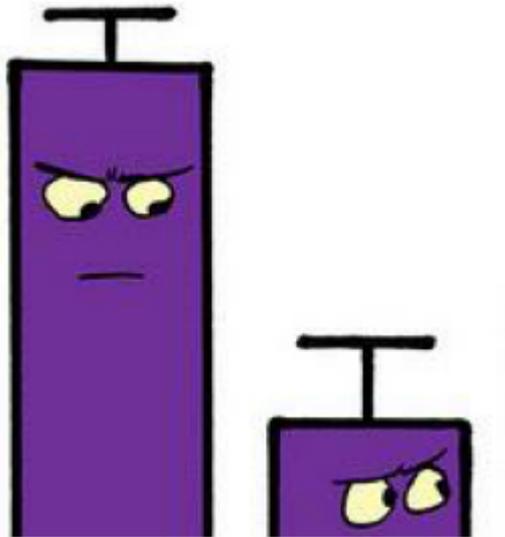
We can turn off the axes & grid lines and show a title using the relevant functions.

```
plt.grid(False)
plt.title('A data science meme')
plt.axis('off')
plt.imshow(img);
```



To display a part of the image, we can simply select a slice from the numpy array.

```
plt.grid(False)
plt.axis('off')
plt.imshow(img_array[125:325, 105:305]);
```

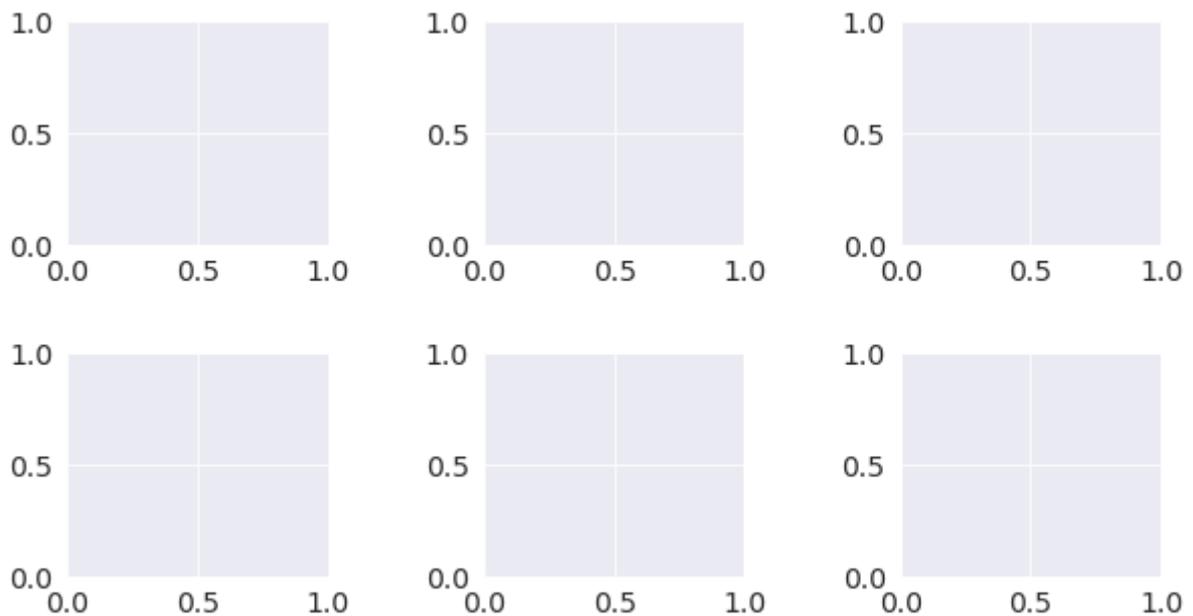


Plotting multiple charts in a grid

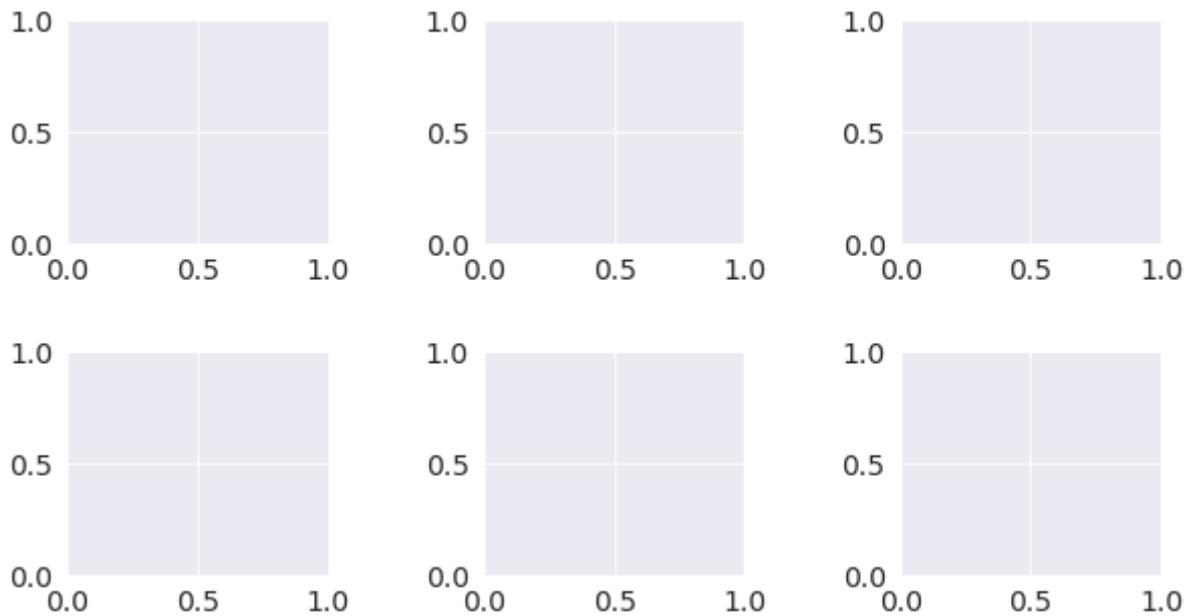
Matplotlib and Seaborn also support plotting multiple charts in a grid, using `plt.subplots`, which returns a set of axes for plotting.

Here's a single grid showing the different types of charts we've covered in this tutorial.

```
plt.subplots(2, 3); # Create a grid of graphs 2 x 3  
plt.tight_layout(pad=2) # Makes it so the numbers don't overlap
```



```
fig, axes = plt.subplots(2, 3); # Create a grid of graphs 2 x 3  
plt.tight_layout(pad=2) # Makes it so the numbers don't overlap
```



`axes` # shows axes created in previous cell

```
array([[<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>],
       [<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>]], dtype=object)
```

`axes.shape` # shows the shape of our layout is 2 grids by 3 grids

```
(2, 3)
```

`axes[0,0]` # points to a grid within the whole, a subgrid

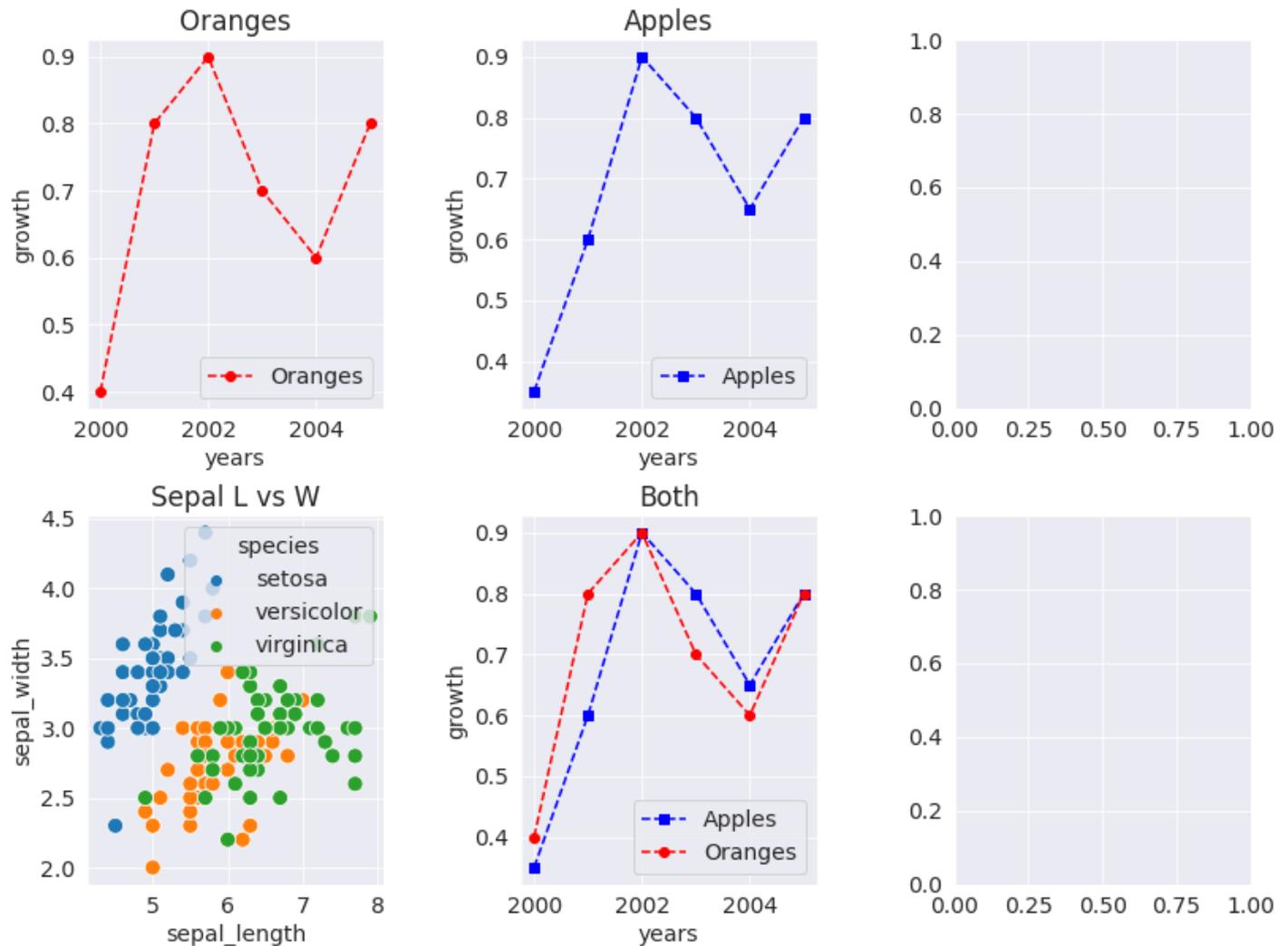
```
<AxesSubplot:>
```

```
# Create a grid of graphs 2 x 3
fig, axes = plt.subplots(2, 3, figsize=(12,9));
plt.tight_layout(pad=3) # Makes it so the numbers/labels don't overlap
axes[0,0].plot(years, oranges, "o--r") # creates the 0,0 grid
axes[0,0].set_title('Oranges') # titles 0,0 as Oranges
axes[0,0].set_xlabel('years') # labels the x axis
axes[0,0].set_ylabel('growth') # labels the y axis
axes[0,0].legend(['Oranges']) # creates a legend in the grid
axes[0,1].plot(years, apples, "s--b") # creates the 0,1 grid
axes[0,1].set_title("Apples") # titles 0, 1 as Apples
axes[0,1].legend(['Apples'])
axes[0,1].set_xlabel('years')
axes[0,1].set_ylabel('growth')
axes[1,1].plot(years, apples, "s--b" )
axes[1,1].plot(years, oranges, "o--r" )
axes[1,1].set_title("Both")
axes[1,1].legend(['Apples', 'Oranges'])
axes[1,1].set_xlabel('years')
axes[1,1].set_ylabel('growth')

axes[1,0].set_title("Sepal L vs W")
```

```
sns.scatterplot(x=flowers_df.sepal_length,
                y=flowers_df.sepal_width,
                hue=flowers_df.species,
                s=100, ax=axes[1,0])
```

```
<AxesSubplot:title={'center':'Sepal L vs W'}, xlabel='sepal_length',
ylabel='sepal_width'>
```



```
fig, axes = plt.subplots(2, 3, figsize=(16, 8))
```

```
# Use the axes for plotting
axes[0,0].plot(years, apples, 's-b')
axes[0,0].plot(years, oranges, 'o--r')
axes[0,0].set_xlabel('Year')
axes[0,0].set_ylabel('Yield (tons per hectare)')
axes[0,0].legend(['Apples', 'Oranges']);
axes[0,0].set_title('Crop Yields in Kanto')
```

```
# Pass the axes into seaborn
axes[0,1].set_title('Sepal Length vs. Sepal Width')
sns.scatterplot(x=flowers_df.sepal_length,
                y=flowers_df.sepal_width,
```

```

hue=flowers_df.species,
s=100,
ax=axes[0,1]);

# Use the axes for plotting
axes[0,2].set_title('Distribution of Sepal Width')
axes[0,2].hist([setosa_df.sepal_width, versicolor_df.sepal_width, virginica_df.sepal_wi
bins=np.arange(2, 5, 0.25),
stacked=True);

axes[0,2].legend(['Setosa', 'Versicolor', 'Virginica']);

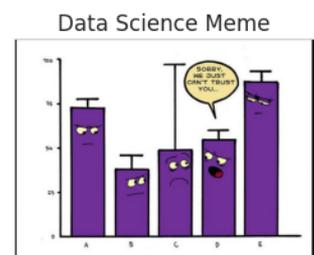
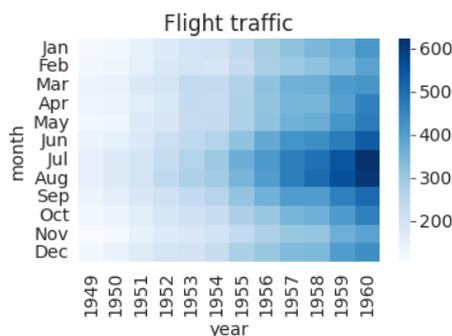
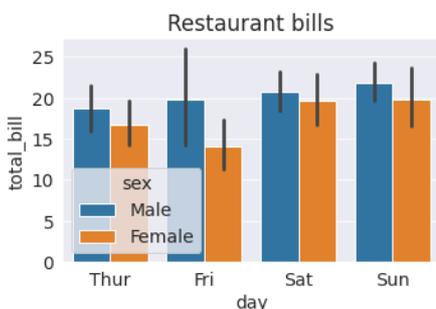
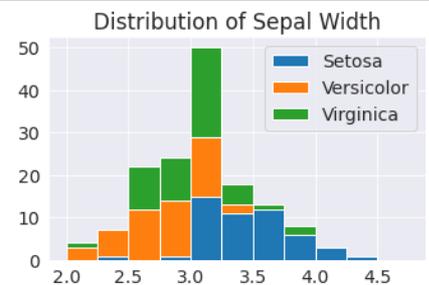
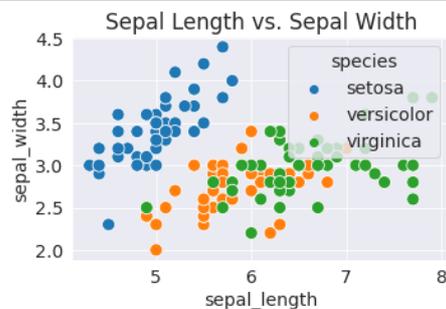
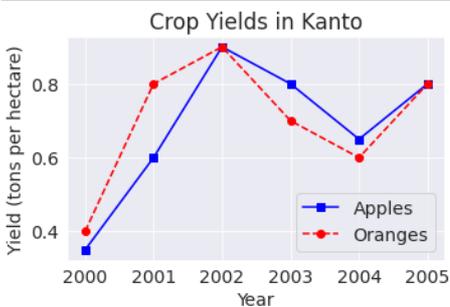
# Pass the axes into seaborn
axes[1,0].set_title('Restaurant bills')
sns.barplot(x='day', y='total_bill', hue='sex', data=tips_df, ax=axes[1,0]);

# Pass the axes into seaborn
axes[1,1].set_title('Flight traffic')
sns.heatmap(flights_df, cmap='Blues', ax=axes[1,1]);

# Plot an image using the axes
axes[1,2].set_title('Data Science Meme')
axes[1,2].imshow(img)
axes[1,2].grid(False)
axes[1,2].set_xticks([])
axes[1,2].set_yticks([])

plt.tight_layout(pad=2);

```

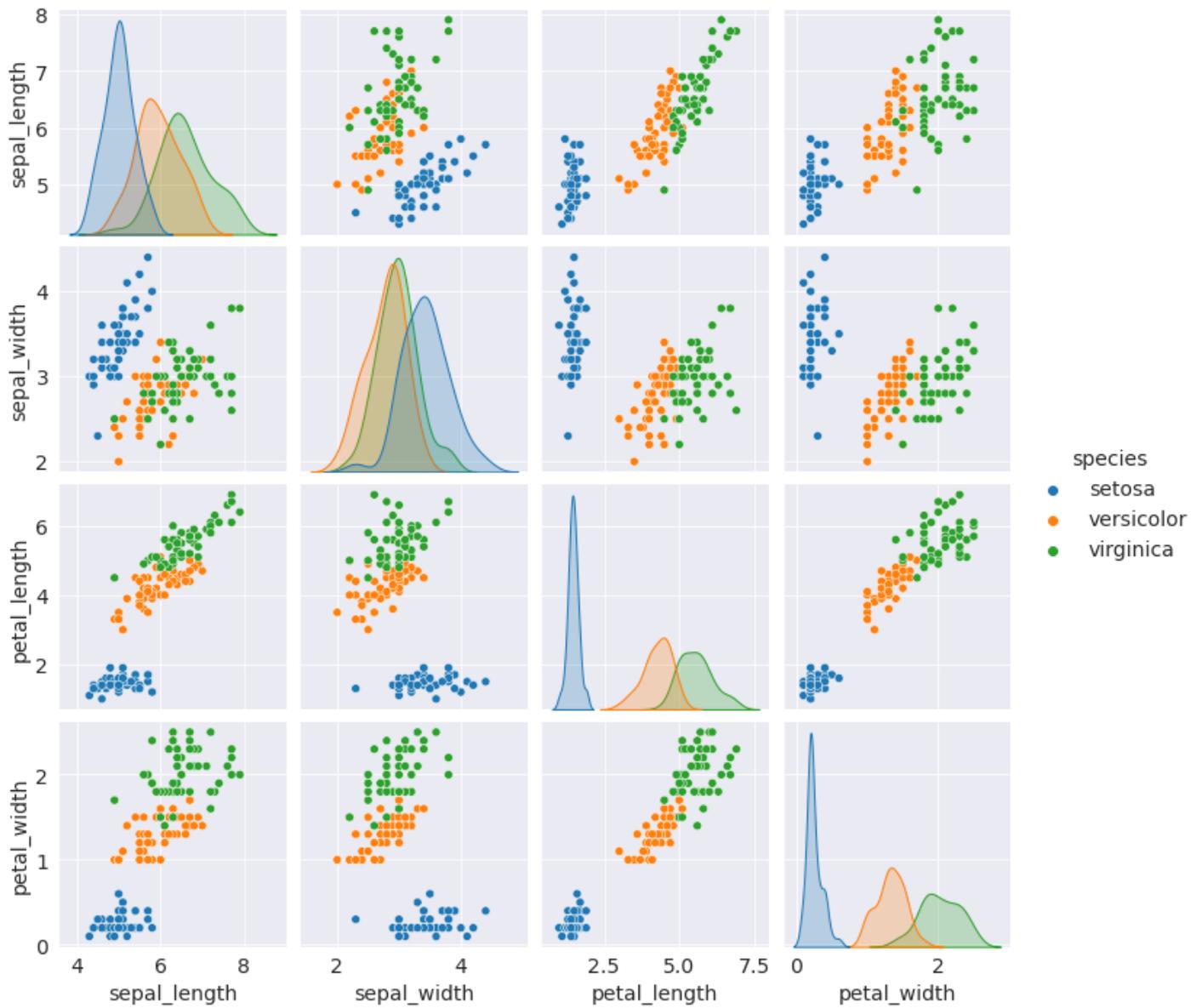


See this page for a full list of supported functions: https://matplotlib.org/3.3.1/api/axes_api.html#the-axes-class.

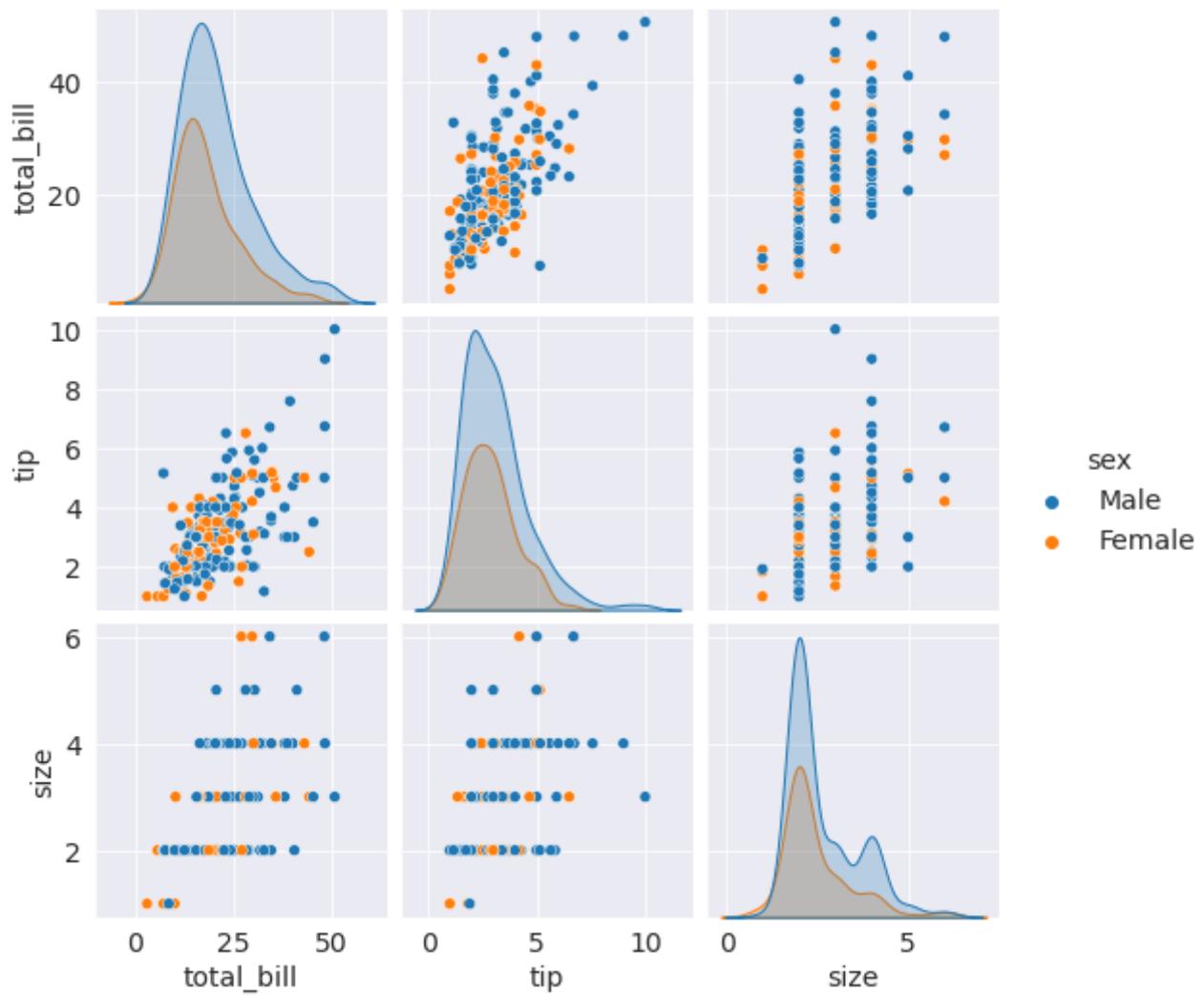
Pair plots with Seaborn

Seaborn also provides a helper function `sns.pairplot` to automatically plot several different charts for pairs of features within a dataframe.

```
sns.pairplot(flowers_df, hue='species');
```



```
sns.pairplot(tips_df, hue='sex');
```



Let's save and upload our work before continuing.

```
import jovian
```

```
jovian.commit()
```

[jovian] Attempting to save notebook..

Summary and Further Reading

We have covered the following topics in this tutorial:

- Creating and customizing line charts using Matplotlib
- Visualizing relationships between two or more variables using scatter plots
- Studying distributions of variables using histograms & bar charts to
- Visualizing two-dimensional data using heatmaps
- Displaying images using Matplotlib's `plt.imshow`
- Plotting multiple Matplotlib and Seaborn charts in a grid

In this tutorial we've covered some of the fundamental concepts and popular techniques for data visualization using Matplotlib and Seaborn. Data visualization is a vast field and we've barely scratched the surface here. Check out these references to learn and discover more:

- Data Visualization cheat sheet: <https://jovian.ml/aakashns/dataviz-cheatsheet>
- Seaborn gallery: <https://seaborn.pydata.org/examples/index.html>
- Matplotlib gallery: <https://matplotlib.org/3.1.1/gallery/index.html>
- Matplotlib tutorial: <https://github.com/rougier/matplotlib-tutorial>

You are now ready to move on to the next tutorial: [Exploratory Data Analysis - A Case Study](#).

Questions for Revision

Try answering the following questions to test your understanding of the topics covered in this notebook:

1. What is data visualization?
2. What is Matplotlib?
3. What is Seaborn?
4. How do you install Matplotlib and Seaborn?
5. How you import Matplotlib and Seaborn? What are the common aliases used while importing these modules?
6. What is the purpose of the magic command `%matplotlib inline`?
7. What is a line chart?
8. How do you plot a line chart in Python? Illustrate with an example.
9. How do you specify values for the X-axis of a line chart?
10. How do you specify labels for the axes of a chart?
11. How do you plot multiple line charts on the same axes?
12. How do you show a legend for a line chart with multiple lines?
13. How you set a title for a chart?
14. How do you show markers on a line chart?
15. What are the different options for styling lines & markers in line charts? Illustrate with examples?
16. What is the purpose of the `fmt` argument to `plt.plot`?
17. How do you markers without a line using `plt.plot`?
18. Where can you see a list of all the arguments accepted by `plt.plot`?
19. How do you change the size of the figure using Matplotlib?
20. How do you apply the default styles from Seaborn globally for all charts?
21. What are the predefined styles available in Seaborn? Illustrate with examples.
22. What is a scatter plot?
23. How is a scatter plot different from a line chart?
24. How do you draw a scatter plot using Seaborn? Illustrate with an example.
25. How do you decide when to use a scatter plot v.s. a line chart?
26. How do you specify the colors for dots on a scatter plot using a categorical variable?
27. How do you customize the title, figure size, legend, etc., for Seaborn plots?
28. How do you use a Pandas dataframe with `sns.scatterplot`?
29. What is a histogram?

30. When should you use a histogram v.s. a line chart?
31. How do you draw a histogram using Matplotlib? Illustrate with an example.
32. What are "bins" in a histogram?
33. How do you change the sizes of bins in a histogram?
34. How do you change the number of bins in a histogram?
35. How do you show multiple histograms on the same axes?
36. How do you stack multiple histograms on top of one another?
37. What is a bar chart?
38. How do you draw a bar chart using Matplotlib? Illustrate with an example.
39. What is the difference between a bar chart and a histogram?
40. What is the difference between a bar chart and a line chart?
41. How do you stack bars on top of one another?
42. What is the difference between `plt.bar` and `sns.barplot`?
43. What do the lines cutting the bars in a Seaborn bar plot represent?
44. How do you show bar plots side-by-side?
45. How do you draw a horizontal bar plot?
46. What is a heat map?
47. What type of data is best visualized with a heat map?
48. What does the `pivot` method of a Pandas dataframe do?
49. How do you draw a heat map using Seaborn? Illustrate with an example.
50. How do you change the color scheme of a heat map?
51. How do you show the original values from the dataset on a heat map?
52. How do you download images from a URL in Python?
53. How do you open an image for processing in Python?
54. What is the purpose of the PIL module in Python?
55. How do you convert an image loaded using PIL into a Numpy array?
56. How many dimensions does a Numpy array for an image have? What does each dimension represent?
57. What are "color channels" in an image?
58. What is RGB?
59. How do you display an image using Matplotlib?
60. How do you turn off the axes and gridlines in a chart?
61. How do you display a portion of an image using Matplotlib?
62. How do you plot multiple charts in a grid using Matplotlib and Seaborn? Illustrate with examples.
63. What is the purpose of the `plt.subplots` function?
64. What are pair plots in Seaborn? Illustrate with an example.
65. How do you export a plot into a PNG image file using Matplotlib?
66. Where can you learn about the different types of charts you can create using Matplotlib and Seaborn?

Data Visualization Guide

Vol 1

Data Visualization

Data visualization is the graphic representation of data. It involves producing images that communicate relationships among the represented data to viewers. Visualizing data is an essential part of data analysis and machine learning, but choosing the right type of visualization is often challenging. This guide provides an introduction to popular data visualization techniques, by presenting sample use cases and providing code examples using Python.

Types of graphs covered:

- Line graph
- Scatter plot
- Histogram and Frequency Distribution
- Heatmap
- Contour Plot
- Box Plot
- Bar Chart

Import libraries

- [Matplotlib](#): Plotting and visualization library for Python. We'll use the pyplot module from matplotlib. As convention, it is often imported as plt.
- [Seaborn](#): An easy-to-use visualization library that builds on top of Matplotlib and lets you create beautiful charts with just a few lines of code.

```
# Uncomment the next line to install the required libraries  
# !pip install matplotlib seaborn --upgrade --quiet
```

```
# Import libraries  
import matplotlib  
import matplotlib.pyplot as plt  
import seaborn as sns
```

```
# Configuring styles
sns.set_style("darkgrid")
matplotlib.rcParams['font.size'] = 14
matplotlib.rcParams['figure.figsize'] = (9, 5)
matplotlib.rcParams['figure.facecolor'] = '#00000000'
```

Line Chart

A line chart displays information as a series of data points or markers, connected by a straight lines. You can customize the shape, size, color and other aesthetic elements of the markers and lines for better visual clarity.

Example

We'll create a line chart to compare the yields of apples and oranges over 12 years in the imaginary region of Hoenn.

```
# Sample data
years = range(2000, 2012)
apples = [0.895, 0.91, 0.919, 0.926, 0.929, 0.931, 0.934, 0.936, 0.937, 0.9375, 0.9372,
oranges = [0.962, 0.941, 0.930, 0.923, 0.918, 0.908, 0.907, 0.904, 0.901, 0.898, 0.9, 0.895]

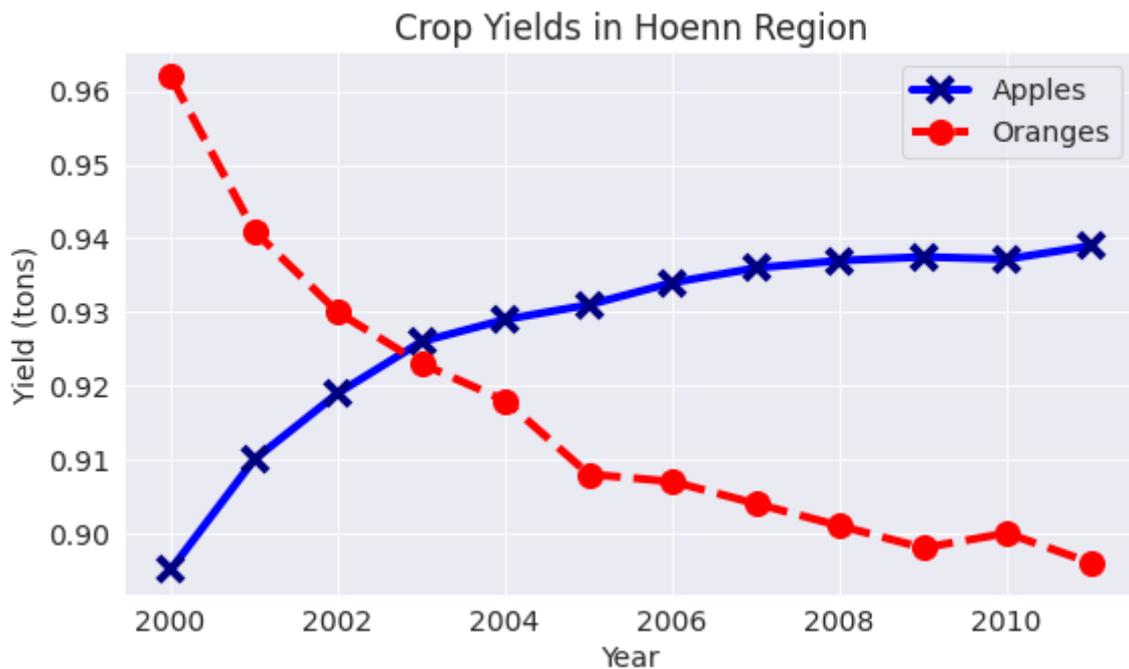
# First line
plt.plot(years, apples, 'b-x', linewidth=4, markersize=12, markeredgewidth=4, markeredgecolor='b')

# Second line
plt.plot(years, oranges, 'r--o', linewidth=4, markersize=12,);

# Title
plt.title('Crop Yields in Hoenn Region')

# Line labels
plt.legend(['Apples', 'Oranges'])

# Axis labels
plt.xlabel('Year'); plt.ylabel('Yield (tons)');
```



Scatter Plot

In a scatter plot, the values of 2 variables are plotted as points on a 2-dimensional grid. Additionally, you can also use a third variable to determine the size or color of the points.

Example

The [Iris flower dataset](#) provides samples measurements of sepals and petals for 3 species of flowers. The Iris dataset is included with the `seaborn` library, and can be loaded as a `pandas` dataframe.

```
# Load data into a Pandas dataframe
data = sns.load_dataset("iris")

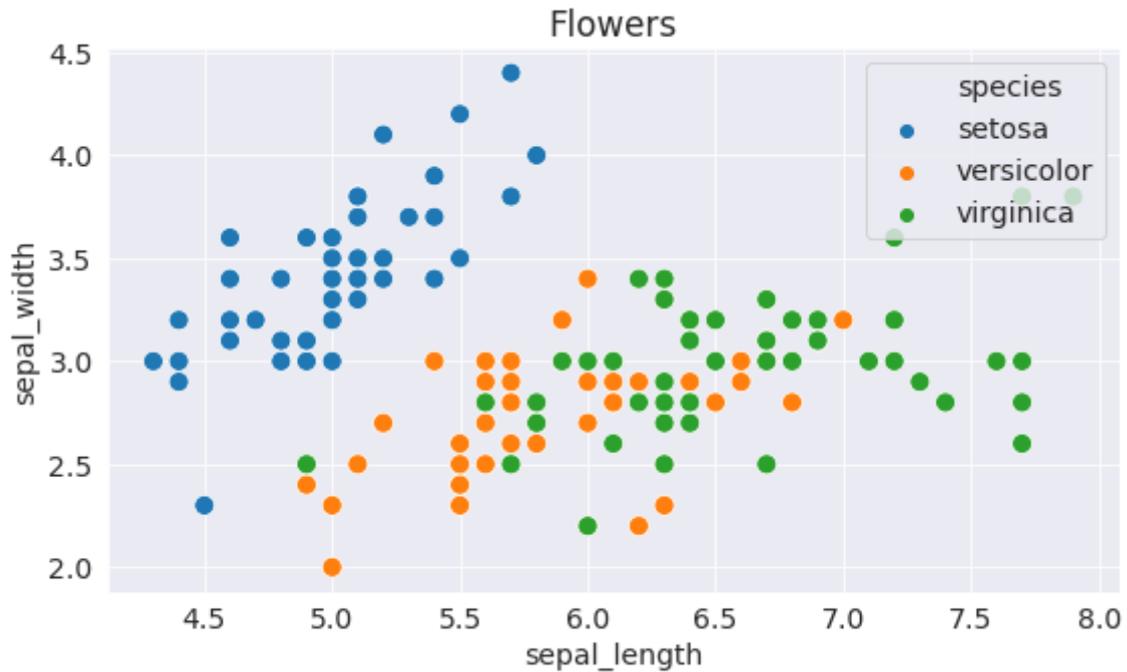
# View the data
data.sample(5)
```

	sepal_length	sepal_width	petal_length	petal_width	species
136	6.3	3.4	5.6	2.4	virginica
16	5.4	3.9	1.3	0.4	setosa
36	5.5	3.5	1.3	0.2	setosa
8	4.4	2.9	1.4	0.2	setosa
89	5.5	2.5	4.0	1.3	versicolor

We can use a scatter plot to visualize sepal length & sepal width vary across different species of flowers. The points for each species form a separate cluster, with some overlap between the Versicolor and Virginica species.

```
# Create a scatter plot
sns.scatterplot(data.sepal_length, # X-axis
                data.sepal_width, # Y-axis
                hue=data.species, # Dot color
                s=100);
```

```
# Chart title
plt.title("Flowers");
```



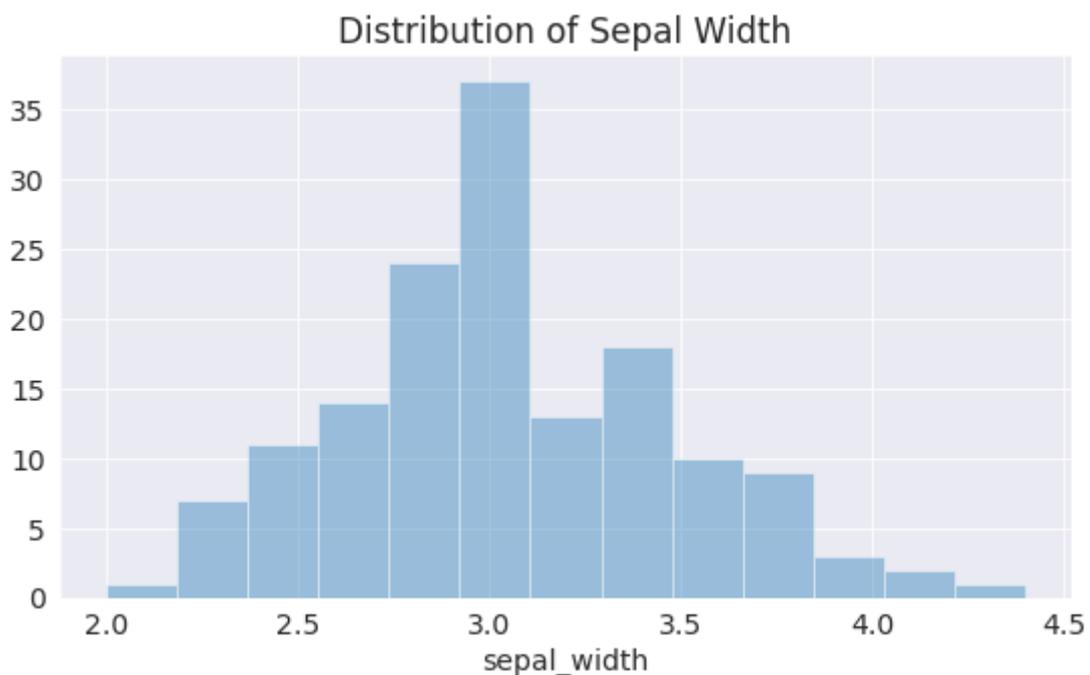
Histogram and Frequency Distribution

A histogram represents the distribution of data by forming bins along the range of the data and then drawing bars to show the number of observations that fall in each bin.

Example

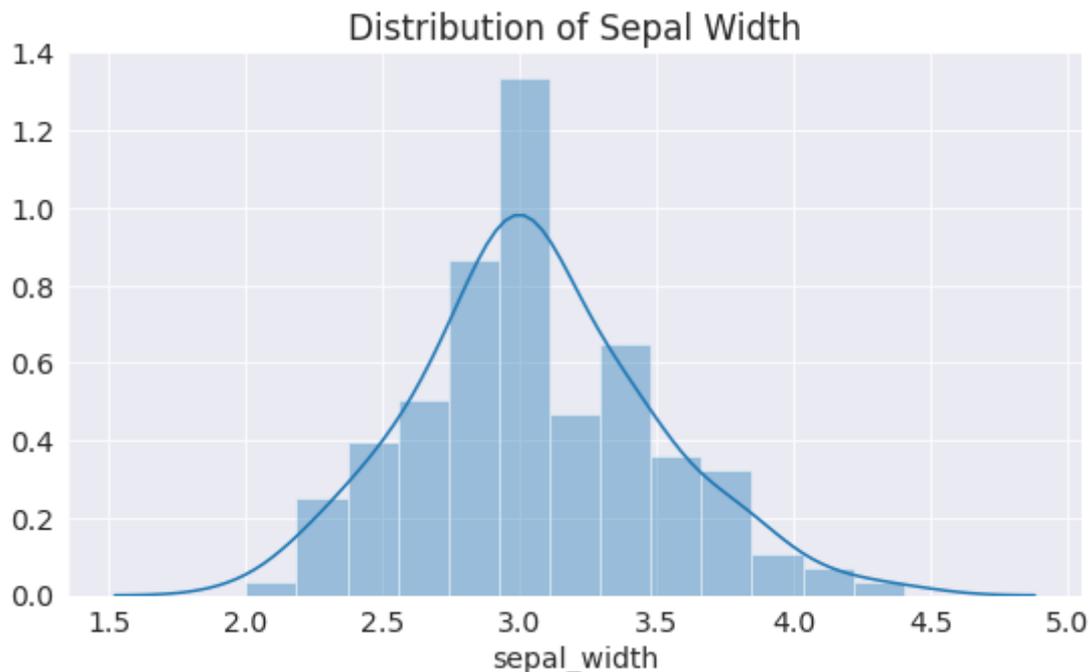
We can use a histogram to visualize how the values of sepal width are distributed.

```
plt.title("Distribution of Sepal Width")
sns.distplot(data.sepal_width, kde=False);
```



We can immediately see that values of sepal width fall in the range 2.0 - 4.5, and around 35 values are in the range 2.9 - 3.1. We can also look at this data as a frequency distribution, where the values on Y-axis are percentages instead of counts.

```
plt.title("Distribution of Sepal Width")  
  
sns.distplot(data.sepal_width);
```



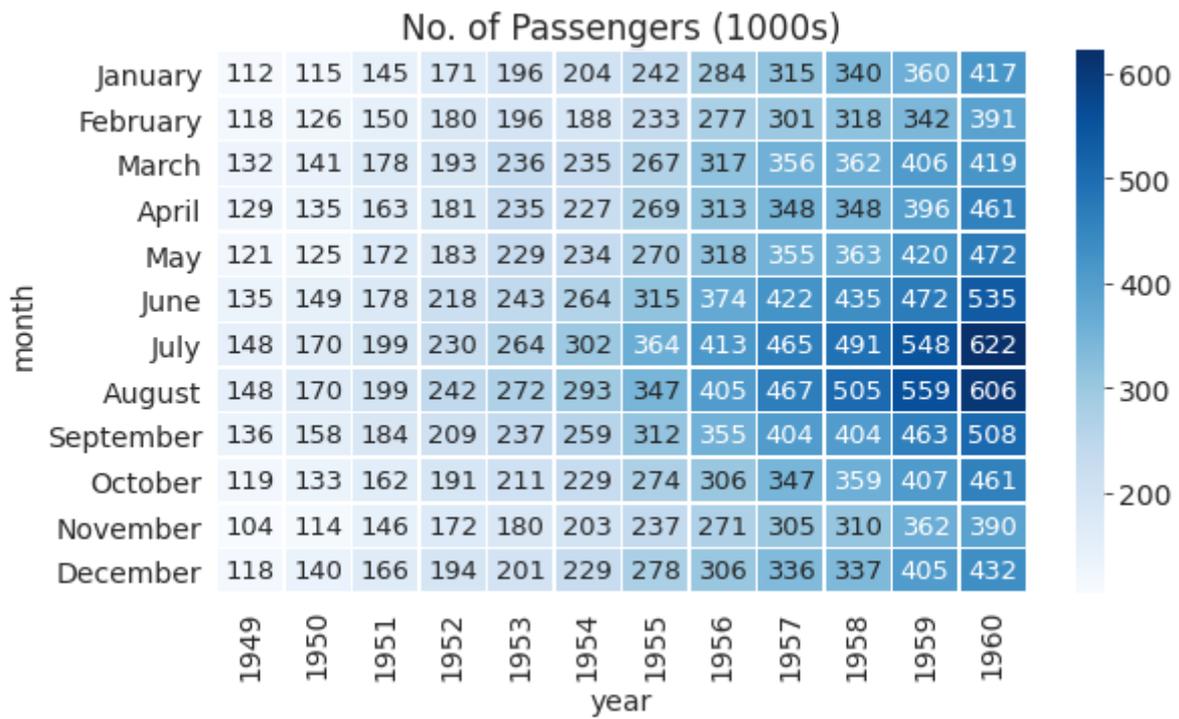
Heatmap

A heatmap is used to visualize 2-dimensional data like a matrix or a table using colors.

Example

We'll use another sample dataset from Seaborn, called "flights", to visualize monthly passenger footfall at an airport over 12 years.

```
# Load the example flights dataset as a matrix  
flights = sns.load_dataset("flights").pivot("month", "year", "passengers")  
  
# Chart Title  
plt.title("No. of Passengers (1000s)")  
  
# Draw a heatmap with the numeric values in each cell  
sns.heatmap(flights,  
            fmt="d",  
            annot=True,  
            linewidths=.5,  
            cmap='Blues',  
            annot_kws={"fontsize":13});
```



Contour Plot

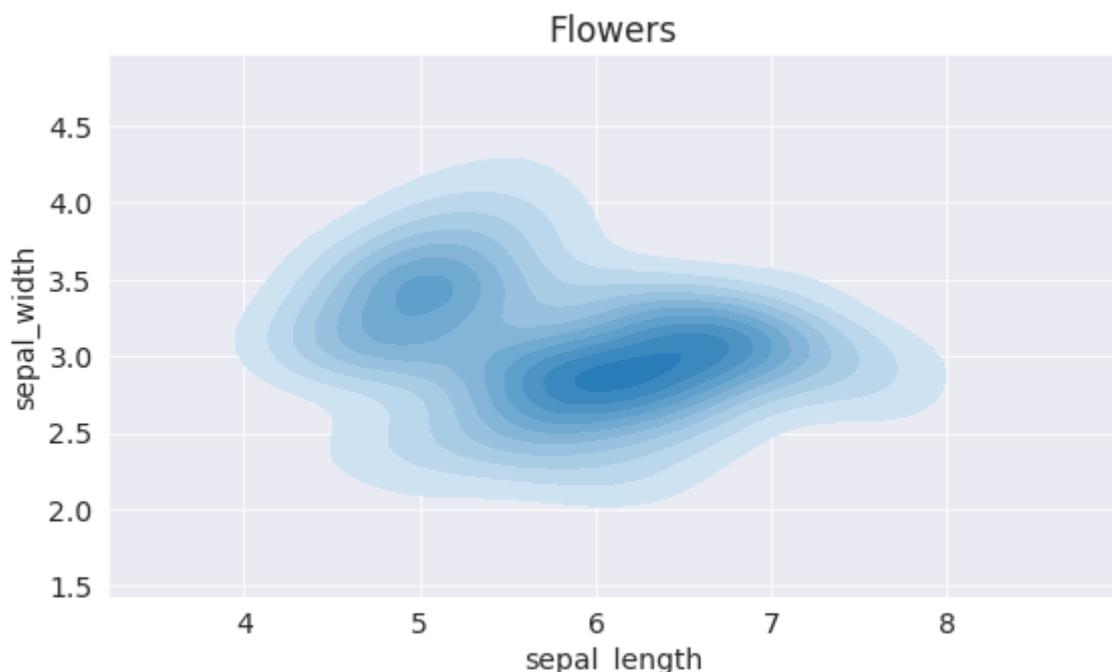
Contour plot uses contours or color-coded regions helps us to visualize 3 numerical variables in two dimensions. One variable is represented on the horizontal axis and a second variable is represented on the vertical axis. The third variable is represented by a color gradient and isolines (lines of constant value).

Example

We can visualize the values of sepal width & sepal length from the flowers dataset using a contour plot. The shade of blue represent the density of values in a region of the graph.

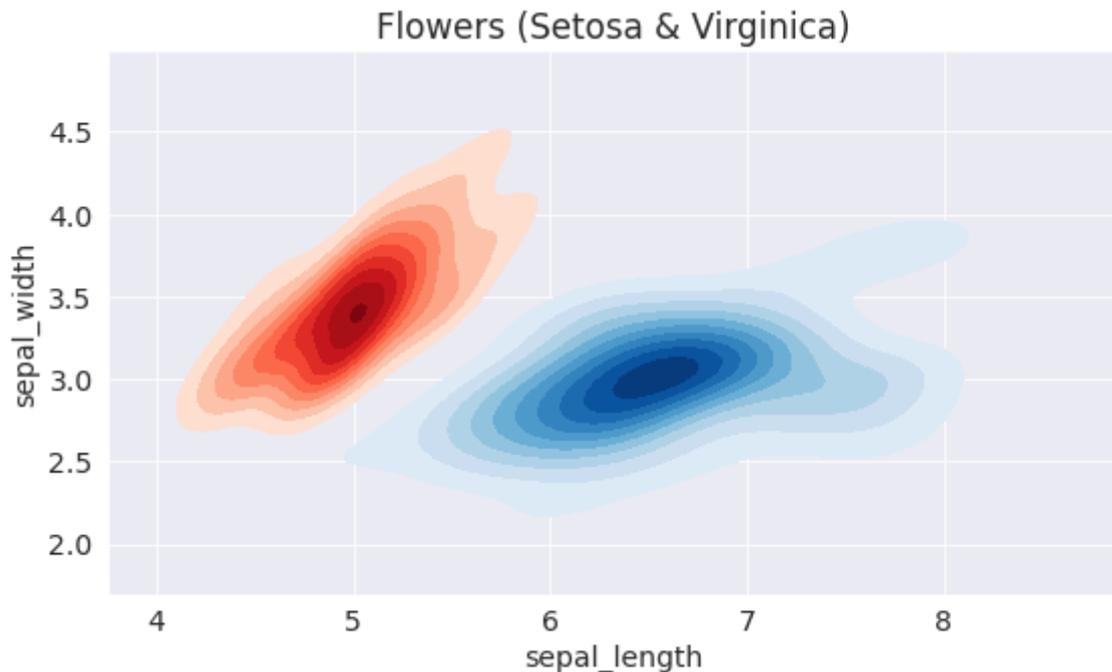
```
plt.title("Flowers")
```

```
sns.kdeplot(data.sepal_length, data.sepal_width, shade=True, shade_lowest=False);
```



We can segment species of flowers by creating multiple contour plots with different colors.

```
setosa = data[data.species == 'setosa']  
virginica = data[data.species == 'virginica']  
  
plt.title("Flowers (Setosa & Virginica)")  
  
sns.kdeplot(setosa.sepal_length, setosa.sepal_width, shade=True, cmap='Reds', shade_lowest=False)  
sns.kdeplot(virginica.sepal_length, virginica.sepal_width, shade=True, cmap='Blues', shade_lowest=False)
```



Box Plot

A box plot shows the distribution of data along a single axis, using a "box" and "whiskers". The lower end of the box represents the 1st quartile (i.e. 25% of values are below it), and the upper end of the box represents the 3rd quartile (i.e. 25% of values are above it). The median value is represented via a line inside the box. The "whiskers" represent the minimum & maximum values (sometimes excluding outliers, which are represented as dots).

Example

We'll use another sample dataset included with Seaborn, called "tips". The dataset contains information about the sex, time of day, total bill and tip amount for customers visiting a restaurant over a week.

```
# Load the example tips dataset  
tips = sns.load_dataset("tips");  
tips
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2

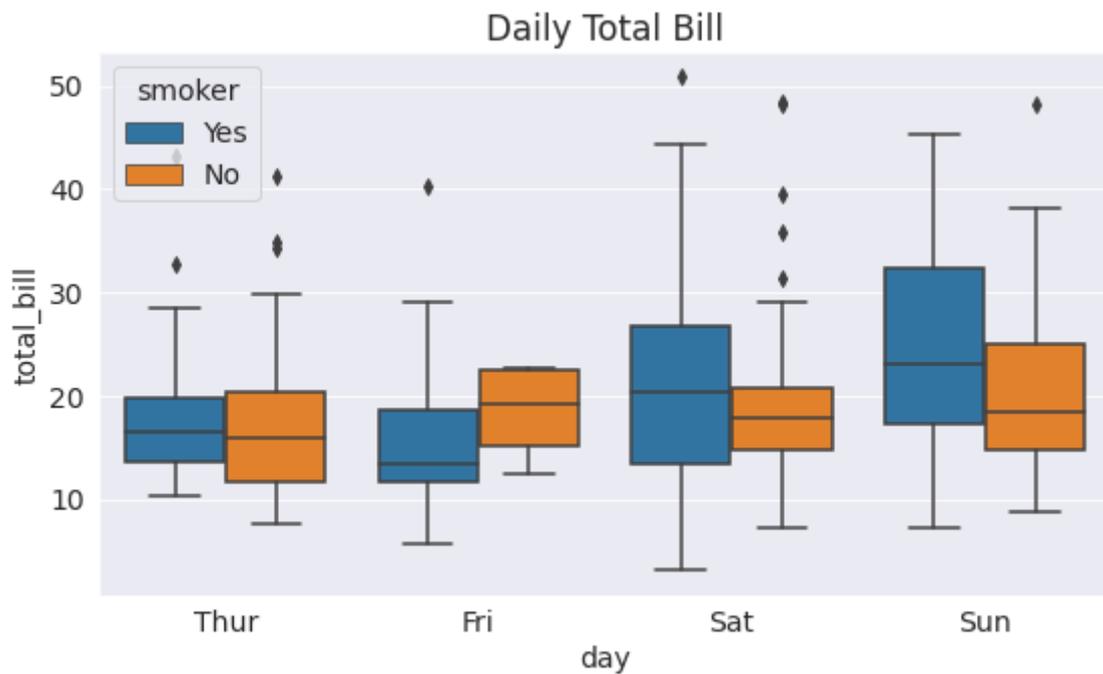
	total_bill	tip	sex	smoker	day	time	size
4	24.59	3.61	Female	No	Sun	Dinner	4
...
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

244 rows × 7 columns

We can use a box plot to visualize the distribution of total bill for each day of the week, segmented by whether the customer was a smoker.

```
# Chart title
plt.title("Daily Total Bill")

# Draw a nested boxplot to show bills by day and time
sns.boxplot(tips.day, tips.total_bill, hue=tips.smoker);
```



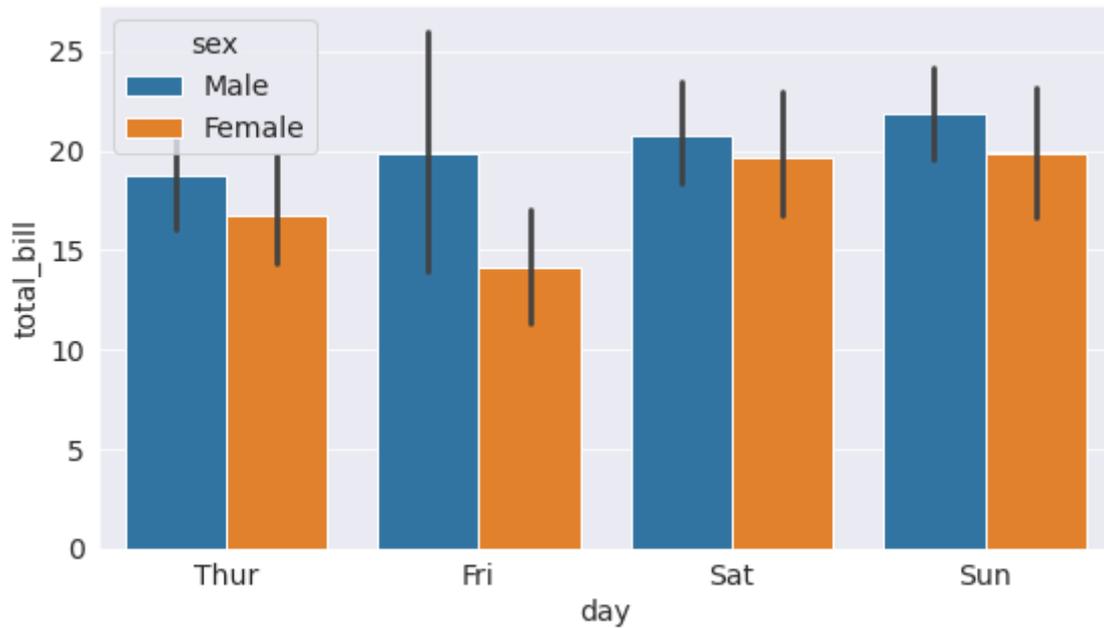
Bar Chart

A bar chart presents categorical data with rectangular bars with heights proportional to the values that they represent. If there are multiple values for each category, then a bar plot can also represent the average value, with confidence intervals.

Example

We can use a bar chart visualize the average value of total bill for different days of the week, segmented by sex, for the "tips" dataset

```
sns.barplot(x="day", y="total_bill", hue="sex", data=tips);
```



Further Reading

This guide intends to serve as introduction to the most commonly used data visualization techniques. With minor modifications to the examples shown above, you can visualize a wide variety of datasets. Visit the official documentation websites for more examples & tutorials:

- Seaborn: <https://seaborn.pydata.org/tutorial.html>
- Matplotlib: <https://matplotlib.org/tutorials/index.html>

To share your data visualizations online, just install the Jovian python library and run `jovian.commit`.

```
!pip install jovian --upgrade --quiet
```

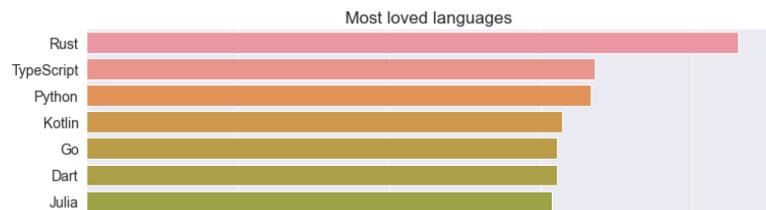
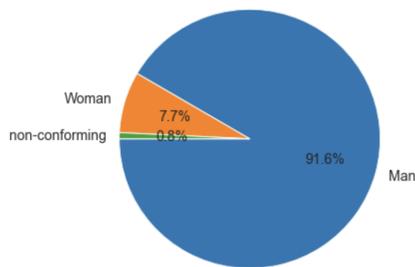
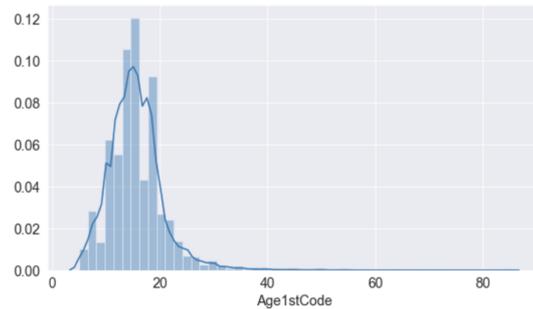
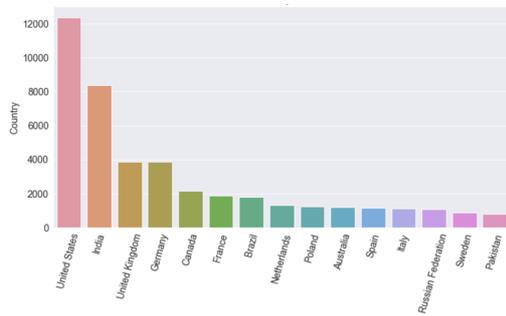
```
import jovian
```

```
jovian.commit(project='dataviz-cheatsheet')
```

```
[jovian] Attempting to save notebook..
```

Exploratory Data Analysis using Python - A Case Study

Analyzing responses from the Stack Overflow Annual Developer Survey 2020



Part 9 of "Data Analysis with Python: Zero to Pandas"

This tutorial series is a beginner-friendly introduction to programming and data analysis using the Python programming language. These tutorials take a practical and coding-focused approach. The best way to learn the material is to execute the code and experiment with it yourself. Check out the full series here:

- [1. First Steps with Python and Jupyter](#)
- [2. A Quick Tour of Variables and Data Types](#)
- [3. Branching using Conditional Statements and Loops](#)
- [4. Writing Reusable Code Using Functions](#)
- [5. Reading from and Writing to Files](#)
- [6. Numerical Computing with Python and Numpy](#)
- [7. Analyzing Tabular Data using Pandas](#)
- [8. Data Visualization using Matplotlib & Seaborn](#)
- [9. Exploratory Data Analysis - A Case Study](#)

The following topics are covered in this tutorial:

- Selecting and downloading a dataset
- Data preparation and cleaning
- Exploratory analysis and visualization
- Asking and answering interesting questions
- Summarizing inferences and drawing conclusions

How to run the code

This tutorial is an executable [Jupyter notebook](#) hosted on [Jovian](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Binder**. You can also select "Run on Colab" or "Run on Kaggle", but you'll need to create an account on [Google Colab](#) or [Kaggle](#) to use these platforms.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Jupyter Notebooks: This tutorial is a [Jupyter notebook](#) - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc., instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things - you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" menu option to clear all outputs and start again from the top.

Introduction

In this tutorial, we'll analyze the StackOverflow developer survey dataset. The dataset contains responses to an annual survey conducted by StackOverflow. You can find the raw data & official analysis here:

<https://insights.stackoverflow.com/survey>.

There are several options for getting the dataset into Jupyter:

- Download the CSV manually and upload it via Jupyter's GUI
- Use the `urlretrieve` function from the `urllib.request` to download CSV files from a raw URL
- Use a helper library, e.g., [opendatasets](#), which contains a collection of curated datasets and provides a helper function for direct download.

We'll use the `opendatasets` helper library to download the files.

```
!pip install jovian opendatasets --upgrade --quiet
```

```
import opendatasets as od
```

```
od.download('stackoverflow-developer-survey-2020')
```

Downloading

<https://raw.githubusercontent.com/JovianML/opendatasets/master/data/stackoverflow->

[developer-survey-2020/survey_results_public.csv](https://raw.githubusercontent.com/JovianML/opendatasets/master/data/stackoverflow-developer-survey-2020/survey_results_public.csv) to `./stackoverflow-developer-survey-2020/survey_results_public.csv`

94609408it [00:03, 24235210.18it/s]

Downloading

https://raw.githubusercontent.com/JovianML/opendatasets/master/data/stackoverflow-developer-survey-2020/survey_results_schema.csv to `./stackoverflow-developer-survey-2020/survey_results_schema.csv`

16384it [00:00, 84844.31it/s]

Downloading

<https://raw.githubusercontent.com/JovianML/opendatasets/master/data/stackoverflow-developer-survey-2020/README.txt> to `./stackoverflow-developer-survey-2020/README.txt`

8192it [00:00, 43205.12it/s]

Let's verify that the dataset was downloaded into the directory `stackoverflow-developer-survey-2020` and retrieve the list of files in the dataset.

```
import os
```

```
os.listdir('stackoverflow-developer-survey-2020')
```

```
['README.txt', 'survey_results_public.csv', 'survey_results_schema.csv']
```

You can through the downloaded files using the "File" > "Open" menu option in Jupyter. It seems like the dataset contains three files:

- `README.txt` - Information about the dataset
- `survey_results_schema.csv` - The list of questions, and shortcodes for each question
- `survey_results_public.csv` - The full list of responses to the questions

Let's load the CSV files using the Pandas library. We'll use the name `survey_raw_df` for the data frame to indicate this is unprocessed data that we might clean, filter, and modify to prepare a data frame ready for analysis.

```
import pandas as pd
```

```
survey_raw_df = pd.read_csv('stackoverflow-developer-survey-2020/survey_results_public.
```

```
survey_raw_df
```

	Respondent	MainBranch	Hobbyist	Age	Age1stCode	CompFreq	CompTotal	ConvertedComp	Country	C
0	1	I am a developer by profession	Yes	NaN	13	Monthly	NaN	NaN	Germany	C

Respondent	MainBranch	Hobbyist	Age	Age1stCode	CompFreq	CompTotal	ConvertedComp	Country	
1	2	I am a developer by profession	No	NaN	19	NaN	NaN	NaN	United Kingdom
2	3	I code primarily as a hobby	Yes	NaN	15	NaN	NaN	NaN	Russian Federation
3	4	I am a developer by profession	Yes	25.0	18	NaN	NaN	NaN	Albania
4	5	I used to be a developer by profession, but no...	Yes	31.0	16	NaN	NaN	NaN	United States
...
64456	64858	NaN	Yes	NaN	16	NaN	NaN	NaN	United States
64457	64867	NaN	Yes	NaN	NaN	NaN	NaN	NaN	Morocco
64458	64898	NaN	Yes	NaN	NaN	NaN	NaN	NaN	Viet Nam
64459	64925	NaN	Yes	NaN	NaN	NaN	NaN	NaN	Poland
64460	65112	NaN	Yes	NaN	NaN	NaN	NaN	NaN	Spain

64461 rows × 61 columns

```
# Opening the schema file with all of the questions and resetting the index
pd.read_csv('stackoverflow-developer-survey-2020/survey_results_schema.csv', index_col
```

Column	QuestionText
Respondent	Randomized respondent ID number (not in order ...
MainBranch	Which of the following options best describes ...
Hobbyist	Do you code as a hobby?
Age	What is your age (in years)? If you prefer not...
Age1stCode	At what age did you write your first line of c...
...	...
WebframeWorkedWith	Which web frameworks have you done extensive d...
WelcomeChange	Compared to last year, how welcome do you feel...
WorkWeekHrs	On average, how many hours per week do you wor...
YearsCode	Including any education, how many years have y...

Column

YearsCodePro NOT including education, how many years have y...

61 rows × 1 columns

```
# Getting the question text from the row 'Respondent'
pd.read_csv('stackoverflow-developer-survey-2020/survey_results_schema.csv',
            index_col = 'Column').loc['Respondent']
```

QuestionText Randomized respondent ID number (not in order ...
Name: Respondent, dtype: object

The dataset contains over 64,000 responses to 60 questions (although many questions are optional). The responses have been anonymized to remove personally identifiable information, and each respondent has been assigned a randomized respondent ID.

Let's view the list of columns in the data frame.

```
survey_raw_df.columns
```

```
Index(['Respondent', 'MainBranch', 'Hobbyist', 'Age', 'Age1stCode', 'CompFreq',
      'CompTotal', 'ConvertedComp', 'Country', 'CurrencyDesc',
      'CurrencySymbol', 'DatabaseDesireNextYear', 'DatabaseWorkedWith',
      'DevType', 'EdLevel', 'Employment', 'Ethnicity', 'Gender', 'JobFactors',
      'JobSat', 'JobSeek', 'LanguageDesireNextYear', 'LanguageWorkedWith',
      'MiscTechDesireNextYear', 'MiscTechWorkedWith',
      'NEWCollabToolsDesireNextYear', 'NEWCollabToolsWorkedWith', 'NEWDevOps',
      'NEWDevOpsImpt', 'NEWEdImpt', 'NEWJobHunt', 'NEWJobHuntResearch',
      'NEWLearn', 'NEWOffTopic', 'NEWOnboardGood', 'NEWOtherComms',
      'NEWOvertime', 'NEWPurchaseResearch', 'NEWPurpleLink', 'NEWSOSites',
      'NEWStuck', 'OpSys', 'OrgSize', 'PlatformDesireNextYear',
      'PlatformWorkedWith', 'PurchaseWhat', 'Sexuality', 'SOAccount',
      'SOComm', 'SOPartFreq', 'SOVisitFreq', 'SurveyEase', 'SurveyLength',
      'Trans', 'UndergradMajor', 'WebframeDesireNextYear',
      'WebframeWorkedWith', 'WelcomeChange', 'WorkWeekHrs', 'YearsCode',
      'YearsCodePro'],
      dtype='object')
```

It appears that shortcodes for questions have been used as column names.

We can refer to the schema file to see the full text of each question. The schema file contains only two columns: `Column` and `QuestionText`. We can load it as Pandas Series with `Column` as the index and the `QuestionText` as the value.

```
schema_fname = 'stackoverflow-developer-survey-2020/survey_results_schema.csv'
schema_raw = pd.read_csv(schema_fname, index_col='Column').QuestionText
```

^^^ Getting just the QuestionText columns which gives a series with index column name and value, which is the full text of the question.

```
schema_raw
```

Column

Respondent	Randomized respondent ID number (not in order ...
MainBranch	Which of the following options best describes ...
Hobbyist	Do you code as a hobby?
Age	What is your age (in years)? If you prefer not...
Age1stCode	At what age did you write your first line of c...
	...
WebframeWorkedWith	Which web frameworks have you done extensive d...
WelcomeChange	Compared to last year, how welcome do you feel...
WorkWeekHrs	On average, how many hours per week do you wor...
YearsCode	Including any education, how many years have y...
YearsCodePro	NOT including education, how many years have y...

Name: QuestionText, Length: 61, dtype: object

We can now use `schema_raw` to retrieve the full question text for any column in `survey_raw_df`.

```
schema_raw['YearsCodePro']
```

```
'NOT including education, how many years have you coded professionally (as a part of your work)?'
```

We've now loaded the dataset. We're ready to move on to the next step of preprocessing & cleaning the data for our analysis.

Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. [Jovian](#) offers an easy way of saving and sharing your Jupyter notebooks online.

```
# Select a project name
project='python-eda-stackoverflow-survey'
```

```
# Install the Jovian library
!pip install jovian --upgrade --quiet
```

```
import jovian
```

```
jovian.commit(project=project)
```

[jovian] Updating notebook "evanmarie/python-eda-stackoverflow-survey" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-eda-stackoverflow-survey>

```
'https://jovian.ai/evanmarie/python-eda-stackoverflow-survey'
```

The first time you run `jovian.commit`, you'll be asked to provide an API Key to securely upload the notebook to your Jovian account. You can get the API key from your [Jovian profile page](#) after logging in / signing up.

`jovian.commit` uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work.

Data Preparation & Cleaning

While the survey responses contain a wealth of information, we'll limit our analysis to the following areas:

- Demographics of the survey respondents and the global programming community
- Distribution of programming skills, experience, and preferences
- Employment-related information, preferences, and opinions

Let's select a subset of columns with the relevant data for our analysis.

```
selected_columns = [  
    # Demographics  
    'Country',  
    'Age',  
    'Gender',  
    'EdLevel',  
    'UndergradMajor',  
    # Programming experience  
    'Hobbyist',  
    'Age1stCode',  
    'YearsCode',  
    'YearsCodePro',  
    'LanguageWorkedWith',  
    'LanguageDesireNextYear',  
    'NEWLearn',  
    'NEWStuck',  
    # Employment  
    'Employment',  
    'DevType',  
    'WorkWeekHrs',  
    'JobSat',  
    'JobFactors',  
    'NEWOvertime',  
    'NEWEdImpt'  
]
```

```
len(selected_columns)
```

Let's extract a copy of the data from these columns into a new data frame `survey_df` . We can continue to modify further without affecting the original data frame.

```
survey_df = survey_raw_df[selected_columns].copy()
```

```
schema = schema_raw[selected_columns]
```

Let's view some basic information about the data frame.

```
survey_df.shape
```

```
(64461, 20)
```

```
survey_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
RangeIndex: 64461 entries, 0 to 64460
```

```
Data columns (total 20 columns):
```

#	Column	Non-Null Count	Dtype
0	Country	64072 non-null	object
1	Age	45446 non-null	float64
2	Gender	50557 non-null	object
3	EdLevel	57431 non-null	object
4	UndergradMajor	50995 non-null	object
5	Hobbyist	64416 non-null	object
6	Age1stCode	57900 non-null	object
7	YearsCode	57684 non-null	object
8	YearsCodePro	46349 non-null	object
9	LanguageWorkedWith	57378 non-null	object
10	LanguageDesireNextYear	54113 non-null	object
11	NEWLearn	56156 non-null	object
12	NEWStuck	54983 non-null	object
13	Employment	63854 non-null	object
14	DevType	49370 non-null	object
15	WorkWeekHrs	41151 non-null	float64
16	JobSat	45194 non-null	object
17	JobFactors	49349 non-null	object
18	NEWOvertime	43231 non-null	object
19	NEWEdImp	48465 non-null	object

```
dtypes: float64(2), object(18)
```

```
memory usage: 9.8+ MB
```

Most columns have the data type `object`, either because they contain values of different types or contain empty values (`NaN`). It appears that every column contains some empty values since the Non-Null count for every column is lower than the total number of rows (64461). We'll need to deal with empty values and manually adjust the data type for each column on a case-by-case basis.

Only two of the columns were detected as numeric columns (`Age` and `WorkWeekHrs`), even though a few other columns have mostly numeric values. To make our analysis easier, let's convert some other columns into numeric data types while ignoring any non-numeric value. The non-numeric are converted to `NaN`.

```
survey_df['Age1stCode'] = pd.to_numeric(survey_df.Age1stCode, errors='coerce')
survey_df['YearsCode'] = pd.to_numeric(survey_df.YearsCode, errors='coerce')
survey_df['YearsCodePro'] = pd.to_numeric(survey_df.YearsCodePro, errors='coerce')
```

Let's now view some basic statistics about numeric columns.

```
survey_df.describe()
```

	Age	Age1stCode	YearsCode	YearsCodePro	WorkWeekHrs
count	45446.000000	57473.000000	56784.000000	44133.000000	41151.000000
mean	30.834111	15.476572	12.782051	8.869667	40.782174
std	9.585392	5.114081	9.490657	7.759961	17.816383
min	1.000000	5.000000	1.000000	1.000000	1.000000
25%	24.000000	12.000000	6.000000	3.000000	40.000000
50%	29.000000	15.000000	10.000000	6.000000	40.000000
75%	35.000000	18.000000	17.000000	12.000000	44.000000
max	279.000000	85.000000	50.000000	50.000000	475.000000

There seems to be a problem with the age column, as the minimum value is 1 and the maximum is 279. This is a common issue with surveys: responses may contain invalid values due to accidental or intentional errors while responding. A simple fix would be to ignore the rows where the age is higher than 100 years or lower than 10 years as invalid survey responses. We can do this using the `.drop` method, [as explained here](#).

```
survey_df.drop(survey_df[survey_df.Age < 10].index, inplace=True)
survey_df.drop(survey_df[survey_df.Age > 100].index, inplace=True)
```

The same holds for `WorkWeekHrs`. Let's ignore entries where the value for the column is higher than 140 hours (~20 hours per day).

```
survey_df.drop(survey_df[survey_df.WorkWeekHrs > 140].index, inplace=True)
```

The gender column also allows for picking multiple options. We'll remove values containing more than one option to simplify our analysis.

```
survey_df['Gender'].value_counts()
```

Man	45895
Woman	3835

```

Non-binary, genderqueer, or gender non-conforming      385
Man;Non-binary, genderqueer, or gender non-conforming  121
Woman;Non-binary, genderqueer, or gender non-conforming 92
Woman;Man                                               73
Woman;Man;Non-binary, genderqueer, or gender non-conforming 25
Name: Gender, dtype: int64

```

```
import numpy as np
```

```
survey_df.where(~(survey_df.Gender.str.contains('; ', na=False)), np.nan, inplace=True)
```

We've now cleaned up and prepared the dataset for analysis. Let's take a look at a sample of rows from the data frame.

```
survey_df.sample(10)
```

	Country	Age	Gender	EdLevel	UndergradMajor	Hobbyist	Age1stCode	YearsCode	YearsCodePro
5484	United States	29.0	Man	Bachelor's degree (B.A., B.S., B.Eng., etc.)	NaN	Yes	8.0	4.0	NaN
48777	United States	32.0	Man	Master's degree (M.A., M.S., M.Eng., MBA, etc.)	A social science (such as anthropology, psycho...	Yes	15.0	13.0	7.0
42005	United States	NaN	NaN	Bachelor's degree (B.A., B.S., B.Eng., etc.)	NaN	Yes	NaN	NaN	NaN
48382	India	NaN	NaN	NaN	NaN	Yes	NaN	NaN	NaN
42493	Switzerland	NaN	NaN	NaN	NaN	Yes	NaN	NaN	NaN
46079	Germany	22.0	Man	Secondary school (e.g. American high school, G...	NaN	Yes	14.0	8.0	NaN Ba
46805	United Kingdom	28.0	Man	Bachelor's degree (B.A., B.S., B.Eng., etc.)	Information systems, information technology, o...	Yes	18.0	8.0	4.0

	Country	Age	Gender	EdLevel	UndergradMajor	Hobbyist	Age1stCode	YearsCode	YearsCodePro	
3979	Iceland	37.0	Man	Master's degree (M.A., M.S., M.Eng., MBA, etc.)	Computer science, computer engineering, or sof...	Yes	13.0	24.0	11.0	B&
56635	Brazil	30.0	Man	Associate degree (A.A., A.S., etc.)	Computer science, computer engineering, or sof...	No	17.0	13.0	11.0	
55689	India	21.0	Man	Bachelor's degree (B.A., B.S., B.Eng., etc.)	Computer science, computer engineering, or sof...	Yes	17.0	4.0	NaN	

Let's save and commit our work before continuing.

```
import jovian
```

```
jovian.commit()
```

[jovian] Updating notebook "evanmarie/python-eda-stackoverflow-survey" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/evanmarie/python-eda-stackoverflow-survey>

```
'https://jovian.ai/evanmarie/python-eda-stackoverflow-survey'
```

Exploratory Analysis and Visualization

Before we ask questions about the survey responses, it would help to understand the respondents' demographics, i.e., country, age, gender, education level, employment level, etc. It's essential to explore these variables to understand how representative the survey is of the worldwide programming community. A survey of this scale generally tends to have some [selection bias](#).

Let's begin by importing `matplotlib.pyplot` and `seaborn`.

```
import seaborn as sns
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

sns.set_style('darkgrid')
matplotlib.rcParams['font.size'] = 14
matplotlib.rcParams['figure.figsize'] = (9, 5)
matplotlib.rcParams['figure.facecolor'] = '#00000000'
```

Country

Let's look at the number of countries from which there are responses in the survey and plot the ten countries with the highest number of responses.

```
schema.Country
```

```
'Where do you live?'
```

```
survey_df.Country.nunique()
```

```
183
```

We can identify the countries with the highest number of respondents using the `value_counts` method.

```
top_countries = survey_df.Country.value_counts().head(15)
top_countries
```

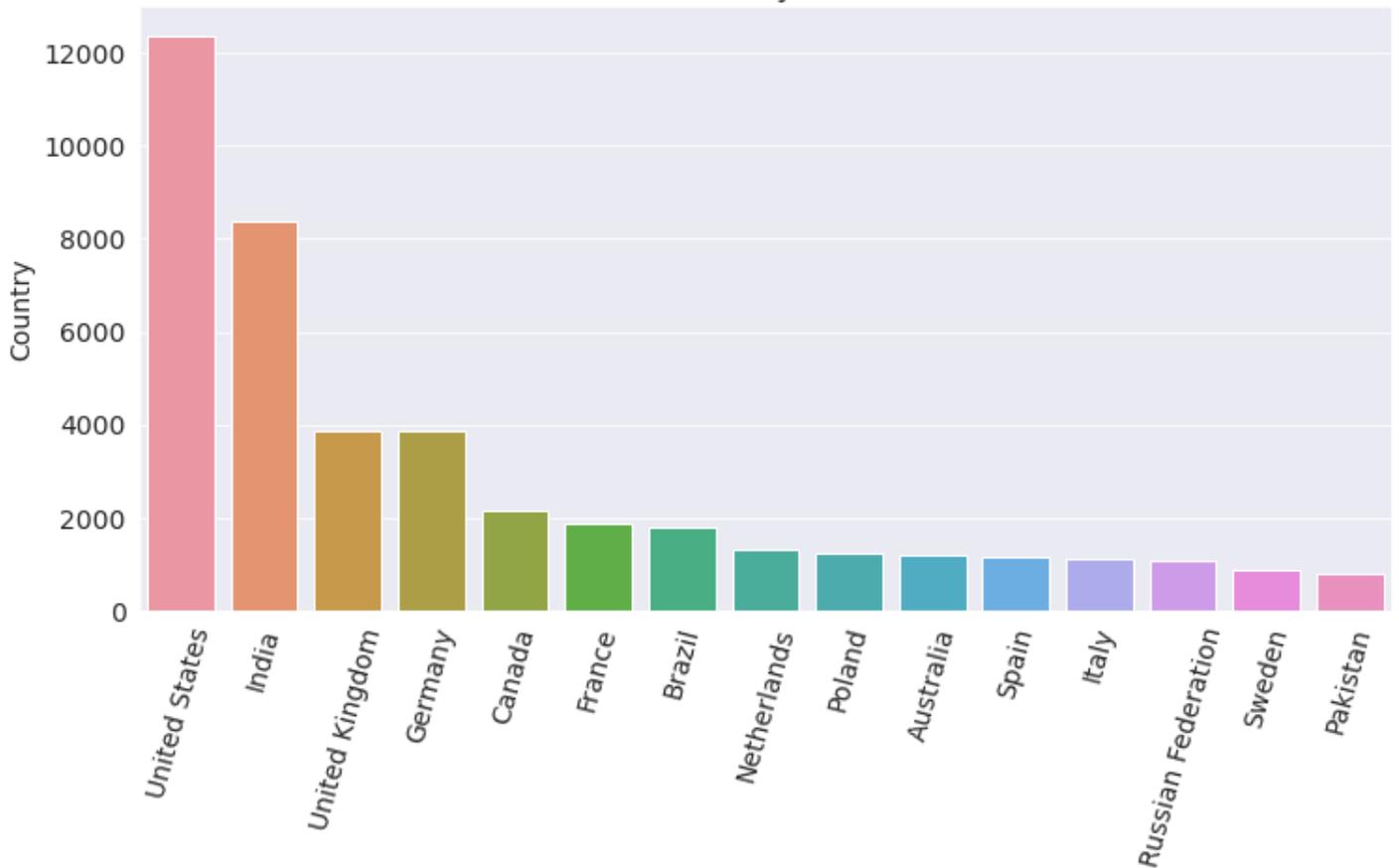
United States	12371
India	8364
United Kingdom	3881
Germany	3864
Canada	2175
France	1884
Brazil	1804
Netherlands	1332
Poland	1259
Australia	1199
Spain	1157
Italy	1115
Russian Federation	1085
Sweden	879
Pakistan	802

Name: Country, dtype: int64

We can visualize this information using a bar chart.

```
plt.figure(figsize=(12,6))
plt.xticks(rotation=75)
plt.title(schema.Country)
sns.barplot(x=top_countries.index, y=top_countries);
```

Where do you live?



It appears that a disproportionately high number of respondents are from the US and India, probably because the survey is in English, and these countries have the highest English-speaking populations. We can already see that the survey may not be representative of the global programming community - especially from non-English speaking countries. Programmers from non-English speaking countries are almost certainly underrepresented.

Exercise: Try finding the percentage of responses from English-speaking vs. non-English speaking countries. You can use [this list of languages spoken in different countries](#).

```
survey_df['english_speaking'] = np.where(survey_df['Country'].isin(["United States", "U
# variable name = the dataframe, the column in the df, if that field is yes, the column
english_speakers = survey_df[survey_df['english_speaking'] == 'yes']['english_speaking']
english_speakers
```

19626

```
print(f"There are at least {english_speakers:,} respondents who probably speak English.
```

There are at least 19,626 respondents who probably speak English.

```
total_respondents = len(survey_df)
english_percentage = english_speakers / total_respondents * 100
```

```
print(f"The total percentage of English speaking respondents is {english_percentage:.2f}
```

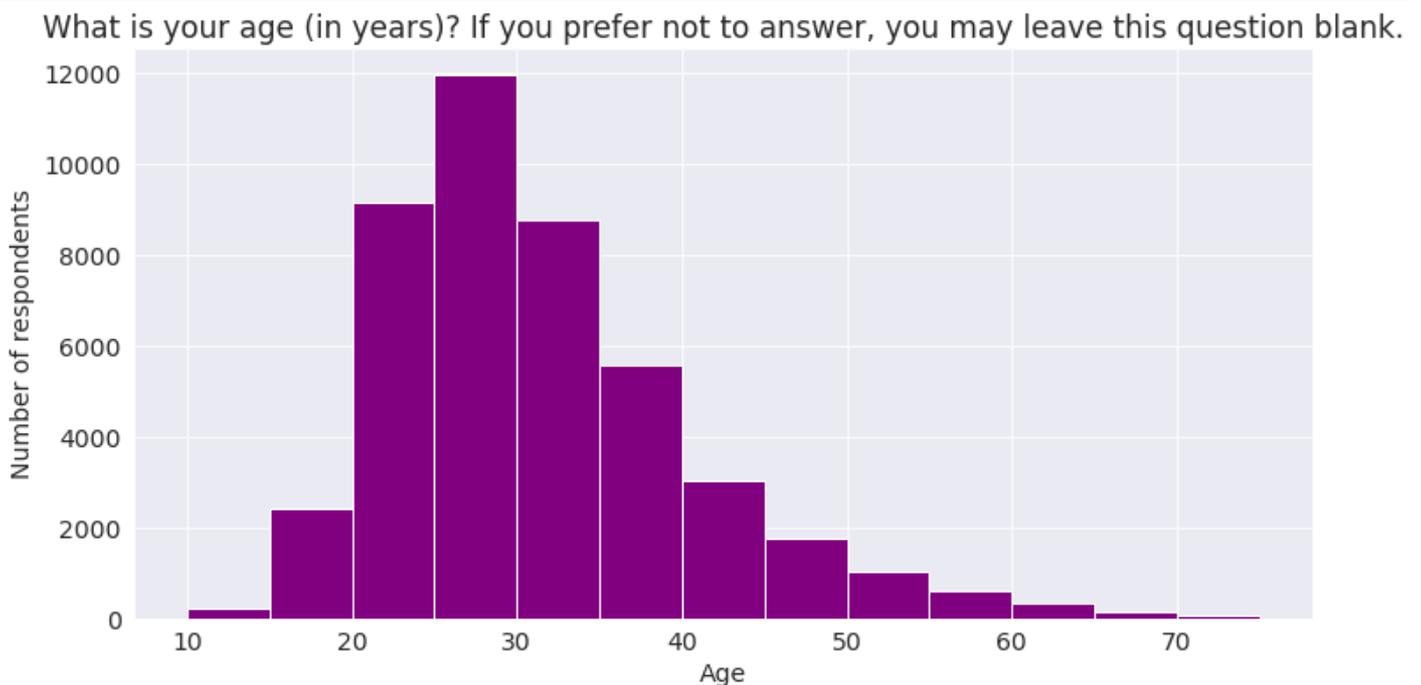
The total percentage of English speaking respondents is 30.52%

Age

The distribution of respondents' age is another crucial factor to look at. We can use a histogram to visualize it.

```
plt.figure(figsize=(12, 6))
plt.title(schema.Age)
plt.xlabel('Age')
plt.ylabel('Number of respondents')

plt.hist(survey_df.Age, bins=np.arange(10,80,5), color='purple');
```



It appears that a large percentage of respondents are 20-45 years old. It's somewhat representative of the programming community in general. Many young people have taken up computer science as their field of study or profession in the last 20 years.

Exercise: You may want to filter out responses by age (or age group) if you'd like to analyze and compare the survey results for different age groups. Create a new column called `AgeGroup` containing values like `Less than 10 years`, `10-18 years`, `18-30 years`, `30-45 years`, `45-60 years` and `Older than 60 years`. Then, repeat the analysis in the rest of this notebook for each age group.

Gender

Let's look at the distribution of responses for the Gender. It's a well-known fact that women and non-binary genders are underrepresented in the programming community, so we might expect to see a skewed distribution here.

```
schema.Gender
```

'Which of the following describe you, if any? Please check all that apply. If you

prefer not to answer, you may leave this question blank.'

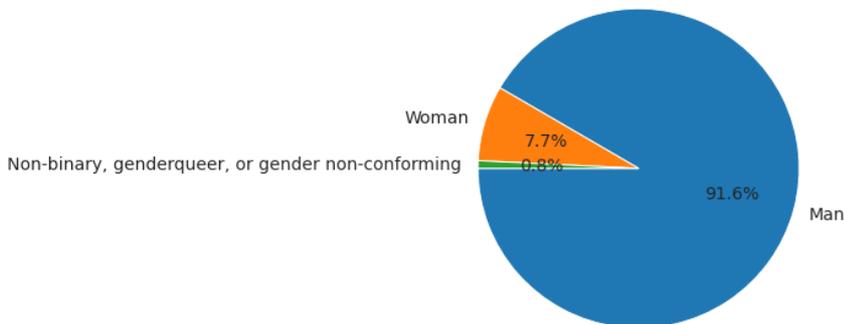
```
gender_counts = survey_df.Gender.value_counts()
gender_counts
```

```
Man                45895
Woman              3835
Non-binary, genderqueer, or gender non-conforming    385
Name: Gender, dtype: int64
```

A pie chart would be a great way to visualize the distribution.

```
plt.figure(figsize=(12,6))
plt.title(schema.Gender)
plt.pie(gender_counts, labels=gender_counts.index, autopct='%1.1f%%', startangle=180);
```

Which of the following describe you, if any? Please check all that apply. If you prefer not to answer, you may leave this question blank.



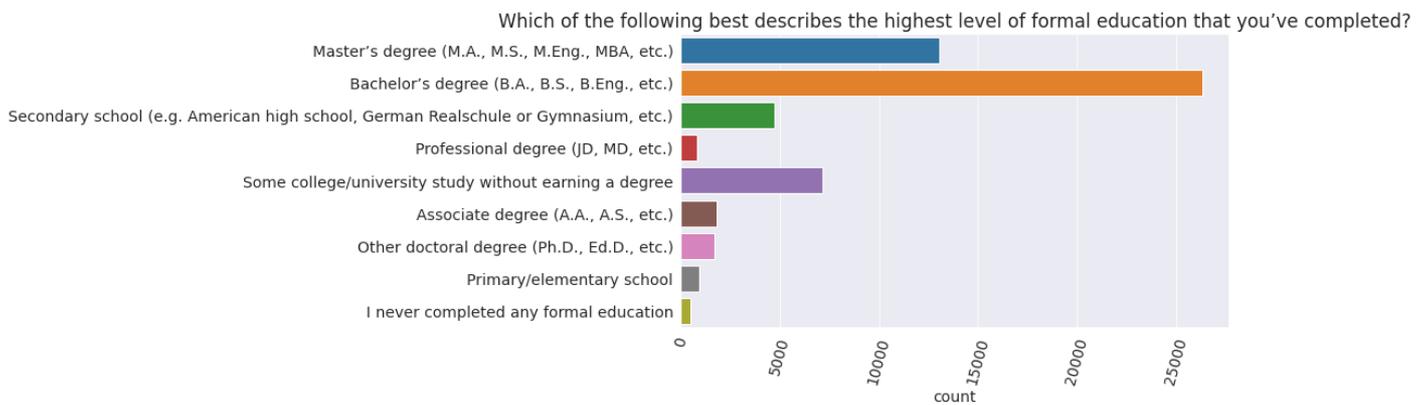
Only about 8% of survey respondents who have answered the question identify as women or non-binary. This number is lower than the overall percentage of women & non-binary genders in the programming community - which is estimated to be around 12%.

Exercise: It would be interesting to compare the survey responses & preferences across genders. Repeat this analysis with these breakdowns. How do the relative education levels differ across genders? How do the salaries vary? You may find this analysis on the [Gender Divide in Data Science](#) useful.

Education Level

Formal education in computer science is often considered an essential requirement for becoming a programmer. However, there are many free resources & tutorials available online to learn programming. Let's compare the education levels of respondents to gain some insight into this. We'll use a horizontal bar plot here.

```
sns.countplot(y=survey_df.EdLevel)
plt.xticks(rotation=75);
plt.title(schema['EdLevel'])
plt.ylabel(None);
```



It appears that well over half of the respondents hold a bachelor's or master's degree, so most programmers seem to have some college education. However, it's not clear from this graph alone if they hold a degree in computer science.

Exercises: The graph currently shows the number of respondents for each option. Can you modify it to show the percentage instead? Further, try comparing the percentages for each degree for men vs. women.

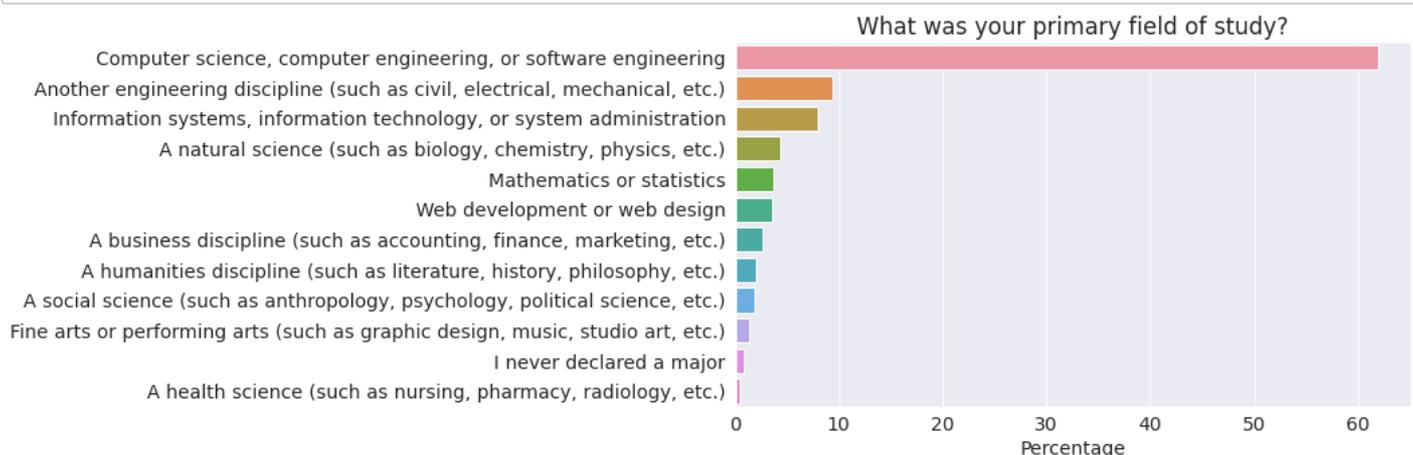
Let's also plot undergraduate majors, but this time we'll convert the numbers into percentages and sort the values to make it easier to visualize the order.

```
schema.UndergradMajor
```

```
'What was your primary field of study?'
```

```
undergrad_pct = survey_df.UndergradMajor.value_counts() * 100 / survey_df.UndergradMajor.value_counts().sum()
sns.barplot(x=undergrad_pct, y=undergrad_pct.index)

plt.title(schema.UndergradMajor)
plt.ylabel(None);
plt.xlabel('Percentage');
```



It turns out that 40% of programmers holding a college degree have a field of study other than computer science - which is very encouraging. It seems to suggest that while a college education is helpful in general, you do not need to pursue a major in computer science to become a successful programmer.

Exercises: Analyze the `NEWEdImpt` column for respondents who hold some college degree vs. those who don't. Do you notice any difference in opinion?

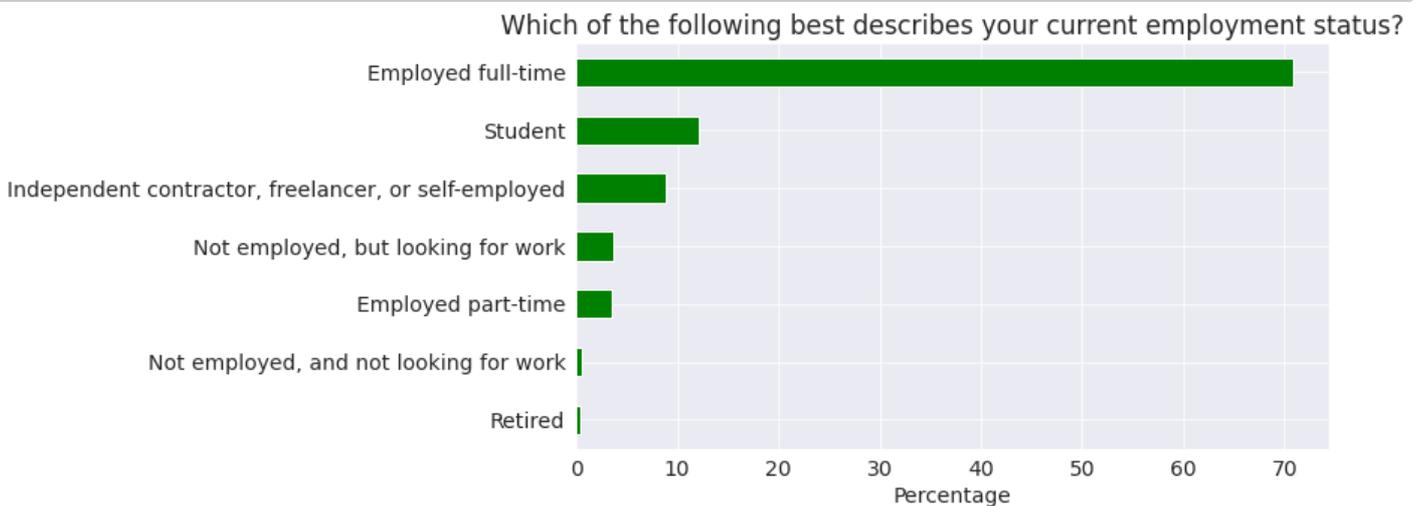
Employment

Freelancing or contract work is a common choice among programmers, so it would be interesting to compare the breakdown between full-time, part-time, and freelance work. Let's visualize the data from the `Employment` column.

```
schema.Employment
```

```
'Which of the following best describes your current employment status?'
```

```
(survey_df.Employment.value_counts(normalize=True, ascending=True)*100).plot(kind='barh')
plt.title(schema.Employment)
plt.xlabel('Percentage');
```



```
survey_df.DevType.value_counts()
```

```
Developer, full-stack
```

```
4396
```

```
Developer, back-end
```

```
3056
```

```
Developer, back-end;Developer, front-end;Developer, full-stack
```

```
2214
```

```
Developer, back-end;Developer, full-stack
```

```
1465
```

```
Developer, front-end
```

```
1390
```

```
...
```

```
Database administrator;Developer, back-end;Developer, front-end;Developer, full-stack;Developer, QA or test;Senior executive/VP
```

```
1
```

```
Database administrator;Developer, back-end;Developer, front-end;Developer, full-stack;Product manager;Senior executive/VP
```

```
1
```

```
Developer, back-end;Developer, full-stack;Developer, mobile;DevOps specialist;Educator;System administrator
```

1

```
Data or business analyst;Database administrator;Developer, back-end;Developer, desktop or enterprise applications;Developer, front-end;Developer, mobile;Engineering manager
```

1

```
Data or business analyst;Developer, mobile;Senior executive/VP;System administrator
```

1

```
Name: DevType, Length: 8213, dtype: int64
```

It appears that close to 10% of respondents are employed part time or as freelancers.

Exercise: Add a new column `EmploymentType` containing the values `Enthusiast` (student or not employed but looking for work), `Professional` (employed full-time, part-time or freelancing), and `Other` (not employed or retired). For each of the graphs that follow, show a comparison between `Enthusiast` and `Professional`.

The `DevType` field contains information about the roles held by respondents. Since the question allows multiple answers, the column contains lists of values separated by a semi-colon `;`, making it a bit harder to analyze directly.

```
schema.DevType
```

```
'Which of the following describe you? Please select all that apply.'
```

```
survey_df.DevType.value_counts()
```

```
Developer, full-stack
```

```
4396
```

```
Developer, back-end
```

```
3056
```

```
Developer, back-end;Developer, front-end;Developer, full-stack
```

```
2214
```

```
Developer, back-end;Developer, full-stack
```

```
1465
```

```
Developer, front-end
```

```
1390
```

```
...
```

```
Database administrator;Developer, back-end;Developer, front-end;Developer, full-stack;Developer, QA or test;Senior executive/VP
```

1

```
Database administrator;Developer, back-end;Developer, front-end;Developer, full-stack;Product manager;Senior executive/VP
```

1

```
Developer, back-end;Developer, full-stack;Developer, mobile;DevOps specialist;Educator;System administrator
```

1

Data or business analyst;Database administrator;Developer, back-end;Developer, desktop or enterprise applications;Developer, front-end;Developer, mobile;Engineering manager
 1
 Data or business analyst;Developer, mobile;Senior executive/VP;System administrator
 1
 Name: DevType, Length: 8213, dtype: int64

Let's define a helper function that turns a column containing lists of values (like `survey_df.DevType`) into a data frame with one column for each possible option.

```
def split_multicolumn(col_series):
    result_df = col_series.to_frame()
    options = []
    # Iterate over the column
    for idx, value in col_series[col_series.notnull()].iteritems():
        # Break each value into list of options
        for option in value.split(';'):
            # Add the option as a column to result
            if not option in result_df.columns:
                options.append(option)
                result_df[option] = False
            # Mark the value in the option column as True
            result_df.at[idx, option] = True
    return result_df[options]
```

```
dev_type_df = split_multicolumn(survey_df.DevType)
```

```
dev_type_df
```

	Developer, desktop or enterprise applications	Developer, full-stack	Developer, mobile	Designer	Developer, front-end	Developer, back-end	Developer, QA or test	DevOps specialist	Developer, game or graphics	Dat adminis
0	True	True	False	False	False	False	False	False	False	
1	False	True	True	False	False	False	False	False	False	
2	False	False	False	False	False	False	False	False	False	
3	False	False	False	False	False	False	False	False	False	
4	False	False	False	False	False	False	False	False	False	
...	
64456	False	False	False	False	False	False	False	False	False	
64457	False	False	False	False	False	False	False	False	False	
64458	False	False	False	False	False	False	False	False	False	
64459	False	False	False	False	False	False	False	False	False	
64460	False	False	False	False	False	False	False	False	False	

64306 rows x 23 columns

The `dev_type_df` has one column for each option that can be selected as a response. If a respondent has chosen an option, the corresponding column's value is `True` . Otherwise, it is `False` .

We can now use the column-wise totals to identify the most common roles.

```
dev_type_totals = dev_type_df.sum().sort_values(ascending=False)
dev_type_totals
```

Developer, back-end	26996
Developer, full-stack	26915
Developer, front-end	18128
Developer, desktop or enterprise applications	11687
Developer, mobile	9406
DevOps specialist	5915
Database administrator	5658
Designer	5262
System administrator	5185
Developer, embedded applications or devices	4701
Data or business analyst	3970
Data scientist or machine learning specialist	3939
Developer, QA or test	3893
Engineer, data	3700
Academic researcher	3502
Educator	2895
Developer, game or graphics	2751
Engineering manager	2699
Product manager	2471
Scientist	2060
Engineer, site reliability	1921
Senior executive/VP	1292
Marketing or sales professional	625

dtype: int64

As one might expect, the most common roles include "Developer" in the name.

Exercises:

- Can you figure out what percentage of respondents work in roles related to data science?
- Which positions have the highest percentage of women?

We've only explored a handful of columns from the 20 columns that we selected. Explore and visualize the remaining columns using the empty cells below.

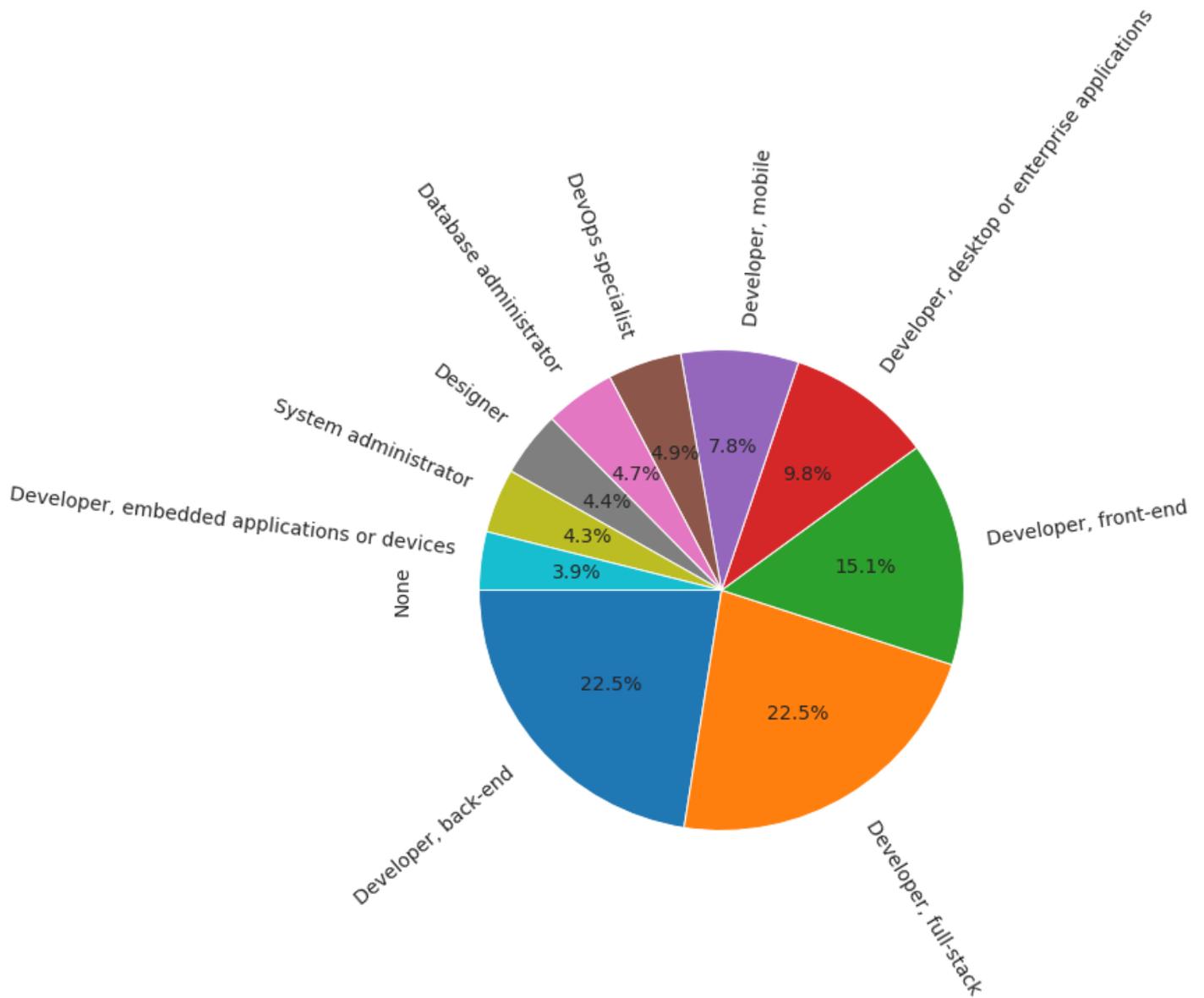
```
dev_type_pct = dev_type_totals / len(survey_df) * 100
dev_type_pct
```

Developer, back-end	41.980531
---------------------	-----------

Developer, full-stack	41.854570
Developer, front-end	28.190216
Developer, desktop or enterprise applications	18.174043
Developer, mobile	14.626940
DevOps specialist	9.198209
Database administrator	8.798557
Designer	8.182751
System administrator	8.063011
Developer, embedded applications or devices	7.310360
Data or business analyst	6.173607
Data scientist or machine learning specialist	6.125400
Developer, QA or test	6.053867
Engineer, data	5.753740
Academic researcher	5.445837
Educator	4.501913
Developer, game or graphics	4.277983
Engineering manager	4.197120
Product manager	3.842565
Scientist	3.203434
Engineer, site reliability	2.987280
Senior executive/VP	2.009144
Marketing or sales professional	0.971916

dtype: float64

```
# Create a pie chart of the top 10 developer types  
wedgeprops = {"linewidth":3, "edgecolor":"black"} # Darker lines  
dev_type_pct[:10].plot(kind='pie', figsize=(10,8), autopct='%1.1f%%', startangle=180, r
```



```
dev_type_female_df = survey_df[survey_df['Gender']== 'Woman']
```

```
dev_type_female_df["DevType"].value_counts()
```

```
Developer, full-stack
384
Developer, back-end
271
Developer, front-end
235
Developer, back-end;Developer, front-end;Developer, full-stack
141
Developer, mobile
105
...
Database administrator;Designer;Developer, desktop or enterprise applications;System
administrator
1
Designer;Developer, embedded applications or devices;Developer, mobile
```

```
1
Developer, back-end;Developer, QA or test;Educator;Engineer, data;Engineering
manager;Product manager
1
Data or business analyst;Data scientist or machine learning specialist;Developer, back-
end;Developer, desktop or enterprise applications;Developer, embedded applications or
devices;Developer, front-end      1
Academic researcher;Data or business analyst;Data scientist or machine learning
specialist;Database administrator;Designer;Engineer, data;Engineering manager;Senior
executive/VP;System administrator      1
Name: DevType, Length: 814, dtype: int64
```

Let's save and upload our work before continuing.

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Updating notebook "evanmarie/python-eda-stackoverflow-survey" on
https://jovian.ai
[jovian] Committed successfully! https://jovian.ai/evanmarie/python-eda-stackoverflow-survey
'https://jovian.ai/evanmarie/python-eda-stackoverflow-survey'
```

Asking and Answering Questions

We've already gained several insights about the respondents and the programming community by exploring individual columns of the dataset. Let's ask some specific questions and try to answer them using data frame operations and visualizations.

Q: What are the most popular programming languages in 2020?

To answer, this we can use the `LanguageWorkedWith` column. Similar to `DevType`, respondents were allowed to choose multiple options here.

```
survey_df.LanguageWorkedWith
```

```
0          C#;HTML/CSS;JavaScript
1          JavaScript;Swift
2    Objective-C;Python;Swift
3                          NaN
4          HTML/CSS;Ruby;SQL
...
64456                          NaN
64457  Assembly;Bash/Shell/PowerShell;C;C#;C++;Dart;G...
64458                          NaN
64459          HTML/CSS
64460  C#;HTML/CSS;Java;JavaScript;SQL
Name: LanguageWorkedWith, Length: 64306, dtype: object
```

First, we'll split this column into a data frame containing a column of each language listed in the options.

```
languages_worked_df = split_multicolumn(survey_df.LanguageWorkedWith)
```

```
languages_worked_df.sample(10)
```

	C#	HTML/CSS	JavaScript	Swift	Objective-C	Python	Ruby	SQL	Java	PHP	...	VBA	Perl	Scala
8740	False	True	True	False	False	True	False	True	True	False	...	False	False	False
25747	True	True	True	False	False	False	False	True	False	False	...	False	False	False
638	True	False	False	False	False	False	False	True	False	False	...	False	False	False
18869	True	False	False	False	False	False	False	True	False	False	...	False	False	False
18373	True	True	True	False	False	False	False	True	False	False	...	False	False	False
34682	False	True	True	False	False	False	False	False	False	False	...	False	False	False
25830	False	True	True	False	False	False	False	False	True	False	...	False	False	False
52995	False	False	False	False	False	True	False	True	True	True	...	False	False	False
26229	False	True	True	False	False	True	False	True	True	False	...	False	False	False
21042	False	True	False	False	False	False	True	True	False	False	...	False	False	False

10 rows × 25 columns

It appears that a total of 25 languages were included among the options. Let's aggregate these to identify the percentage of respondents who selected each language.

```
languages_worked_percentages = languages_worked_df.mean().sort_values(ascending=False)  
languages_worked_percentages
```

```
JavaScript          59.893323  
HTML/CSS           55.801947  
SQL                 48.444935  
Python              39.001026  
Java                35.618760  
Bash/Shell/PowerShell 29.239884  
C#                  27.803004  
PHP                 23.130035  
TypeScript          22.461357  
C++                 21.114670  
C                   19.236152  
Go                  7.758219  
Kotlin              6.887382  
Ruby                6.229590  
Assembly            5.447392  
VBA                  5.394520  
Swift               5.226573  
R                   5.064846  
Rust                4.498803
```

Objective-C	3.603085
Dart	3.517557
Scala	3.150561
Perl	2.757130
Haskell	1.861413
Julia	0.782198

dtype: float64

We can plot this information using a horizontal bar chart.

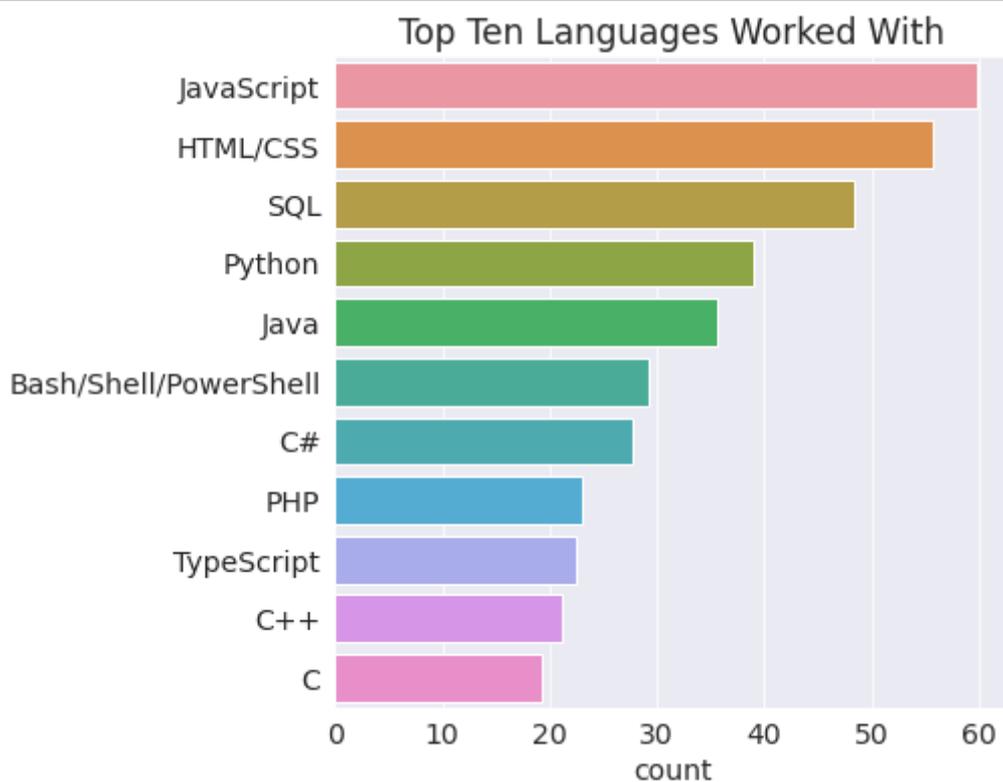
```
# My own plot based on the one below
```

```
plt.figure(figsize=(6,6))
```

```
sns.barplot(x=languages_worked_percentages[:11], y=languages_worked_percentages.index[0
```

```
plt.title("Top Ten Languages Worked With")
```

```
plt.xlabel('count');
```



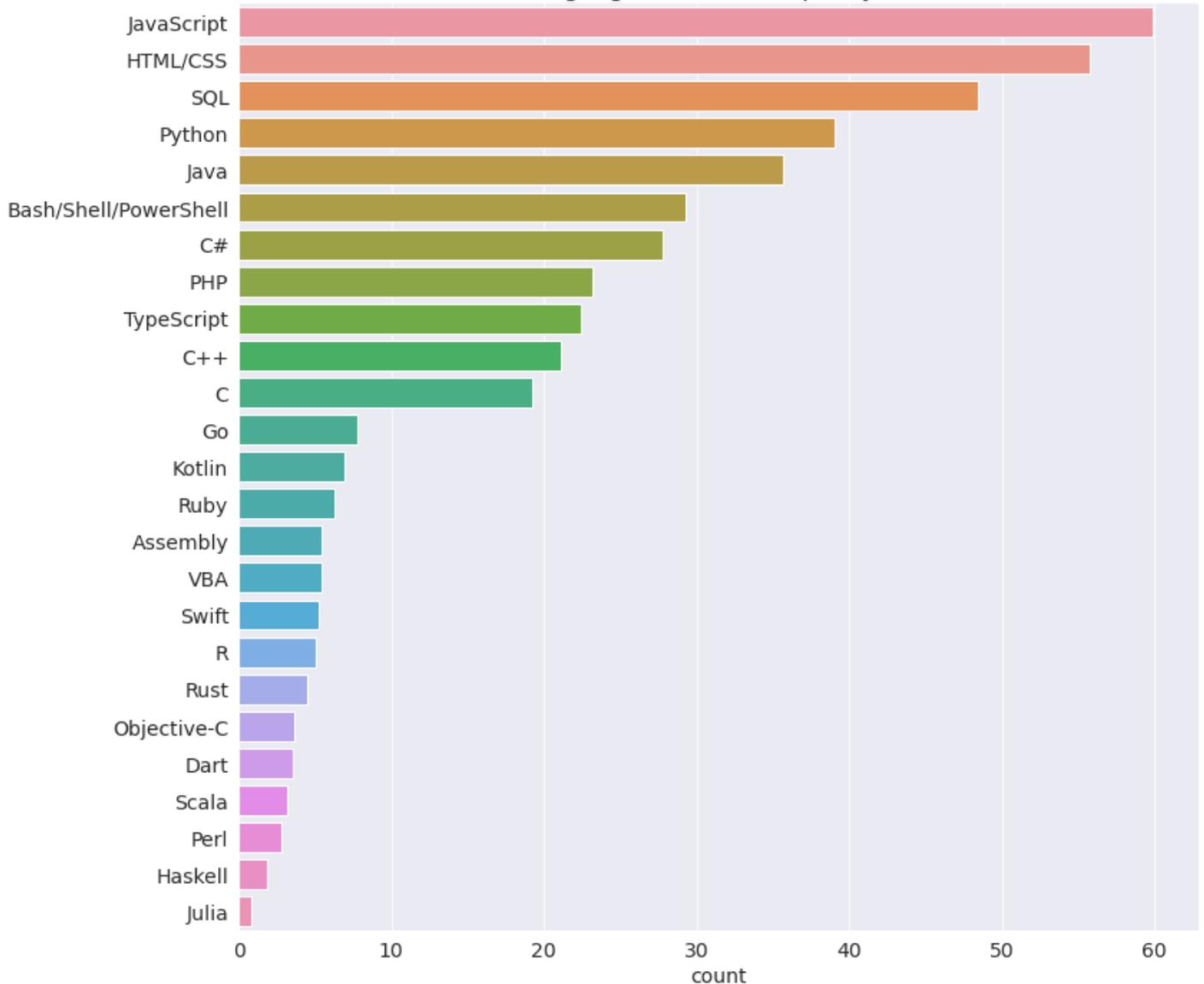
```
plt.figure(figsize=(12, 12))
```

```
sns.barplot(x=languages_worked_percentages, y=languages_worked_percentages.index)
```

```
plt.title("Languages used in the past year");
```

```
plt.xlabel('count');
```

Languages used in the past year



Perhaps unsurprisingly, Javascript & HTML/CSS comes out at the top as web development is one of today's most sought skills. It also happens to be one of the easiest to get started. SQL is necessary for working with relational databases, so it's no surprise that most programmers work with SQL regularly. Python seems to be the popular choice for other forms of development, beating out Java, which was the industry standard for server & application development for over two decades.

Exercises:

- What are the most common languages used by students? How does the list compare with the most common languages used by professional developers?
- What are the most common languages among respondents who do not describe themselves as "Developer, front-end"?
- What are the most common languages among respondents who work in fields related to data science?
- What are the most common languages used by developers older than 35 years of age?
- What are the most common languages used by developers in your home country?

My Work on the Exercises Above:

```
survey_df['Employment'].value_counts()
```

```
Employed full-time          44958
Student                    7734
Independent contractor, freelancer, or self-employed  5619
Not employed, but looking for work  2324
Employed part-time         2200
Not employed, and not looking for work  318
Retired                    241
Name: Employment, dtype: int64
```

```
survey_students_df = survey_df[survey_df['Employment'] == "Student"]
```

```
student_languages_df = split_multicolumn(survey_students_df.LanguageWorkedWith)
```

```
student_languages_percentages = student_languages_df.mean().sort_values(ascending=False)
student_languages_percentages
```

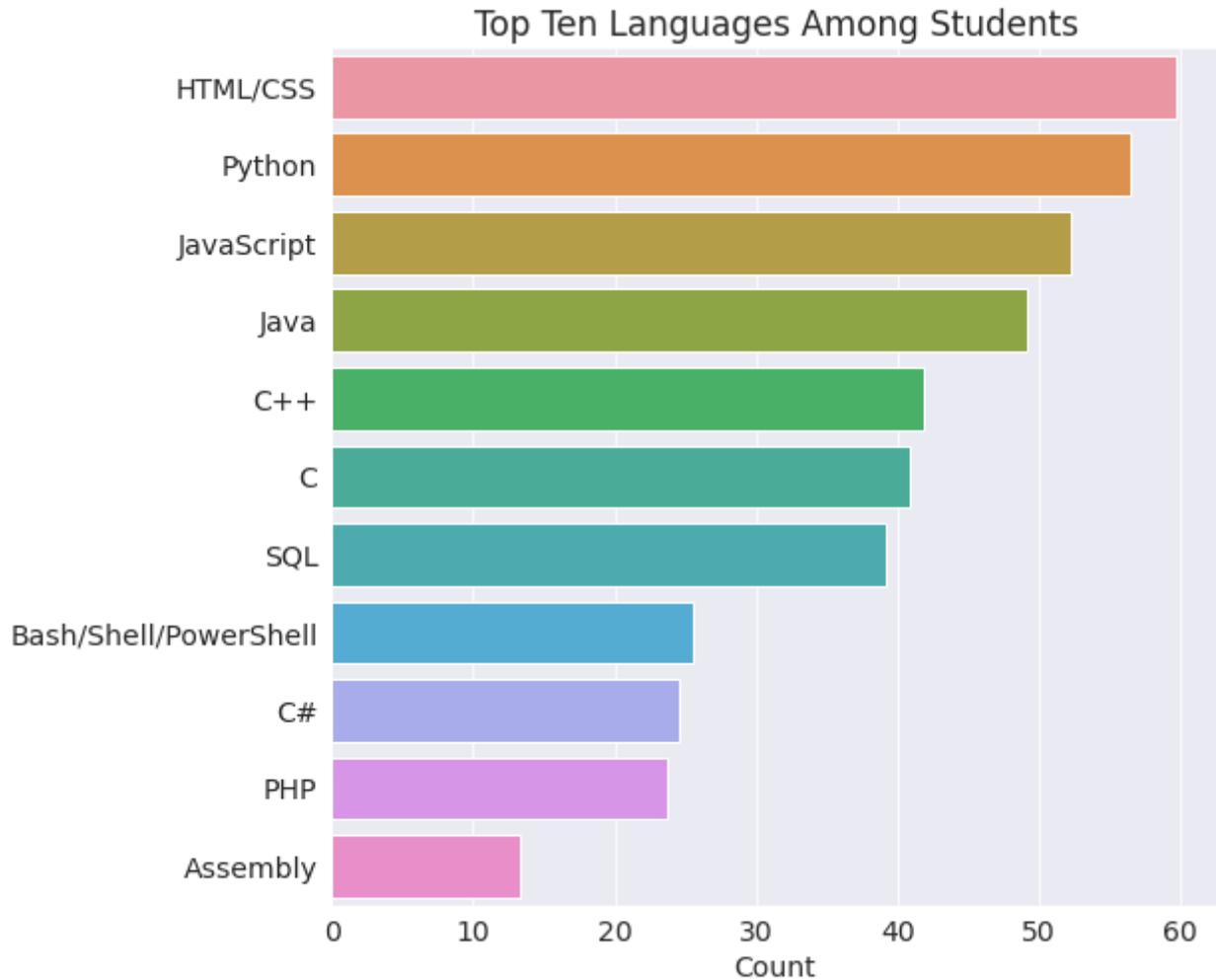
```
HTML/CSS          59.749160
Python            56.490820
JavaScript        52.275666
Java              49.107836
C++               41.802431
C                 40.884407
SQL               39.216447
Bash/Shell/PowerShell  25.536592
C#                24.605637
PHP               23.674683
Assembly         13.317817
TypeScript       11.546418
R                 7.408844
Rust              7.357124
Kotlin           7.008017
Go                5.986553
Dart              5.391777
Swift            4.641841
Haskell          4.085855
Ruby              3.904836
VBA               3.878976
Scala             2.068787
Objective-C       2.042927
Perl              1.409361
Julia             1.318852
dtype: float64
```

```
# I DID IT!!!
```

```
plt.figure(figsize=(8,8))
```

```
sns.barplot(x=student_languages_percentages[0:11], y=student_languages_percentages.index)
plt.title("Top Ten Languages Among Students")
plt.xlabel('Count')
```

```
Text(0.5, 0, 'Count')
```



```
us_survey_df = survey_df[survey_df['Country'] == "United States"]
```

```
us_survey_languages_df = split_multicolumn(us_survey_df.LanguageWorkedWith)
```

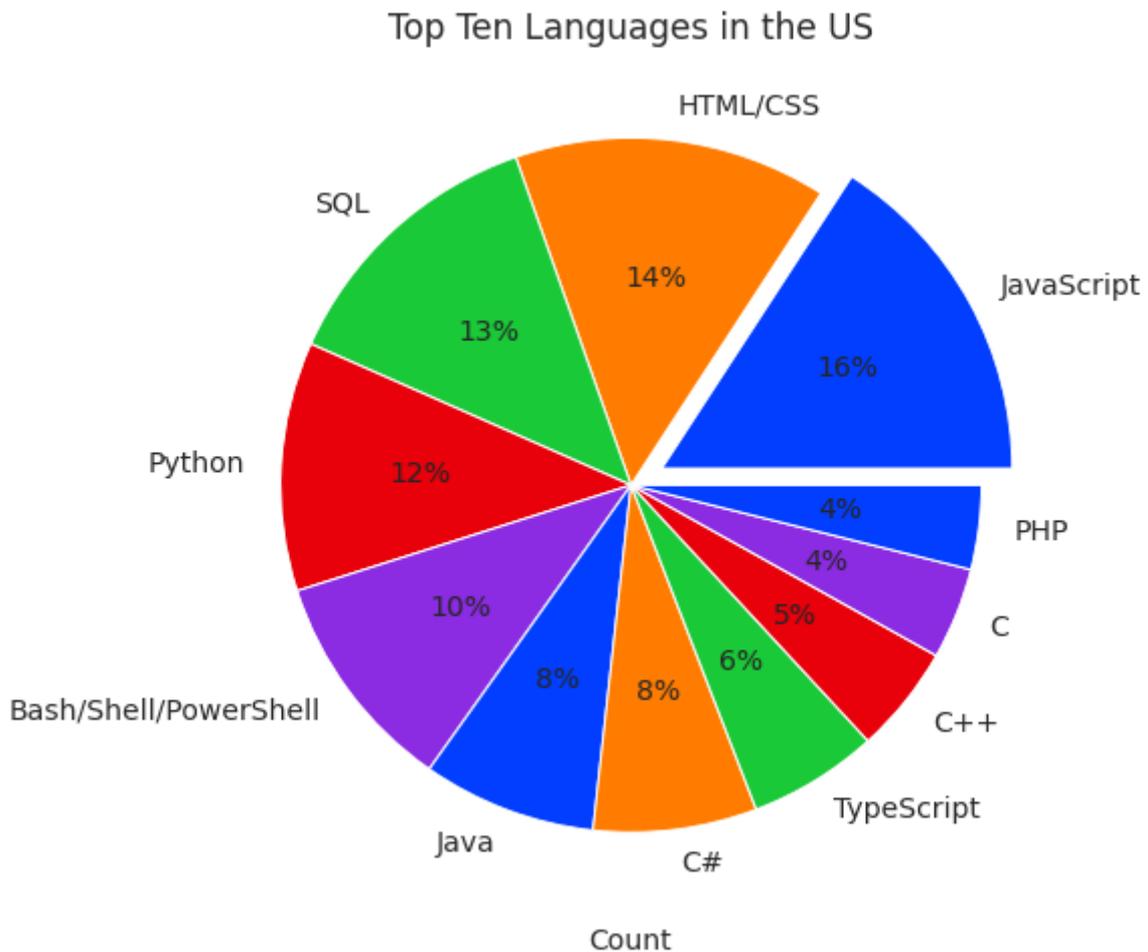
```
us_languages_percentages = us_survey_languages_df.mean().sort_values(ascending=False) *  
us_languages_percentages
```

JavaScript	65.386792
HTML/CSS	59.502061
SQL	53.528413
Python	47.716434
Bash/Shell/PowerShell	42.446043
Java	33.053108
C#	31.088837
TypeScript	24.719101
C++	20.620807
C	17.532940
PHP	16.110258
Go	10.985369

Ruby	9.869857
Rust	6.967909
R	6.668822
Swift	6.628405
VBA	5.933231
Assembly	5.852397
Kotlin	5.642228
Perl	4.227629
Objective-C	4.082128
Scala	4.041710
Dart	2.384609
Haskell	2.328025
Julia	1.155929

dtype: float64

```
plt.figure(figsize=(8,8))
#define Seaborn color palette to use
colors = sns.color_palette('bright')[0:5]
# I chose to make the highest percentage language pop out
explode = (0.1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
plt.pie(us_languages_percentages[0:11], explode = explode, labels = us_languages_perce
plt.title("Top Ten Languages in the US")
plt.xlabel('Count');
```



Q: Which languages are the most people interested to learn over the next year?

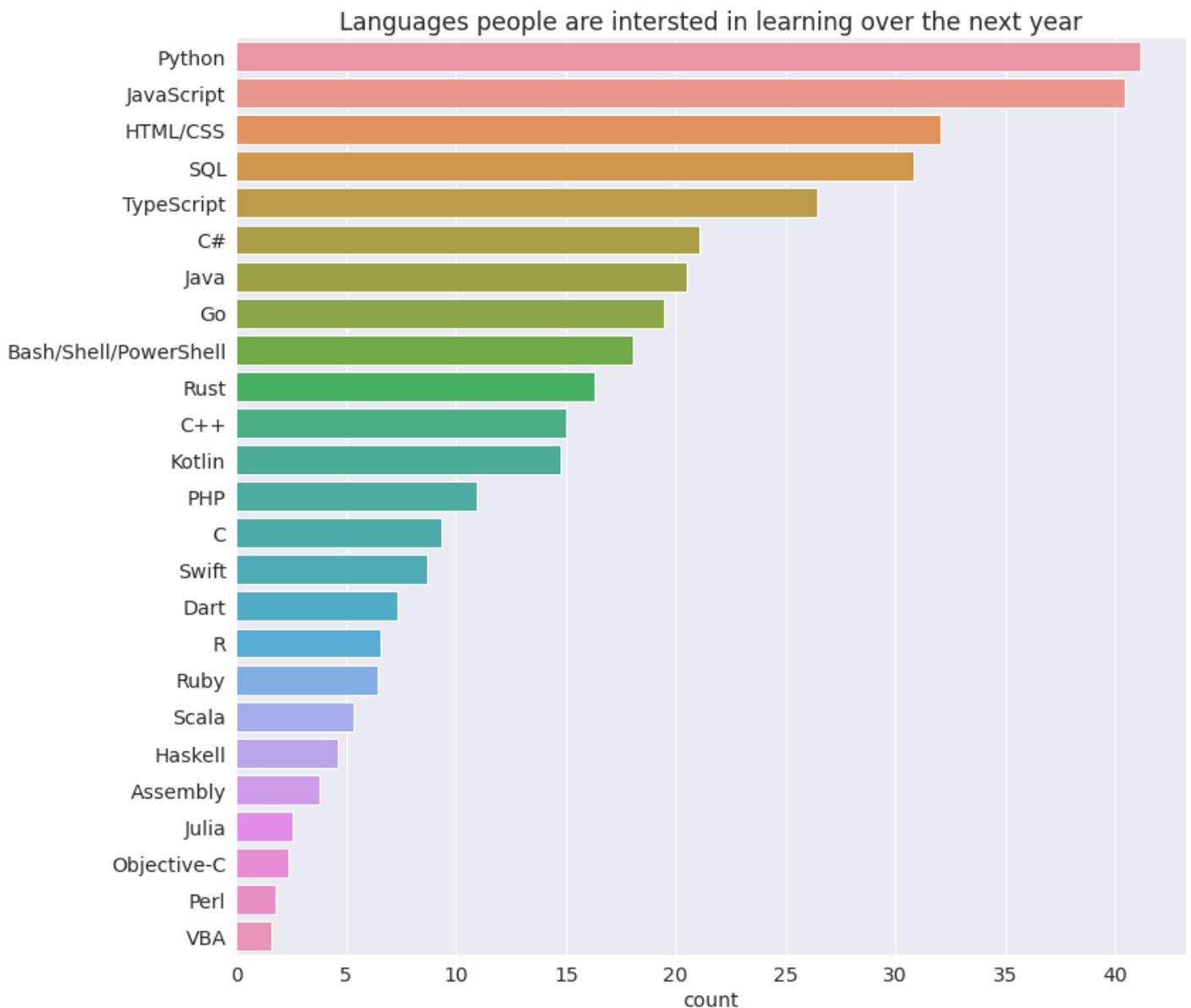
For this, we can use the `LanguageDesireNextYear` column, with similar processing as the previous one.

```
languages_interested_df = split_multicolumn(survey_df.LanguageDesireNextYear)
languages_interested_percentages = languages_interested_df.mean().sort_values(ascending
languages_interested_percentages
```

Python	41.143906
JavaScript	40.425466
HTML/CSS	32.028116
SQL	30.799614
TypeScript	26.451653
C#	21.058688
Java	20.464653
Go	19.432090
Bash/Shell/PowerShell	18.057413
Rust	16.270643
C++	15.014151
Kotlin	14.760676
PHP	10.947657
C	9.359935
Swift	8.692812
Dart	7.308805
R	6.571704
Ruby	6.425528
Scala	5.326097
Haskell	4.593662
Assembly	3.766367
Julia	2.540976
Objective-C	2.338818
Perl	1.761888
VBA	1.611047

dtype: float64

```
plt.figure(figsize=(12, 12))
sns.barplot(x=languages_interested_percentages, y=languages_interested_percentages.index)
plt.title("Languages people are intersted in learning over the next year");
plt.xlabel('count');
```



Once again, it's not surprising that Python is the language most people are interested in learning - since it is an easy-to-learn general-purpose programming language well suited for a variety of domains: application development, numerical computing, data analysis, machine learning, big data, cloud automation, web scraping, scripting, etc. We're using Python for this very analysis, so we're in good company!

Exercises: Repeat the exercises from the previous question, replacing "most common languages" with "languages people are interested in learning/using."

Q: Which are the most loved languages, i.e., a high percentage of people who have used the language want to continue learning & using it over the next year?

While this question may seem tricky at first, it's straightforward to solve using Pandas array operations. Here's what we can do:

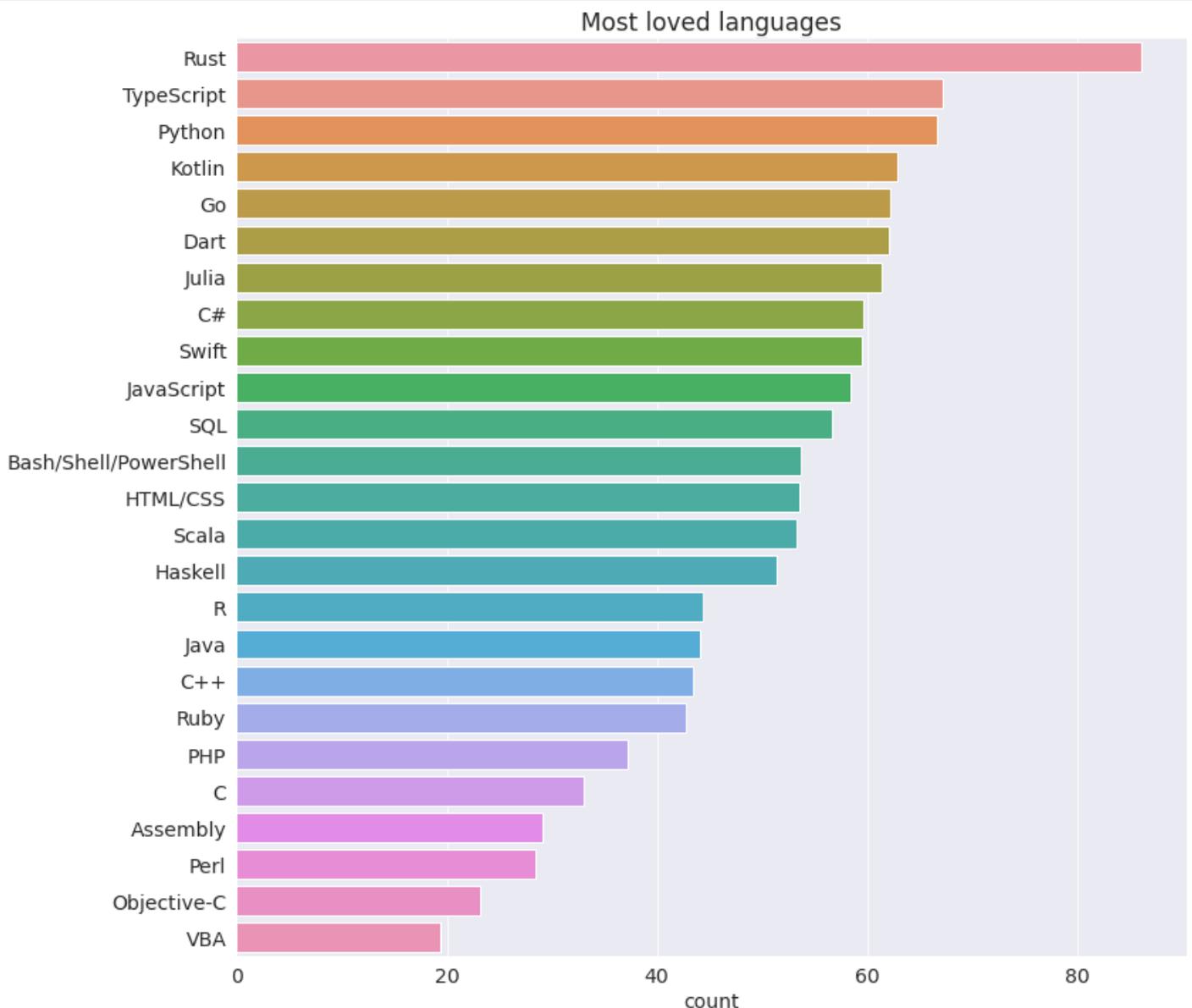
- Create a new data frame `languages_loved_df` that contains a True value for a language only if the corresponding values in `languages_worked_df` and `languages_interested_df` are both True
- Take the column-wise sum of `languages_loved_df` and divide it by the column-wise sum of `languages_worked_df` to get the percentage of respondents who "love" the language
- Sort the results in decreasing order and plot a horizontal bar graph

```
languages_loved_df = languages_worked_df & languages_interested_df
```

```
not_loved_percentages = (languages_loved_df.sum() * 100 / languages_worked_df.sum()).sort
```

```
languages_loved_percentages = (languages_loved_df.sum() * 100 / languages_worked_df.sum())
```

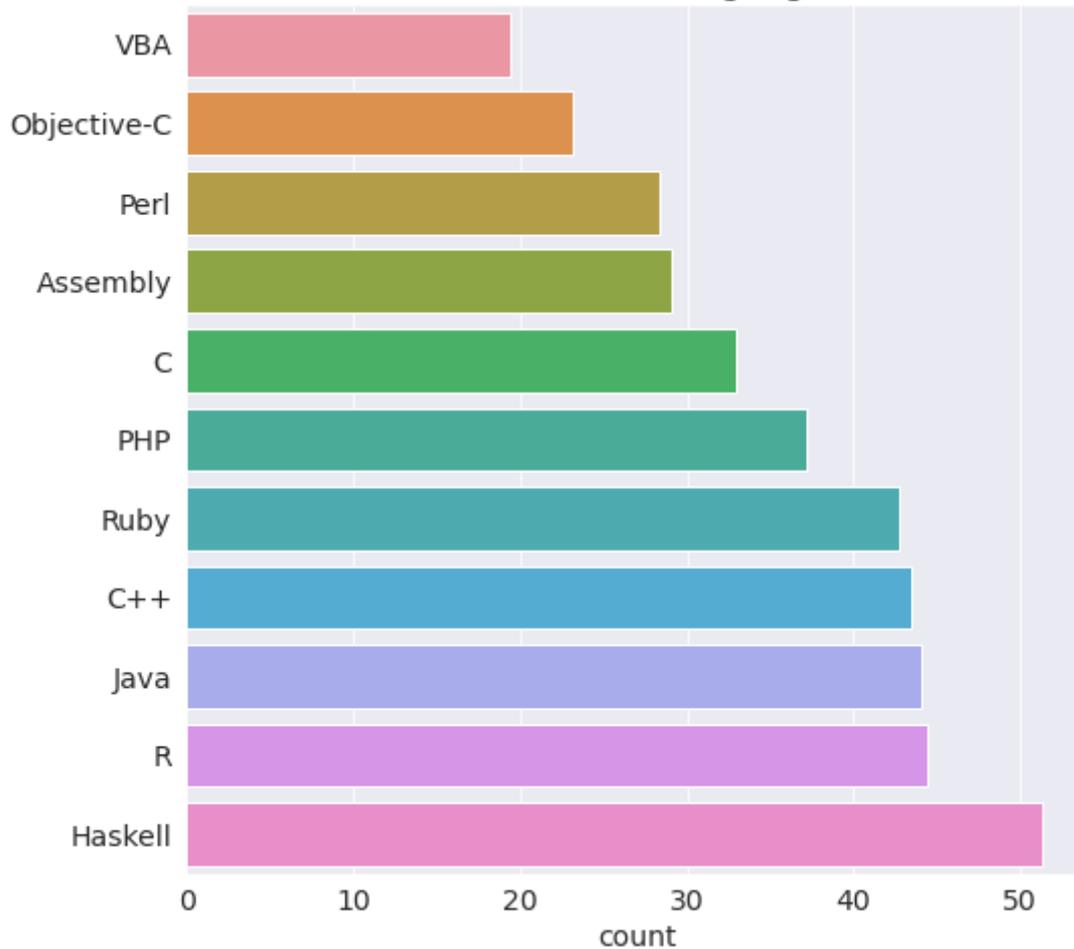
```
plt.figure(figsize=(12, 12))  
sns.barplot(x=languages_loved_percentages, y=languages_loved_percentages.index)  
plt.title("Most loved languages");  
plt.xlabel('count');
```



Exercise = do the same for languages considered to be NOT loved

```
plt.figure(figsize=(8, 8))  
sns.barplot(x=not_loved_percentages[0:11], y=not_loved_percentages.index[0:11])  
plt.title("Most Unloved languages");  
plt.xlabel('count');
```

Most Unloved languages



```
women_survey_df = survey_df[survey_df["Gender"] == "Woman"]
```

```
women_languages = split_multicolumn(women_survey_df.LanguageWorkedWith)
```

```
women_languages_percentages = women_languages.mean().sort_values(ascending=False) * 100
```

```
women_languages_percentages
```

HTML/CSS	66.049544
JavaScript	65.867014
SQL	54.732725
Python	41.147327
Java	37.627119
Bash/Shell/PowerShell	27.275098
C#	25.189048
PHP	23.494133
TypeScript	21.069100
C++	19.061278
C	16.531943
R	9.335072
Ruby	8.448501
Kotlin	5.788787
Swift	5.684485

```

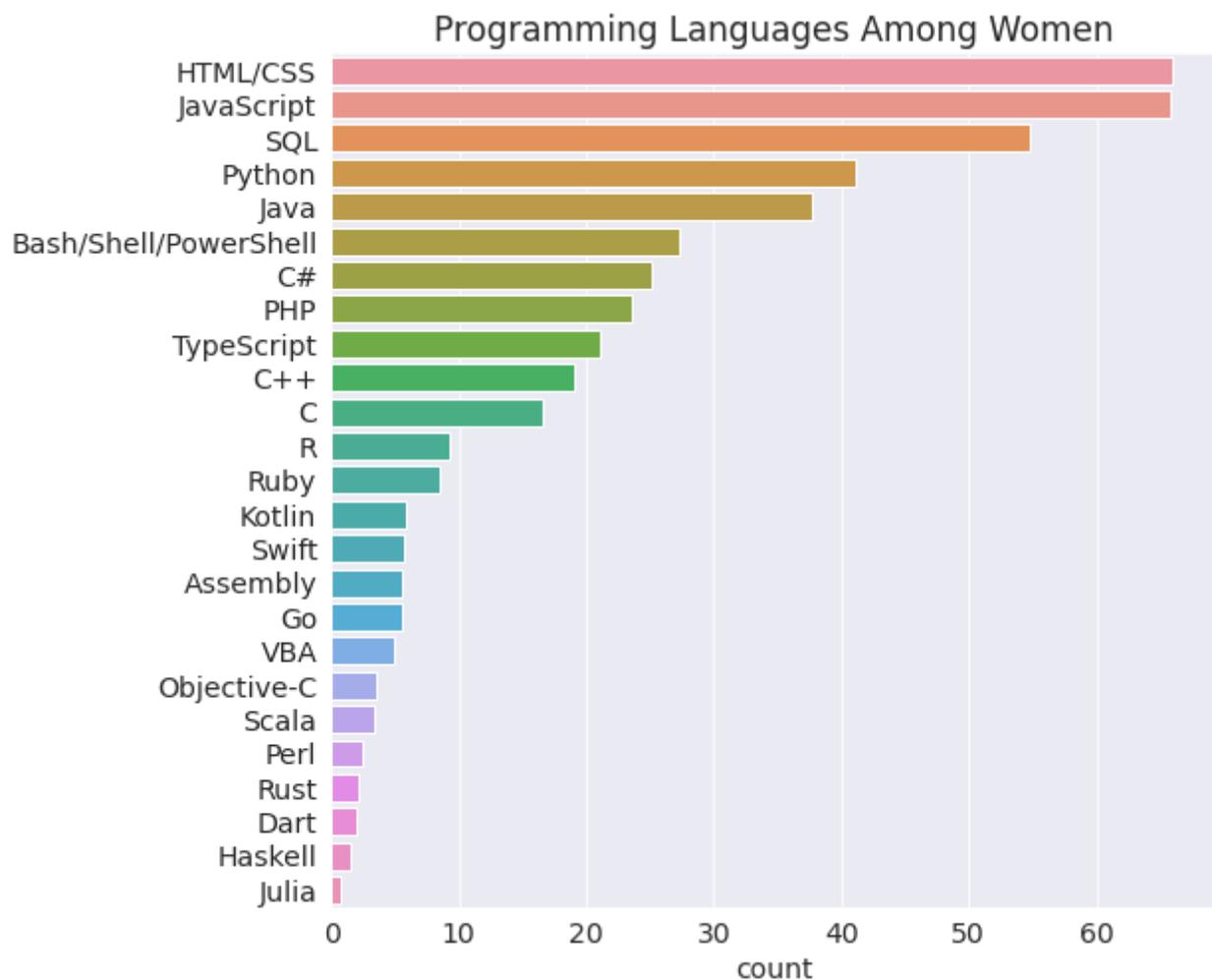
Assembly          5.606258
Go                5.554107
VBA              4.954368
Objective-C      3.546284
Scala            3.441982
Perl             2.372881
Rust             2.059974
Dart             2.007823
Haskell          1.564537
Julia            0.756193
dtype: float64

```

```

plt.figure(figsize=(8, 8))
sns.barplot(x=women_languages_percentages, y=women_languages_percentages.index)
plt.title("Programming Languages Among Women");
plt.xlabel('count');

```



```
women_us_languages = women_languages & us_survey_languages_df
```

```
women_us_languages_percentages = women_us_languages.mean().sort_values(ascending = False)
```

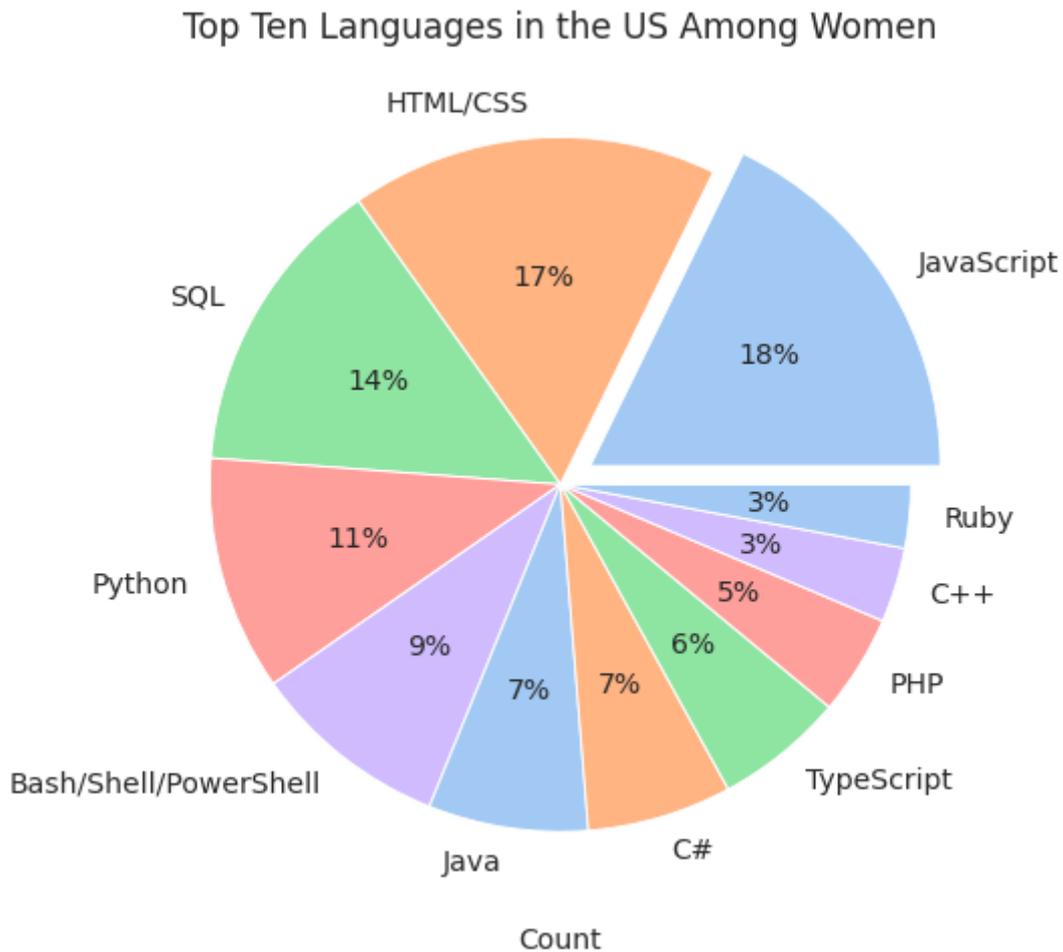
```
women_us_languages_percentages[0:11]
```

```
JavaScript          5.112434
```

HTML/CSS	4.874339
SQL	4.021164
Python	3.141534
Bash/Shell/PowerShell	2.632275
Java	2.116402
C#	1.917989
TypeScript	1.712963
PHP	1.335979
C++	0.998677
Ruby	0.846561

dtype: float64

```
plt.figure(figsize=(8,8))
#define Seaborn color palette to use
colors = sns.color_palette('pastel')[0:5]
# I chose to make the highest percentage language pop out
explode = (0.1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
plt.pie(women_us_languages_percentages[0:11], explode = explode, labels = women_us_lang
plt.title("Top Ten Languages in the US Among Women")
plt.xlabel('Count');
```



[Rust](#) has been StackOverflow's most-loved language for [four years in a row](#). The second most-loved language is TypeScript, a popular alternative to JavaScript for web development.

Python features at number 3, despite already being one of the most widely-used languages in the world. Python has a solid foundation, is easy to learn & use, has a large ecosystem of domain-specific libraries, and a massive

worldwide community.

Exercises: What are the most dreaded languages, i.e., languages which people have used in the past year but do not want to learn/use over the next year. Hint: `~languages_interested_df`.

Q: In which countries do developers work the highest number of hours per week? Consider countries with more than 250 responses only.

To answer this question, we'll need to use the `groupby` data frame method to aggregate the rows for each country. We'll also need to filter the results to only include the countries with more than 250 respondents.

```
# Pull out the info by country, work week hours, and age, and group by each one  
# consecutively  
survey_df.groupby('Country')[['WorkWeekHrs', 'Age']].mean()
```

	WorkWeekHrs	Age
Country		
Afghanistan	46.500000	23.571429
Albania	43.962963	26.766667
Algeria	36.862069	28.019608
Andorra	42.000000	39.000000
Angola	24.500000	25.500000
...
Venezuela, Bolivarian Republic of...	40.125000	29.090909
Viet Nam	41.391667	25.786585
Yemen	40.000000	31.000000
Zambia	38.000000	29.000000
Zimbabwe	39.928571	25.578947

183 rows × 2 columns

```
countries_df = survey_df.groupby('Country')[['WorkWeekHrs']].mean().sort_values('WorkWeekHrs')
```

```
countries_df.sort_values('WorkWeekHrs', ascending=False)
```

	WorkWeekHrs
Country	
Kuwait	58.222222
Iraq	52.153846
Grenada	50.000000
Maldives	47.300000
Afghanistan	46.500000
...	...
North Korea	NaN

	WorkWeekHrs
Country	
Saint Lucia	NaN
Sierra Leone	NaN
Solomon Islands	NaN
Timor-Leste	NaN

183 rows × 1 columns

```
# Taking only countries with more than 250 responses
high_response_countries_df = countries_df.loc[survey_df.Country.value_counts() > 250].h
```

```
high_response_countries_df.sort_values('WorkWeekHrs', ascending=False)
```

	WorkWeekHrs
Country	
Iran	44.337748
Israel	43.915094
China	42.150000
United States	41.802982
Greece	41.402724
Viet Nam	41.391667
South Africa	41.023460
Turkey	40.982143
Sri Lanka	40.612245
New Zealand	40.457551
Belgium	40.444444
Canada	40.208837
Hungary	40.194340
Bangladesh	40.097458
India	40.090603

```
survey_df.sample(10)
```

	Country	Age	Gender	EdLevel	UndergradMajor	Hobbyist	Age1stCode	YearsCode	YearsCo
21320	Republic of Moldova	23.0	Man	Bachelor's degree (B.A., B.S., B.Eng., etc.)	Another engineering discipline (such as civil,...	Yes	15.0	2.0	
22062	Italy	NaN	NaN	Bachelor's degree (B.A., B.S., B.Eng., etc.)	Another engineering discipline (such as civil,...	Yes	16.0	30.0	

	Country	Age	Gender	EdLevel	UndergradMajor	Hobbyist	Age1stCode	YearsCode	YearsCoc
55920	United Kingdom	25.0	Man	Bachelor's degree (B.A., B.S., B.Eng., etc.)	Another engineering discipline (such as civil,...	Yes	19.0	6.0	
41003	Spain	NaN	NaN	Master's degree (M.A., M.S., M.Eng., MBA, etc.)	Computer science, computer engineering, or sof...	Yes	13.0	21.0	
47035	Canada	24.0	Man	Associate degree (A.A., A.S., etc.)	Computer science, computer engineering, or sof...	Yes	6.0	19.0	
36344	India	NaN	NaN	Bachelor's degree (B.A., B.S., B.Eng., etc.)	Computer science, computer engineering, or sof...	Yes	22.0	12.0	
48957	United States	NaN	NaN	NaN	NaN	Yes	NaN	NaN	
17152	United States	21.0	Non-binary, genderqueer, or gender non-conforming	Some college/university study without earning ...	I never declared a major	Yes	11.0	10.0	
23905	Norway	29.0	Man	Bachelor's degree (B.A., B.S., B.Eng., etc.)	Computer science, computer engineering, or sof...	Yes	14.0	11.0	
42936	United Kingdom	NaN	NaN	NaN	NaN	Yes	NaN	NaN	

10 rows × 21 columns

The Asian countries like Iran, China, and Israel have the highest working hours, followed by the United States. However, there isn't too much variation overall, and the average working hours seem to be around 40 hours per week.

Exercises:

- How do the average work hours compare across continents? You may find this list of [countries in each continent](#) useful.
- Which role has the highest average number of hours worked per week? Which one has the lowest?
- How do the hours worked compare between freelancers and developers working full-time?

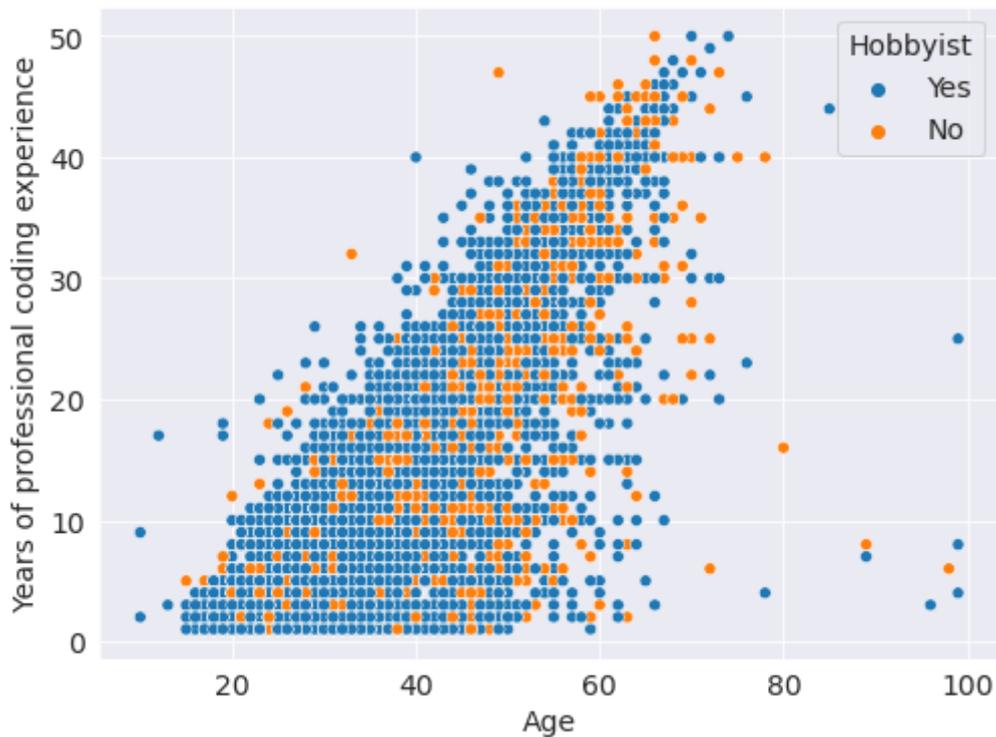
Q: How important is it to start young to build a career in programming?

Let's create a scatter plot of Age vs. YearsCodePro (i.e., years of coding experience) to answer this question.

```
schema.YearsCodePro
```

'NOT including education, how many years have you coded professionally (as a part of your work)?'

```
plt.figure(figsize=(8,6))
sns.scatterplot(x='Age', y='YearsCodePro', hue='Hobbyist', data=survey_df)
plt.xlabel("Age")
plt.ylabel("Years of professional coding experience");
```



```
women_survey_df.sample(5)
```

	Country	Age	Gender	EdLevel	UndergradMajor	Hobbyist	Age1stCode	YearsCode	YearsCodePro	
28935	United Kingdom	37.0	Woman	Bachelor's degree (B.A., B.S., B.Eng., etc.)	Fine arts or performing arts (such as graphic ...	Yes	17.0	20.0	13.0	
52129	Germany	21.0	Woman	Secondary school (e.g. American high school, G...	NaN	Yes	18.0	3.0	NaN	
55113	Netherlands	25.0	Woman	Master's degree (M.A., M.S., M.Eng., MBA, etc.)	Computer science, computer engineering, or sof...	Yes	18.0	5.0	9.0	H

	Country	Age	Gender	EdLevel	UndergradMajor	Hobbyist	Age1stCode	YearsCode	YearsCodePro
48560	United Kingdom	23.0	Woman	Bachelor's degree (B.A., B.S., B.Eng., etc.)	Computer science, computer engineering, or sof...	Yes	18.0	5.0	2.0
617	United Kingdom	37.0	Woman	Master's degree (M.A., M.S., M.Eng., MBA, etc.)	Computer science, computer engineering, or sof...	Yes	14.0	23.0	12.0

5 rows × 21 columns

```
women_countries_pro = women_survey_df[women_survey_df['YearsCodePro'] != 'NaN']
continents = ['North America', 'South America', 'Europe', 'Africa', 'Asia', 'Australia']
women_countries_pro
```

```
continent_list = 'https://hub.jovian.ml/wp-content/uploads/2020/09/countries.csv'
```

```
od.download(continent_list)
```

```
continents = pd.read_csv('')
```

```
women_countries_pro["Continent"] =
```

```
plt.figure(figsize=(10,10))
sns.scatterplot(x='Age', y='YearsCodePro', hue='Country', data=women_countries_pro)
plt.xlabel("Age")
plt.ylabel("Countries");
```



- Kazakhstan
- Finland
- Hong Kong (S.A.R.)
- Pakistan
- Russian Federation
- Viet Nam
- Iran
- China
- Tunisia
- Sri Lanka
- Uzbekistan
- Slovenia
- United Arab Emirates
- Armenia
- Sudan
- Saudi Arabia
- Kenya
- Qatar
- Greece
- Myanmar
- Somalia
- Namibia
- Bulgaria
- Estonia
- Lithuania
- Latvia
- Azerbaijan
- Nigeria
- Morocco
- Malta
- Argentina
- Bhutan
- Hungary
- Czech Republic
- Swaziland
- Kosovo
- Thailand
- Georgia
- Paraguay
- Iceland
- Peru
- Panama
- Taiwan
- Mauritius
- Côte d'Ivoire
- Jordan
- Albania
- Madagascar
- Belarus
- Bahamas

- Afghanistan
- Uruguay
- Nicaragua
- Rwanda
- United Republic of Tanzania
- Venezuela, Bolivarian Republic of...
- Lebanon
- Cyprus
- Cameroon
- Oman
- Angola
- The former Yugoslav Republic of Macedonia
- Senegal
- Guyana
- Jamaica
- Dominican Republic
- Trinidad and Tobago
- Bahrain
- Ghana
- Nomadic
- Montenegro



You can see points all over the graph, which indicates that you can **start programming professionally at any age**. Many people who have been coding for several decades professionally also seem to enjoy it as a hobby.

We can also view the distribution of the `Age1stCode` column to see when the respondents tried programming for the first time.

```
plt.title(schema.Age1stCode)
sns.histplot(x=survey_df.Age1stCode, bins=30, kde=True);
```

As you might expect, most people seem to have had some exposure to programming before the age of 40. However, but there are people of all ages and walks of life learning to code.

Exercises:

- How does programming experience change opinions & preferences? Repeat the entire analysis while comparing the responses of people who have more than ten years of professional programming experience vs. those who don't. Do you see any interesting trends?
- Compare the years of professional coding experience across different genders.



Hopefully, you are already thinking of many more questions you'd like to answer using this data. Use the empty cells below to ask and answer more questions.

Let's save and commit our work before continuing

```
import jovian
```

```
jovian.commit()
```

Inferences and Conclusions

We've drawn many inferences from the survey. Here's a summary of a few of them:

- Based on the survey respondents' demographics, we can infer that the survey is somewhat representative of the overall programming community. However, it has fewer responses from programmers in non-English-speaking countries and women & non-binary genders.
- The programming community is not as diverse as it can be. Although things are improving, we should make more efforts to support & encourage underrepresented communities, whether in terms of age, country, race, gender, or otherwise.
- Although most programmers hold a college degree, a reasonably large percentage did not have computer science as their college major. Hence, a computer science degree isn't compulsory for learning to code or building a career in programming.
- A significant percentage of programmers either work part-time or as freelancers, which can be a great way to break into the field, especially when you're just getting started.
- Javascript & HTML/CSS are the most used programming languages in 2020, closely followed by SQL & Python.
- Python is the language most people are interested in learning - since it is an easy-to-learn general-purpose programming language well suited for various domains.
- Rust and TypeScript are the most "loved" languages in 2020, both of which have small but fast-growing communities. Python is a close third, despite already being a widely used language.
- Programmers worldwide seem to be working for around 40 hours a week on average, with slight variations by country.
- You can learn and start programming professionally at any age. You're likely to have a long and fulfilling career if you also enjoy programming as a hobby.

Exercises

There's a wealth of information to be discovered using the survey, and we've barely scratched the surface. Here are some ideas for further exploration:

- Repeat the analysis for different age groups & genders, and compare the results

- Pick a different set of columns (we chose 20 out of 65) to analyze other facets of the data
- Prepare an analysis focusing on diversity - and identify areas where underrepresented communities are at par with the majority (e.g., education) and where they aren't (e.g., salaries)
- Compare the results of this year's survey with the previous years and identify interesting trends

References and Future Work

Check out the following resources to learn more about the dataset and tools used in this notebook:

- Stack Overflow Developer Survey: <https://insights.stackoverflow.com/survey>
- Pandas user guide: https://pandas.pydata.org/docs/user_guide/index.html
- Matplotlib user guide: <https://matplotlib.org/3.3.1/users/index.html>
- Seaborn user guide & tutorial: <https://seaborn.pydata.org/tutorial.html>
- opendatasets Python library: <https://github.com/JovianML/opendatasets>

As a next step, you can try out a project on another dataset of your choice:

<https://jovian.ml/aakashns/zerotopandas-course-project-starter> .

```
import jovian
```

```
jovian.commit()
```