

PythonCrashCourse_Notebook

March 8, 2024

Crash Course

In this notebook, we'll be learning the fundamentals of Python (all of which apply to programming in general!). We'll be working through a dataset of the most streamed Spotify songs compiled in 2023 .

1 Learning Objectives

1.1 At the end of today, you'll be able to:

- Understand the basic components of a Jupyter notebook
- Write and debug basic Python code
- Use functions, loops, conditional statements, and apply programming fundamentals applicable across computing languages
- Analyze a dataset using Python libraries built for data science
- Learn and discuss best practices for data visualization to communicate your results

IMPORTANT: This notebook is READ-ONLY. To edit and run this notebook, follow the steps in the next section.

1.2 Before you start - Save this notebook!

When you open a new Colab notebook from the WebCampus (like you hopefully did for this one), you cannot save changes. So it's best to store the Colab notebook in your personal drive "**File > Save a copy in drive...**" **before** you do anything else.

The file will open in a new tab in your web browser, and it is automatically named something like: "**CopyofPythonCrashCourse_Notebook.ipynb**". You can rename this to just the title of the assignment "**PythonCrashCourse_Notebook.ipynb**". Make sure you do keep an informative name (like the name of the assignment) to help you be able to come back to this after you complete this part of the assignment.

Where does the notebook get saved in Google Drive?

By default, the notebook will be copied to a folder called "Colab Notebooks" at the root (home) directory of your Google Drive.

2 Pre-Workshop Survey on Coding Background & Attitudes

Survey [link](#)

Please complete this short survey prior to the start of the session. All responses are anonymous.

3 Section 0: Python & Colab

Welcome to Google Colab! We'll be working in Colab for this introductory Python course.

3.1 What is Colab?

Colab is a browser-based environment - you can develop, run, and share code directly on your browser without having to install any software or libraries on your computer. Additionally, code written and run from colab runs on google's cloud computers, so you aren't limited by your laptop's speed or compute power.

Note: One downside of Colab is that if you turn off your computer, are idle, or refresh the page, your current runtime is lost. You will need to run the notebook from the beginning in order to redo any variables, functions, or operations you have performed. You can more easily rerun your notebook by clicking Runtime»Run all, or Runtime»Run before at the top of the page. This will attempt to run all of your current cells (until one gets an error) or all of the cells above the current cell your cursor is on.

Colab can be especially helpful for data science tasks, because it gives you a space to write code, generate/view graphs, and organize your thought process. For those of you who have used jupyter before, colab acts a simple browser-based jupyter notebook. We'll be using colab with Python, but colab can be used with a variety of different coding languages.

A Colab notebook (a file like this one) consists of cells, snippets of code or text that you can create, edit, and move around to keep things organized. These cells can be:

- **text cells** or
- **code cells**

You can **double click** on a cell in order to edit it.

3.2 Text Cells

This is a **text cell**. You can **double-click** to edit this cell. Text cells use markdown syntax. To learn more, see the [markdown guide](#).

You can format text cells by **changing the font**, adding: 1. Numbered lists, or * Bullet points

You can even add HTML to your text cells to embed images or add other fun effects.

You can also add math to text cells using [LaTeX](#) to be rendered by [MathJax](#). Just place the statement within a pair of \$ signs. For example $\sqrt{3x-1} + (1+x)^2$ becomes $\sqrt{3x-1} + (1+x)^2$

$x)^2$. To insert a text cell, click on a cell on the page and then click the “+ Text” button at the top. Or scroll to the middle of the notebook page before/after a cell and you should see these options appear:

This will make a text cell directly above or below the cell you are on.

3.3 0.0 Exercise

Make a text cell below this one.

In your text cell:

1. Write a one sentence description about yourself, and why you chose to take this workshop.
2. Write a list of 3 small tasks you would like to be able to accomplish with Python/colab after today.
3. Using the HTML syntax ``, embed the image hosted at:

https://imgs.xkcd.com/comics/future_self.png

3.4 Code Cells

Colab notebooks also consist of **coding cells**, cells that have executable code written in them.

To run a coding cell, either: 1. hover over the cell and click the “play” arrow that appears on the left upper corner 2. Or click into the cell and then press Ctrl+Enter (or Shift+Return).

You should see a check mark if the cell runs successfully .

Coding cells can look like the following, where you can see raw code and comments (anything after the `#` hashtag).

Task: Run this cell!

```
[ ]: n_sections = 10
# This is a comment! When there is a # before a line in a coding cell,
# python will ignore that line of code, rather than trying to run it.
n_exercises_per_section = 3

'''
You can also
write longer
comments
by using
this syntax.
'''

print("Number of exercises in this notebook:",
      n_sections*n_exercises_per_section)
```

Cells can also look like the following where you can hide code by adding the following line to the top of your page.

```
# @title TITLE GOES HERE.
```

You can click on the arrow > at the top left to show the code or on **Show code**. You can still run hidden code cells by pushing “play” , or clicking into the cell and typing Ctrl+Enter. To hide the code once more, double click on the text part of the cell.

```
[ ]: #@title Hidden Code Example

n_sections = 10
n_exercises_per_section = 3
print("Number of exercises in this notebook:",
      ↪n_sections*n_exercises_per_section)
```

3.5 0.1 Coding Exercise

Fill in the following code cell and run it.

```
[ ]: breakfast_yesterday = "coffee" ### REPLACE WITH YOUR CODE HERE ###
lunch_yesterday = "chicken salad" ### REPLACE WITH YOUR CODE HERE ###
print("I had ", breakfast_yesterday, " for breakfast",
      ↪"and ", lunch_yesterday, " for lunch.")
```

3.6 0.2 Coding Exercise

Delete the text cell below, and replace it with a code cell that prints **every programmer’s first words**:

“Hello, world!”

DELETE ME!

4 Section 1: Setup and import files


There are several steps we need to do to access the dataset before we can load the file: 1. Copy the data file to working memory to run in this notebook 2. Check that we have it available to us

Other options (backup if the above doesn’t work): 1. Mount our gdrive (connect to this notebook) 2. Fetch the data file from my github repository 3. Change into our directory where the file is located

We will start by downloading the data file (**spotify-2023-clean.csv**) to access for today.

Task: Run the cell below. If it works, you should see text that looks like this:

```
[ ]: ### RUN THIS CELL ###
!wget --no-check-certificate 'https://docs.google.com/uc?
  ↪export=download&id=1hv66NjJSa98UUVsbEppZjtOCInmbbRES' -O spotify-2023-clean.
  ↪csv
```

Task: On the left side of the Colab window, click the  icon, and double click on your files to see what the raw data looks like. You should see a file called **spotify-2023-clean.csv**.

^ If this was not successful for you, please let us know and we will help troubleshoot!

4.1 Backup Import of Files From Github- Only needed if above didn't work ^^

If the above worked, please skip these steps. If it didn't work, try this out and let us know!

```
[ ]: #@title Backup Solution - Step 1
# Step 1: Setup and add files needed to access gdrive
from google.colab import drive # these lines
    ↳ mount your gdrive to access the files we import below
drive.mount('/content/gdrive', force_remount=True)
```

```
[ ]: #@title Backup Solution - Step 2:4
# Step 2: Make a folder to store the files
!mkdir -p '/content/gdrive/My Drive/PythonCrashCourseMaterials/'

# Step 3: Change directory to the correct location in gdrive (modified way to
    ↳ do this from before)
import os
os.chdir('/content/gdrive/My Drive/PythonCrashCourseMaterials/')

# Step 4: These lines clone (copy) all the files you will need from where I
    ↳ store the code+data for the course (github)
# Second part of the code copies the files to this location and folder in your
    ↳ own gdrive
!git clone https://github.com/tmckim/python-crash-course/ '/content/gdrive/My
    ↳ Drive/PythonCrashCourseMaterials/'
```

```
[ ]: #@title Backup Solution - Step 5
# Step 5
# Change directory into the folder where the resources for this assignment are
    ↳ stored in gdrive
os.chdir('/content/gdrive/My Drive/PythonCrashCourseMaterials/')
```

4.2 Backup of the Backup - Only needed if BOTH of the above didn't work ^^

If all of the above fails, please follow this [link](#) to download the data file locally to your computer. You can then navigate to the icon on the left and drag and drop the file there for access.

4.3 Python Libraries

Python comes with many useful built in classes and functions - lists, dictionaries, counters, and their associated functions, are some examples. You can accomplish many simple tasks using native Python code.

However, just like in many computer languages, the beauty of Python is that other people have spent much time developing useful code that they have tested robustly and wish to share with others.

These are called libraries or packages. We can import popular packages into the Python/Colab environment and if the package has not already been installed on our version of Python, we can easily install the package and use it.

(In Colab, or in a linux environment if you run Python locally, we can easily install new packages using `!pip install [PYTHON PACKAGE]`)

For this Colab notebook we will be working with four packages that typically come pre-installed on most versions of Python. They are:

- **numpy**: A library for working with numerical lists, called arrays
- **pandas**: A library for working with two dimensional lists, called dataframes
- **matplotlib**: A package for plotting, commonly used in tandem with numpy and pandas
- **seaborn**: A package for plotting, commonly used in tandem with pandas

We can import these libraries (or certain classes/functions from these libraries) into our local runtime using the following code. The nicknames I used (np, plt, pd, and sns) are pretty standard nicknames that most Python programmers use, though you could use anything.

Task: Run the following cell to import the relevant libraries.

```
[ ]: ### RUN THIS CELL ###
# Import our plotting package from matplotlib and shorten it to plt
import matplotlib.pyplot as plt

# Specify that all plots will happen inline & in high resolution
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

# Import pandas for working with databases and shorten it to pd
import pandas as pd

# Import numpy and shorten it to np
import numpy as np

# Import plotting package seaborn and shorten to sns
import seaborn as sns

# Print statement to confirm
print('Packages imported!')
```

Task: Run this cell to set formatting options. These can be changed according to your preferences.

```
[ ]: ### RUN THIS CELL ###
# We want to see all the columns/rows in the dataset
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', None)
pd.options.display.float_format = '{:.1f}'.format # don't use scientific
notation
```

4.4 Importing Data

Read the file from the colab files into your runtime. There are several options for reading a file in, but we will use the **pandas** package (which we imported as **pd**): * **pandas read_table()** or **read_csv()** function, which reads a file with multiple data types into a dataframe.

```
[ ]: ### RUN THIS CELL ###  
## Import file (cleaned up)  
spotify = pd.read_csv('spotify-2023-clean.csv')  
print('data imported!')
```

```
[ ]: ### RUN THIS CELL ###  
## Type the name of the dataframe to view it  
spotify
```

```
[ ]: ### RUN THIS CELL ###  
## Another quick preview of variables  
spotify.info()
```

We have 953 rows (observations) and 24 columns. In the column labeled **Column**, the names of each column are listed and it shows how many non-null values are present, meaning how much data there is. Any difference between this number and the total number of entries (953 shown at the top) indicates there is missing data. The last column **Dtype** tells us what type of data you have in each column. You can see here that many are numbers and have no decimals (**int64**) or have decimal values (**float64**). Note that some of our columns that have strings or text values are right now labeled as **object** (first entry). We will adjust this along the way to make sure they are strings.

We can see that we do have missing data here. Particularly note these 3 columns: **streams**, **in_shazam_charts**, and **key**. If you go back up and review the table, you will see that missing values are labeled with **NaN**, which stands for **Not a Number**. This is typically a common naming convention to use, and you will see it across datasets. We will discuss this more as we go, and ultimately, when working datasets that have NaNs, you will need to make some decisions on what you do with this data.

4.5 Show me the data

I know that you don't know much yet about the data, but let's see if you can make some predictions. Who do you think might be among the top artists as of 2023? Let's see if you were right!

Task: Run the cell below to see the answer. Notice that it only takes 13 lines of code to produce our plot. By the end of today, you will be able to understand what this all means- woot!

And the award for the most songs in our dataset goes to

```
[ ]: ### RUN THIS CELL ###  
  
# set a theme for consistency and format
```

```

sns.set_theme(style="ticks", font_scale=1.25) # set the theme
↳ we want in seaborn (sns) - always start with this- only need to do once

# This takes the data out of the larger dataset that we need
artist_counts = spotify['artist_name'].value_counts()
artist_plot = artist_counts.head(15)

# Set this text for the plot- easiest to edit here as needed
x_label_name = 'Artist Name' # x-axis label
↳ text
y_label_name = 'Number of Songs' # y-axis label
↳ text
plot_title = 'Top 15 Artists With Most Songs' # plot title text

# Example of barplot
ax = sns.barplot(x = artist_plot.index, y = artist_plot, hue = artist_plot,
↳ palette = 'flare') # this is the main plotting code

# Adjustment + aesthetics of plot
ax.figure.set_size_inches(6.5, 4.5) # this sets the
↳ size and we will use to be consistent across all plots below
sns.despine() # remove top and
↳ right axes (w/o this, full box border)
plt.xlabel(x_label_name) # use matplotlib
↳ (plt) to adjust the x axis on this one
plt.ylabel(y_label_name) # use matplotlib
↳ (plt) to adjust the y axis on this one
plt.xticks(rotation=45,ha='right') # Rotate x-axis
↳ labels for better readability
plt.title(plot_title, y=1.1) # add a title so
↳ we know what we plotted!
plt.ylim(0,40); # use matplotlib
↳ (plt) to adjust the y axis limits

```

Did you guess it would be Taylor Swift?

4.6 Most Streamed Spotify Songs from 2023 Dataset

The data was downloaded from [kaggle](#) and has 24 columns: **track_name**: string, the title or name of the song **artist(s)_name**: string, the name of the artist or group responsible for creating the song **artist_count**: int, number of artists contributing to the song **released_year**: int, the year when the song was released **released_month**: int, the month when the song was released **released_day**: int, the day of the month when the song was released **in_spotify_playlists**: int, the number of Spotify playlists the song is included in **in_spotify_charts**: int, the number to indicate presence and rank of the song on Spotify charts **streams**: int, total number of streams on Spotify (ranges from thousands to billions) **in_apple_playlists**: int, the number of Apple Music playlists the song is included in **in_apple_charts**: int, the number to indicate presence and rank of the song

on Apple Music charts `in_deezer_playlists`: int, the number of Deezer playlists the song is included in `in_deezer_charts`: int, the number to indicate presence and rank of the song on Deezer charts `in_shazam_charts`: int, the number to indicate presence and rank of the song on Shazam charts `bpm`: int, beats per minute (bpm); a measure to indicate song tempo `key`: string, key of the song `mode`: string, mode of the song (major or minor) `danceability_%`: int, percentage indicating how suitable the song is for dancing `valence_%`: int, percentage indicating positivity of the song's musical content `energy_%`: int, percentage indicating the perceived energy level of the song `acousticness_%`: int, percentage indicating the amount of acoustic sound in the song `instrumentalness_%`: int, percentage indicating the amount of instrumental content in the song `liveness_%`: int, percentage indicating the presence of live performance elements `speechiness_%`: int, percentage indicating the amount of spoken words in the song

Explore the first and last 5 rows using `head` and `tail`

```
[ ]: ### RUN THIS CELL ###
spotify.head()
```

```
[ ]: ### RUN THIS CELL ###
spotify.tail()
```

5 Section 2: Variables and Data Types

Image [source](#)

Sometimes we might want to keep track of how a value is changing over time, or refer to a value over and over again. A **variable** is a container for values or data. Variables can be referenced later on, or changed over time.

In Python, a variable is created the moment we *assign* a value to it. In the following code, we are creating a **variable** `my_favorite_number` that holds the value 42.

Task: Review the code below as we walk through it, and run each cell.

```
[ ]: # assignment statement: assign the value 42 to my_favorite_number
my_favorite_number = 42
```

We can then reference the variable later on:

```
[ ]: # Show me the number
print(my_favorite_number)
```

Use the current value of the data in another variable:

```
[ ]: # Arithmetic operations
another_fun_number = my_favorite_number/2
```

```
[ ]: # What should this value be?
another_fun_number
```

Or change the value of the variable:

```
[ ]: # Re-assign the original value
my_favorite_number = my_favorite_number/6
my_favorite_number
```

From now on, if we use `my_favorite_number`, Python will substitute the value we assigned to it. The variable is just a name for a value. Naming it hopefully helps us remember what we did so we can use it again in the future! In Python, variable names: * can include letters, digits, and underscores * cannot start with a digit * are case sensitive (upper vs. lower case) Example: `hours_sleep` and `Hours_sleep` are two **different** variables!

5.1 2.0 Coding Exercise

Task: Run the code below to see our first error! Before you run the cell, think about why this will error based on what we just reviewed.

```
[ ]: # This should error- why?
42_fav_number = 42
```

Task: Edit the code above to fix the variable name so that it works and does not error. Put your code in the cell below.

```
[ ]: ### YOUR CODE HERE ###
```

```
[ ]: #@title Example Solution
fav_number_42 = 42
fav_number_42
```

As you've just seen, Python can be used as a calculator and you can easily perform many mathematical operations. Numbers fall into two categories, and additionally there are several other **data types**:

- **Integers:** These are values like 1, 2, 200, 0, -1, etc. Data of this type in Python are called `int`.
- **Floats:** Floats are numerical values that can take on decimal values as well. Data of this type in Python are called `float`.
- **Strings:** Strings are collections of numbers or letters that make up a word, name or sentence. They are usually declared using quotation marks. `"banana"`, `"42"`, and `"42 bananas"` are all strings. Data of this type in Python are abbreviated `str`.
- **Booleans:** True or False. In Python, this type is called `bool`.
- **Lists:** Collections of elements that have a distinct order. Python's data type for this is `list`. (We will get to this in the next section!)

Task: Run this cell.

```
[ ]: ### RUN THIS CELL ###
an_int = 5
a_float = 9.99
a_string = 'Hello World!'
```

```

a_new_string = "Python Crash Course"
a_bool = True
another_bool = 10<7

print(an_int)
print(a_float)
print(a_string)
print(a_new_string)
print(a_bool)
print(another_bool)

```

```

[ ]: #@title Example Solution
a_new_string

```

5.2 2.1 Coding Exercise

What will be the values of `number_a`, `number_b`, `string_a` after the following code is run?

```

number_a = 42
string_a = "forty two"
number_b = number_a/2
number_a = number_a/6

```

Task: Now type this out into a code cell and check if you were correct!

```

[ ]: ### YOUR CODE HERE ###

```

```

[ ]: #@title Example Solution
number_a = 42
string_a = "forty two"
number_b = number_a/2
number_a = number_a/6
print('number_a: ', number_a)
print('number_b: ', number_b)
print('string_a: ', string_a)

```

5.2.1 Tip

You can use the `%whos` command at any time to see what variables you have created and what modules you have loaded into the computer's memory.

```

[ ]: %whos

```

5.3 2.2 Coding Exercise

There is another word for data types in programming - classes. Variables of a specific class are sometimes called objects of that class.

In Python you can look at the class of a variable/object using the function `type()`.

Print the class types of in the following cell: * the value "abcd1234" * the result of $10 > 9$ * the sum of $2 + 1.5$ * whatever "hungry" + "hippo" returns * whatever "twenty" + 7 returns

Hint: One of these should cause Python to throw an error!

```
[ ]: ## YOUR CODE HERE ###

[ ]: #@title Example Solution
print(type('abcd1234'))
print(type(10 > 9))
print(type(2+1.5))
print(type("hungry" + "hippo"))
print('twenty' + 7)
```

6 Section 3: Lists

Many times we aren't just going to be working with single values - we want a way to store multiple values, strings, etc. **Lists** are built into python (so we don't have to load a library to use them). We create a list by putting values inside square brackets and separating the values with commas:

```
[ ]: our_first_list = [2, 'banana', [1,2,3], np.max, 2.5]
print(our_first_list)
```

Notice in the above example, the types within the list were heterogeneous. Lists can store elements of different types. In the following examples you will see examples of lists with all the same data types as well.

```
[ ]: ### RUN THIS CELL ###
# This code will become clear once we review indices in lists below
print(type(our_first_list[0])) #2
print(type(our_first_list[1])) #'banana'
print(type(our_first_list[2])) #[1,2,3] - a list within a list!
print(type(our_first_list[3])) #np.max - numpy max function
print(type(our_first_list[4])) #2.5
```

6.1 3.0 Coding Exercise

We just saw that we can initialize a list by using square brackets with elements separated by commas. We can also initialize an empty list by using just two square brackets.

Task: Fill in the cell below. The first list should contain as many items as you can think of (hint: **brown**, **blue**, **black** will get you started). The second list should be initialized to be empty.

Notice it won't produce any output. Based on what you've seen above so far, how can you see what is inside each list? Use the second cell to write your own code to show the output.

```
[ ]: colors_starting_with_b = ### YOUR CODE HERE ###
     colors_starting_with_k = ### YOUR CODE HERE ###
```

```
[ ]: ### YOUR CODE HERE ###
```

```
[ ]: #@title Example Solution
     colors_starting_with_b = ['brown', 'blue', 'beige', 'black']
     colors_starting_with_k = []
     print(colors_starting_with_b)
     print(colors_starting_with_k)
```

6.2 Indexing

Now a few new list concepts: We can reference a specific element in a list by using square brackets and the index of the element. Note that Python starts indexing at 0! The first element is element 0, the second element is element 1, etc. (Now do you understand why this notebook is numbered the way it is?).

6.3 3.1 Coding Exercise

Task: BEFORE you run the cell below, predict what you think the output will be - Don't peek yet! This is a good check of your understanding so far. Run the cell and add your own code to preview the output.

```
[ ]: # What will be the output here?
     flowers = ['violet', 'tulip', 'rose', 'gardenia']
     flower_with_thorns = flowers[2]
```

```
[ ]: ### YOUR CODE HERE ###
```

We can also add elements, or lists, to the end of lists using `append`. >**Task:** Review the line of code below and predict where `pluto` will be added. Then show the output to check your understanding.

```
[ ]: planets = ['mercury', 'venus', 'earth', 'mars', 'jupiter', 'saturn', 'uranus',
               ↪ 'neptune']
     planets.append('pluto')
```

```
[ ]: ### YOUR CODE HERE ###
```

We can also remove elements using `remove`. >**Task:** Run the line of code below to kick out pluto from our list - bye, pluto!

```
[ ]: # The remove function removes one instance of an element from the list
     planets.remove('pluto')
     print('Pluto is no longer a planet in our list :(\n', planets)
```

Lists can even take on multiple dimensions, and you can use multiple indexes (indices) for referencing. >**Task:** BEFORE you run the cell below, predict what you think the output will be - Don't peek yet! This is a good check of your understanding so far. Don't worry if you haven't gotten

the hang of it yet - this takes practice and we will review together. Use the code cell provided to print the result when you are ready to check!

```
[ ]: famous_trios = [['luke', 'leia', 'han solo'], ['dory', 'nemo', 'marlin'],  
    ↳ ['harry', 'hermione', 'ron'], ['blossom', 'bubbles', 'buttercup']]  
famous_trios.append(['aragorn', 'legolas', 'gimli'])  
the_chosen_one = famous_trios[2][0]
```

```
[ ]: ### YOUR CODE HERE ###
```

```
[ ]: #@title Example Solution  
famous_trios
```

6.4 List Review

The order of indices is [row #][column #]. And don't forget that Python indices start from 0 (and not 1)! The second row (`famous_trios[2]`) consists of: `['harry', 'hermione', 'ron']`. We then want position 0 `famous_trios[2][0]` which results to the output above.

Let's examine the figure below: Rows are colored according to the brackets: * row 0: Green * row 1: Blue * row 2: Purple

Columns are colored according to the highlighted ovals: * col 0: Pink * col 1: Orange * col 2: Yellow

Image [source](#)

6.5 A Preview of Class Structure

Many libraries in Python will use more sophisticated classes, that come with their own attributes and functions. For example, we will work with `numpy` and `pandas` in a bit.

A [list](#) is the most sophisticated class we looked at so far. It too has *attributes*, properties that belong to a specific object of the class, and *functions* that *belong to it*. You can call functions that belong to a certain class using `object_name.function_name(additional_parameters)`. We actually already did this above with both `append` and `remove`.

Here are some more examples of attributes and functions of the list class.

Sometimes functions that act on a class change the properties of an object, sometimes they return a value, and sometimes they do both.

```
[ ]: ### RUN THIS CELL ###  
  
my_sisters_pets = ['dog', 'dog', 'cat', 'cat', 'gecko', 'cat', 'cat']  
  
# The sort function sorts the given list alphabetically/numerically.  
my_sisters_pets.sort()  
print('My sisters pets, in alphabetical order: ', my_sisters_pets)
```

```

# The count function counts the number of times a certain element appears in
↳ the list.
print('My sister has ', my_sisters_pets.count('cat'), 'cats.')

# The remove function removes one instance of an element from the list.
my_sisters_pets.remove('cat') # My sister's cat got lost ):
print('My sister has ', my_sisters_pets.count('cat'), 'cats.')

# We've used the append function before!
my_sisters_pets.append('cat') # Luckily, a neighbor found him!
print('My sister has ', my_sisters_pets.count('cat'), 'cats.')

```

6.6 3.2 Coding Exercise

Create a list of 7 items of clothing that you know you currently have in your closet. Use just a word that generally describes each item - shirt , pants , coat , dress . The same type of item can appear multiple times.

Update your list according to the following events, and print the list after each update. * You donated a pair of pants ! (Or an item of your choice, if you aren't a big fan of pants). * You bought a really awesome jacket that you love. * You want to sort the items alphabetically, so that you can quickly organize your wardrobe. * You want to count the number of shirts that are in your wardrobe.

```
[ ]: ### YOUR CODE HERE ###
```

```

[ ]: #@title Example Solution
wardrobe = ['shirt', 'pants', 'pants', 'coat',
            'sweater', 'sweater', 'shirt']

print(wardrobe)
wardrobe.remove('pants')
print(wardrobe)
wardrobe.append('jacket')
print(wardrobe)
wardrobe.sort()
print(wardrobe)
wardrobe.count('shirt')

```

7 Section 4: Numpy Arrays 8 9 5 0 2

An **array** is a numpy datatype that looks like a numerical list. While in many ways arrays look and act like the list datatype, they have some special features that make them more powerful than the list structure when performing mathematical computation.

Let's talk about the basics of arrays: - Initializing an array: Creating an empty or other type of array. - Indexing of arrays: Getting and setting the value of individual array elements, or smaller

subarrays. - Reshaping of arrays: Changing the shape of a given array, and combining two or more arrays.

An array looks like:

```
[ ]: ### 1D Array ###  
np.array([1,2,3,4,2,3,4])
```

```
[ ]: ### 2D ARRAY ###  
np.array([[1., 4.5, 3.4],  
          [5., 1.3, 1.],  
          [7., 2.9, 0.8],  
          [6.3, 4.9, 0.4]])
```

```
[ ]: # It can contain all strings  
np.array(['eat','sleep','repeat'])
```

```
[ ]: # It can contain booleans  
np.array([True,False,False])
```

```
[ ]: # What happens if we mix types like we did with lists?  
np.array(['eat',1,2])
```

```
[ ]: # Intialize an empty array  
new_array = np.array([])  
new_array
```

7.1 Initializing an Array

There are a few different ways to initialize (create) an array. Above we already did two of these:
- Initialize an empty array using `np.array([])`. - Create an array with your own inputs using `np.array([1,2,3,4 etc.])`

You can also: - Read a list or matrix of numbers from a file into a numpy array using `np.loadtxt()`. (We won't test this one, more on this [here](#)) - Generate an array with specific dimensions and distributions using functions like `np.zeros()`, `np.ones()`, `np.random.random()`, or `np.random.randn()`.

7.2 4.0 Coding Exercise

Let's test our luck by heading to the roulette tables!

Task: Initialize an array that is the equivalent to rolling a six-sided dice, 50 sets (the number of states), 3 times (rolls) per state.

Hint: Check out the `np.random.choice()` function.

Print the dimensions of your array to check you did this correctly.

```
[ ]: state_dice_rolls = ### YOUR CODE HERE ###
```



```
[ ]: #@title Example Solution
state_dice_rolls = np.random.choice([1,2,3,4,5,6],
                                     size=(50,3))

print(state_dice_rolls.shape)
```

7.3 Navigating an Array: Slicing

Navigating a **numpy** array feels somewhat similar to navigating a list. We can access elements in a 1D array using `myarray[30,20]`. We can select multiple values or sections using the `:` operator to stand for “all elements leading up to or after a certain number”. For example, the slice `[0:4]` means: “Start at index 0 and go up to, but **not including**, index 4.” For multidimensional arrays, we can access elements using the same notation as lists (`myarray[1][2]`), or we can access elements using commas to separate the indices (`myarray[1,2]`). The comma notation allows us to select specific slices of the array as well. Check out the following code to understand some of the nuances of navigating an array.

Note: In multidimensional numpy arrays, it is best practice to have consistent numbers of elements per index. We should not have an array that looks like `np.array([[1,2,3],[4,5]])`

Task: Start by showing the output of `exercises_section1` and navigate each line by referring back to this array. BEFORE you run the second cell, predict what you think the output will be - Don’t peek yet! This is a good check of your understanding so far. Don’t worry if this is tough - this takes practice and we will review together.

```
[ ]: # Show the array we will work with to start
exercises_section1 = np.array([[0.0,0.1,0.2],[1.1,1.2,1.3],[2.0,2.1,2.2]])
exercises_section1
```

```
[ ]: # Run this after you review the output above
# Guess what you think each line below will show
this_exercise = exercises_section1[2,2]
print('this exercise:', this_exercise)

exercises_section2 = exercises_section1[1,:]
print('exercises_section1:', exercises_section1)

harder_exercises_completed = exercises_section1[:2,1:]
print('harder_exercises_completed:\n', harder_exercises_completed)
```

In the last example, `harder_exercises_completed`, you see that we also don’t have to include the upper and lower bound on the slice. - If we don’t include the lower bound, Python uses 0 by default - If we don’t include the upper bound, the slice runs to the end of the axis

Note: There is another option we didn’t test - I encourage you to try it out if you’ve finished the above and are waiting to continue - If we don’t include either (i.e., if we just use `:` on its own), the slice includes everything

7.4 4.1 Coding Exercise

Using the `state_dice_rolls` array, find out what number these states rolled on their second dice rolls: Nevada (index 27), New Hampshire (index 28), New Jersey (index 29), New Mexico (index 30), New York (index 31), North Carolina (index 32).

```
[ ]: ### YOUR CODE HERE ###

[ ]: #@title Example Solution
print(state_dice_rolls[27:33,1])
```

7.5 Reshaping and Combining Arrays

We can also reshape multidimensional arrays, and combine arrays.

The `.transpose()` command swaps the dimensions of the array (a 2x6 array becomes 6x2) without changing its data.

To add an element to a numpy array you can use `np.append(orig, array_to_append)` or `np.concatenate([array1, array2, array3])`.

Check out the following example for how `np.reshape()` and `np.append()` work.

```
[ ]: ### RUN THIS CELL ###

orig_array = [[1,2,3],[4,5,6],[7,8,9], [10,11,12]]
reshaped_array = np.transpose(orig_array) # 4 rows x 3 cols becomes: 3 rows x 4
↪ cols

print('orig_array:\n', orig_array)
print('\nreshaped_array:\n', reshaped_array)

to_add = np.array([13,14])
to_add2 = np.array([15,16])
to_add = np.concatenate([to_add, to_add2]) # Concat arrays together.
print('\nto_add:', to_add)

# Specify the axis that we want to add the array to
# (more on axes in a minute!)
final_array = np.append(reshaped_array, [to_add], axis=0)
print('\nfinal_array:\n ', final_array)
```

7.6 4.2 Coding Exercise

1. Reshape your array to be 3 x 50 dice rolls, rather than 50 x 3 dice rolls.
2. Print the 3rd dice roll from the 12th state.

```
np.array([[1,2,3,3,3,1,5,4,4...],
          [6,3,4,5,3,2,5,3,6...]])
```

```
[ ]: state_dice_rolls_resaped = None ### YOUR CODE HERE ###
```

```
[ ]: #@title Example Solution
```

```
state_dice_rolls_resaped = np.transpose(state_dice_rolls)
print(state_dice_rolls_resaped[2][11])
```

8 Section 5: Array Arithmetic

Numpy arrays are especially powerful when wanting to perform mathematical operations on list-like structures.

You may want to perform the same operation on every element in an array, use multiple arrays to perform specific operation, or perform some sort of mathematical operation on a specific dimension of an array. For example, you may want to take the mean of each “column” in an array, or the sum of every “row”. An **axis** is the dimension of the array you are interested in performing aggregation on (more on this later!)

8.1 Element-wise operations

You might want to perform the same operation on every element of an array. There are three options for doing this:

- Because numpy has become so popular, many functions that typically take in an integer or float, can also detect if the input is a **numpy** array, and will perform the function on every element in an array. This is of course the case for **numpy** functions, but is also often the case from functions included in various statistics and math libraries in Python.
- Mathematical operations in native Python code perform the same function on every element on an array.
- Some functions in libraries will only act on a single value or integer. We can use `np.vectorize()` to turn a function that normally just takes a single number as an input, to a function that takes in an array.

```
[ ]: ### RUN THIS CELL ###
```

```
# Numpy function that defaults to performing
# the same operation on every element
my_array = np.array([1,900,43])
print('np.sqrt:', np.sqrt(my_array))

# Using native python code.
sqrt_native_python = my_array**(1/2)
print('sqrt_native_python:', sqrt_native_python)

# Function to take approximate square root by
```

```

# searching for the closest root (to the .1)
def sqrt_approx(x):
    for i in range(round(x)):
        for dec in range(10):
            if (i+dec)**2 > x:
                return i+dec-.1

# Sqrt approx is only set up to work on scalar inputs (single number).
# Lets make a function called sqrt_vectorize()
# that we can apply to arrays.
sqrt_approx_vectorize = np.vectorize(sqrt_approx)

print('sqrt_approx_vectorize:', sqrt_approx_vectorize(my_array))

```

8.2 Operations with Similar Arrays

In this section we will discuss performing a computation using two or more identically-shaped arrays.

We might want to add the corresponding elements from two arrays together, divide them, etc. As long as the arrays have the *same dimensions*, we can use regular math operations to perform computations with multiple arrays, or we can use functions that are built to take in two or more arrays.

When you do simple arithmetic operations on arrays like: addition (+), subtraction (-), multiplication (*), and division (/), the operation is done element-by-element.

```

[ ]: ### RUN THIS CELL ###
height = np.array([5.1,6.5,4.6])
weight = np.array([120,250,100])
bmi = weight/height
print('bmi:', bmi)

```

8.3 5.0 Coding Exercise

Many times we want to do more than add, subtract, multiply, and divide elements. `numpy` has options to do more complex [operations](#).

It's time to start exploring our dataset!

Task: We will find the mean `bpm` from the songs in the dataset that are from `artist_name` Taylor Swift. The beginning code gives you all the data from Taylor Swift's songs. (we will go over what this is actually doing in a bit). You should fill in the code below and use `np.mean` to calculate the average.

```

[ ]: ts_data = spotify[spotify['artist_name'] == 'Taylor Swift'].bpm    # this is
    ↳ the array you need to take the mean in the next line

ts_mean = ### YOUR CODE HERE ###

```

```
[ ]: #@title Example Solutions
ts_data = spotify[spotify['artist_name'] == 'Taylor Swift'].bpm # this is the
↳array to take the mean of
# Various ways to do this:
print(np.mean(ts_data))
print(sum(ts_data)/len(ts_data))
print(ts_data.mean())
```

8.4 5.1 Coding Exercise

Task: Let's calculate the age of each song. Fill in the code below to find the difference between the current year (year) and the released_year array.

```
[ ]: # Calculating age
year = 2024
spotify_year = spotify.released_year # this is the array to calculate the
↳difference
age = ### YOUR CODE HERE ###
```

```
[ ]: #@title Solution
year = 2024
spotify_year = spotify.released_year
age = year - spotify_year
print(age)
print('Youngest', min(age))
print('Oldest', max(age))
print(min(spotify.released_year))
```

8.5 5.2 Coding Exercise

Spoiler Alert The top two artists in terms of streams are: Taylor Swift and The Weeknd.

Task: Write code below to calculate the percentage of songs that each artist is contributing to within this dataset. Check that you came up with the same percentage as listed [here](#). On this website, scroll down to where it shows the csv file, and then select the 'column' tab. Look at the row data for artist_name. Hint: Use length (len) to help you out in your calculations.

```
[ ]: ## Taylor Swift
ts_songs = len(spotify[spotify['artist_name'] == 'Taylor Swift']) # this is
↳the array to do the calculations on
per_ts_songs = ### YOUR CODE HERE ###
```

```
[ ]: #@title Example Solution
ts_songs = len(spotify[spotify['artist_name'] == 'Taylor Swift']) # this is
↳the array to take the mean of
per_ts_songs = round(ts_songs/len(spotify) * 100,2)
per_ts_songs
```

```
[ ]: ## The Weeknd
wknd_data = len(spotify[spotify['artist_name'] == 'The Weeknd']) # this is the
↳ the array to do the calculations on
per_songs = ### YOUR CODE HERE ###

[ ]: #@title Example Solution
wknd_data = len(spotify[spotify['artist_name'] == 'The Weeknd']) # this is the
↳ array to take the mean of
per_songs = round(wknd_data/len(spotify) * 100,2)
per_songs
```

9 Section 6: Pandas Dataframes

A very popular Python data science library is [pandas](#). Pandas is best for working with two dimensional matrices (with samples as rows and features as columns), where different columns may be of different data types. Lists and [numpy](#) arrays can very easily be converted to/from [pandas](#) formats. Pandas can also read in data into batches, making it a nice library for working with big datasets that can not be held entirely in memory.

Let's review our data from [spotify](#) (we loaded in the data into a [pandas](#) dataframe!).

```
[ ]: ### RUN THIS CELL ###
spotify.head(10) # .head() let's us look at the first N rows.
```

Which columns are strings, and which are numerical? You can figure this out (usually) just by looking at what's in the dataframe.

Task: Run the code below to see the Python data types again.

```
[ ]: ### RUN THIS CELL ###
spotify.info()
```

9.1 Navigating a Pandas Dataframe

Dataframes are defined by rows and columns. Each row has an **index** associated with it, and each column has a column name.

Task: Run the code below to print the column names and indexes (in this case the index just happens to be the same as the row number) of the data. (This is the default for the index, but it can be [changed](#). Will we test that out later)

```
[ ]: ### RUN THIS CELL ###
print('index: ', spotify.index)
print('columns: ', spotify.columns)
```

You may want to select several different objects from a [pandas](#) dataframe:

- To select a specific column, or columns, you can use the syntax `df.column_name` or `df[column_names]`. The second syntax works even if your column names have dots or spaces in them, and allows you to select several columns as once.

Task: Run the code below to show two of the column names from the data.

```
[ ]: ### RUN THIS CELL ###
spotify_2cols = spotify[['artist_name', 'streams']]
spotify_2cols.sample(n=15) # take a random
    ↳ sample of 15 to show us
```

- To select a specific row, you can either select the row index name using `.loc[]` or using the row number using `.iloc[]`.

Task: Run the cell below to reindex our dataframe so that the artist name (`artist_name` column) is the index. Then let's look at the row containing Guns N' Roses.

```
[ ]: ### RUN THIS CELL ###
spotify.index = spotify['artist_name']
print("Guns N' Roses row:", spotify.loc["Guns N' Roses"])
print('\n5th row:', spotify.iloc[4])
```

```
[ ]: ### RUN THIS CELL TO SEE AN ARTIST WITH MULTIPLE ROWS ###
print('Dua Lipa row:', spotify.loc['Dua Lipa'])
```

- Finally, you may want to select a single element or subset of elements. You can do this by using `loc[]` and using both the row index and column name as the key(s).

Task: Run the cell below to do this with our dataset

```
[ ]: ### RUN THIS CELL ###
spotify.loc[["Lady Gaga", "Lord Huron", "Michael Buble"],
            ['bpm', 'streams', 'released_year']]
```

Note: Rows, columns, or subsets of data in a dataframe act very similarly to numpy arrays.

9.2 Bonus: Bracket versus Dot Indexing

Figuring out when to use which depends on different factors, and we won't go into those today. I encourage you to play around test out the different ways as you like in the notebook.

More info can be found on these websites: [Pandas documentation](#) [Pandas tutorial](#) [DataSchool](#) <https://www.dunderdata.com/blog/> [DunderData Blog](#) We primarily used brackets above. Test out similar code below, but use `.` indexing.

Task: Choose a column name and use dot indexing to reference it below:
`df.column_name`

```
[ ]: ### YOUR CODE HERE ###
```

```
[ ]: #@title Example Solution
spotify.bpm
```

9.3 6.0 Coding Exercise

Task: Find the `track_name`, `released_year`, and `energy_percent` of 3 artists of your choosing. (from `spotify` dataframe)

```
[ ]: ### YOUR CODE HERE ###
## feel free to show yourself the dataframe first and scroll through it to
↳select your artists :)
```

```
[ ]: #@title Example Solution

spotify.loc[["Lana Del Rey", "Eminem", "The Chainsmokers Halsey"],
            ['track_name', 'released_year', 'energy_percent']]
```

9.4 Editing or Adding To a DataFrame

We learned how to reindex our dataframe using `.index`. We can edit our dataframes in other ways as well: * add/remove a column/row * edit the value of a specific cell, row, or column

Task: Run the following code for examples on this.

```
[ ]: ### RUN THIS CELL ###

# Add a column called "random_number"
spotify['random_number'] = np.random.random(len(spotify))
print('first 5 rows with random_number column:\n ',spotify.head(5))

# Add a row called "CCR_like"
# that is equal to the values for Creedence Clearwater Revival (CCR).
spotify.loc["CCR_like"] = spotify.loc["Creedence Clearwater Revival"]
print('\n\nlast 5 rows, with new CCR_like: \n',spotify.tail(5))

# Change the random_number value just for CCR_like.
spotify.loc['CCR_like', 'random_number'] = 12345
print('\n\nedited CCR_like: ', spotify.loc['CCR_like','random_number'])

# Drop the CCR_like row and the random_number column
# to return to our original dataframe.
spotify = spotify.drop('CCR_like')
spotify = spotify.drop('random_number', axis=1)

print('final dataframe (last rows): ')
spotify.tail()
```


9.5 6.1 Coding Exercise

We calculated the age of each song above and stored it in a variable called `age` (5.1 Coding Exercise). Now we can add the `age` values we calculated above for each song to our dataframe. There are multiple ways this could be done, but we are going to keep practicing using `loc`.

Task: Adjust this example code by replacing the `dataframe`, inserting the correct way to index the dataframe (replacing `...`) in brackets after `loc` and finally by setting it equal to our variable name that we want to use: `dataframe.loc[...,...] = variable_name`

```
[ ]: ### YOUR CODE HERE ###
dataframe.loc[...,...] = variable_name

[ ]: #@title Example Solution
spotify.loc[:, "age"] = age
print(spotify.columns)

[ ]: #@title Example Solution - Another Way
# first need to reset the index!
spotify = spotify.reset_index(drop=True)
spotify.insert(0, "age", age, allow_duplicates=True)
```

9.6 Data types

Values from a single column in a dataframe all have the same datatype (boolean, string, integers, floats, etc.). To see the datatypes for each column, you can use the `.dtypes` property to look over your dataframe's columns' types.

```
[ ]: ### RUN THIS CELL ###
print(spotify.dtypes)
```

Sometimes it can be useful to use the command `.astype('datatype')` to convert a specific column of your dataframe to a certain type (for example, if integers were misread as strings, or 0/1s were misread as integers instead of booleans).

```
[ ]: ### RUN THIS CELL ###

# Several of our column names are objects, but should be strings
# let's convert it to a regular old string!
spotify['mode'] = spotify['mode'].astype('string')
```

```
[ ]: ### RUN THIS CELL ###
spotify.info()
```

Notice that we do have more columns that are objects and we want them to be strings. We will come back to this - you will adjust the remaining datatypes using a for loop. Stay tuned for this in the next section!

9.7 6.2 Coding Exercise

We have a categorical column called `mode` that has two options: major or minor. Maybe we would want to work with this data in separate columns for analysis. To do so, we can convert these data into a 0 or a 1 (boolean) corresponding to the string from the original `mode` column and place these in 2 separate columns. `pandas` has a function to help us do this called `get_dummies`.

Adjust this example code by replacing the `dataframe` and inserting the column name for `columns` and `prefix`: `pd.get_dummies(dataframe, columns=[''], prefix=[''])`

```
[ ]: ### YOUR CODE HERE ###
spotify_dummies = pd.get_dummies(..., columns=[...], prefix=[...])

[ ]: #@title Example Solution
spotify_dummies = pd.get_dummies(spotify, columns=['mode'], prefix=['mode'])
spotify_dummies
```

Let's review what we did and what the data looks like now.

```
[ ]: ### RUN THIS CELL ###
spotify_dummies.columns

[ ]: ### RUN THIS CELL ###
dummy_cols = spotify_dummies[['mode_Major', 'mode_Minor']]
dummy_cols
```

Before we continue, let's set our index back to numbers instead of `artist_name`. This is needed for later plotting, and we don't want to forget to `reset`! >**Task:** Run the cell below.

```
[ ]: ### RUN THIS CELL ###
spotify = spotify.reset_index(drop=True)
```

10 Section 7: For Loops

Many times we want to write code to repeat a process multiple times, as well as be flexible to work in different situations. We can tell Python to execute some code a fixed number of times using a **for loop**. Just like function definitions use a particular syntax, a for loop looks like the following:

```
for COUNTER_VARIABLE in LIST_OF_VALUES:
    DO SOMETHING
    DO SOMETHING ELSE
```

- **for:** The keyword that tells Python we are about to enter a for-loop.
- **COUNTER_VARIABLE:** A variable that will take on the value of whatever element of the `LIST_OF_VALUES` you are on in the current for-loop cycle.
- **LIST_OF_VALUES:** A list of values that you wish to cycle through over the course of the for loop. This can be a list of values, or can use the `range()` function. Example in a bit!
- **DO SOMETHING, DO SOMETHING ELSE:** The code that you want Python to execute every cycle (or *iteration*) of the loop, **indented compared to the for line**.

A for loop can iterate over a list, generally the form [element1, element2, element3...]. If a for loop iterates over a list, the counter variable changes its value to the next element in the list every iteration. Here's an example for loop.

```
[ ]: ### RUN THIS CELL ###
num_fruits = 0
for fruit in ['banana', 'apple', 'cherry', 'pomegranate']:
    print(fruit, "is a fruit")
    num_fruits = num_fruits + 1

print('Wooho, we have', num_fruits, 'fruits!')
```

If you don't have a specific list you want to iterate over, but you know you want to perform a set of actions a fixed number of times, you can use the `range(n)` function in place of the list. The `range` function will let you cycle through the for loop a fixed number (n) of times. Technically, when using the `range` function, the counter variable takes on the values of 0, then 1, then 2, then 3, etc., but it is also ok if you do not reference the counter variable at all in your for loop body.

Here's an example.

```
[ ]: ### RUN THIS CELL ###
for step_i in range(5): # iterating through the for loop 5 times.
    print('We are on iteration', step_i)
```

10.1 7.0 Coding Exercise

Review the code below that prints individual letters of the word `music`. This is a bad (inefficient) approach. It doesn't scale - It doesn't work for longer words and will error for smaller words (imagine if you had thousands of rows of data!)

A better approach to print each letter in the word would be to use a for loop.

Task: Write a for loop in the second cell to produce the same output from the first cell.

```
[ ]: ### RUN THIS CELL ###
word = 'music'
print(word[0])
print(word[1])
print(word[2])
print(word[3])
print(word[4])
```

```
[ ]: ### YOUR CODE HERE ###
```

Task: Now test how flexible your for loop is. Try another word to see it in action!

```
[ ]: ### YOUR CODE HERE ###
```

```
[ ]: #@title Example Solution
word = 'music'
for char in word:
```

```
print(char)
```

10.1.1 A Note on Naming

In the above example, we used the name `char` which was short for `character`. This makes sense in our case, but see the example below. You have the freedom to choose any name that you want. In the case below, if we choose `dog` for the loop variable (counter variable), it will work as long as the same name is used when we start the loop and then within the loop. However, when writing code, it's best to use a name that is meaningful, otherwise it could be difficult to figure out what the loop is doing. Do your future self a favor and use what makes sense (and leave yourself comments!)

```
[ ]: ### RUN THIS CELL ###
word = 'bananas'
for dog in word:
    print(dog)
```

10.2 7.1 Coding Exercise

Use a for loop to convert the string “hello” into a list of letters: `["h","e","l","l","o"]`

Hint: You can create an empty list like this: `my_list = []`.

The solution also includes using our friend `.append` for working with lists (we practiced above).

```
[ ]: ### YOUR CODE HERE ###

[ ]: #@title Example Solution
my_list = []
my_string = "hello"
for char in my_string:
    my_list.append(char)
print(my_list)
```

10.3 7.2 Coding Exercise

Above we changed the `mode` column of data into a string. We need to do this for 3 other columns in our `spotify` dataset. They are: - `artist_name` - `track_name` - `key`

Task: Edit the skeleton code below to use a for loop to change the datatype from object to string. If you need to reference what we did above, it was the section on Data Types.

```
[ ]: ### EDIT THIS CODE TO RUN ###
col_names = [..., ..., ...]

for ... in ...:
    spotify[...] = spotify[...].astype('string')

spotify.info() # use this to verify the cols are strings
```

```
[ ]: #@title Example Solution
col_names = ['artist_name', 'track_name', 'key']

for c in col_names:
    spotify[c] = spotify[c].astype('string')

spotify.info()

# What we previously used for reference:
# spotify['mode'] = spotify['mode'].astype('string')
```

10.4 Bonus: for loops to count data

We know that there are some artists that appear to be dominating this dataset. We can use a for loop to go through and count the songs per each unique artist. This for loop is a bit more complex and uses slightly different syntax than the ones we practiced above.

This is one example of how we can write our own code to review the data manually. In a few sections, we will see we can similarly look at descriptives using methods like `groupby` instead. This is already built in with `pandas` and may be more efficient than writing this out by hand with a for loop. But, this is still good practice as we are learning! For example, you could try to produce this same output using multiple methods to compare.

```
[ ]: # For loop to find and print number of songs per artist
nums = []
↳ # This will allow us to do an overall count too

for i, artist_name in enumerate(spotify.artist_name.unique()):
↳ # we go through one-by-one each unique artist name - remember, we can have
↳ up to 7 together!
    artist_data = spotify[spotify.artist_name == artist_name]
↳ # we find all rows in our dataframe that are for the artist(s)
    artist_count = len(artist_data)
↳ # we use `len` (short for length) to get the # of rows of data (each row
↳ in dataset is a song)
    nums = np.append(nums, artist_count)
↳ # we created an empty array called nums before our loop, and we can add
↳ the numbers to it for each artist
    print(f'# of songs for {artist_name}: ' + str(artist_count))
↳ # this prints out text to tell us how many songs were counted for each
↳ artist

print('Total # of songs in dataset: ' + str(sum(nums)))
↳ # this is to just double check we get out the number we expect!
```

11 Section 8: Conditional Statements

We've started working with the dataset, but so far you probably feel like you know very little about all of the data it contains. That's okay- we have tools we can use to write **robust** code to find information in our dataframe when we don't necessarily know specific information yet at the start. To do so, we can write code that uses the following logic:

If a certain condition is **True**, execute the code one way. If that condition is not true (**False**), execute the code a different way.

We can implement this logic using a **conditional statement**. The syntax for a conditional statement in Python looks like the following:

```
if CONDITION:
    DO SOMETHING
elif ANOTHER_CONDITION:
    DO ANOTHER THING
else:
    DO SOMETHING ELSE
```

- **if**: a keyword that lets Python know you are implementing a conditional.
- **CONDITION(AL)**: A boolean function or variable that returns the value **True** or **False**. The word **boolean** means exactly that - a function or variable can only return True or False.
- **DO SOMETHING**: The body of your code you wish to execute if the **CONDITION** is **True**.
- **elif** (optional): A keyword that lets Python know that it should execute the following body of code if the condition is **True**. (and the previous condition was not true)
- **DO ANOTHER THING** (optional): The body of code you wish to execute if **ANOTHER_CONDITION** is **True**.
- **else** (optional): A keyword that lets Python know that it should execute the following body of code if the condition is not true.
- **DO SOMETHING ELSE** (optional): The body of code you wish to execute if the **CONDITION** or **ANOTHER_CONDITION** is **False**.

We will explore how to use each of these different components of an if-then statement in the following exercises.

We already have some built in functions in **pandas** that return a **boolean** depending on the condition you are looking for in the dataset.

is_na(): Returns **True** if there are missing values (NaNs), and **False** if there is not.

Task: What will the following block of code print? **True** or **False**? Run the code below to see if you are correct.

```
[ ]: ### RUN THIS CELL ###
pd.isna('Python Crash Course')
```

```
[ ]: ### RUN THIS CELL ###
pd.isna(np.nan)
```

```
[ ]: ### RUN THIS CELL ###
# Want to know what np.nan does? Try running just that part here
np.nan
```

Here is a more interesting example to examine any (and all) missing data from our dataframe.

```
[ ]: ### RUN THIS CELL ###
spotify.isna().sum()
```

Why did `sum` work with `True` and `False` values? To count these, we use `True = 1` and `False = 0`. So the above `sum` is really a count of how many `True` (`nan`) values were found. Note: Pandas supports **missing values**. If a value is missing it will be a “nan” value. Many functions (`mean`, `sum`, etc.) have specific parameters you can choose for how to deal with missing values (do we ignore it, do we treat it as a zero, do we return a null?). This is a decision you will need to make based on your question of interest or purpose for running your code/analyses.

We can also use logical operators with our conditional statements: * “is equal to” `==` * “is not equal to” `!=` * “is less than” `<` * “is greater than” `>`

Task: Predict what Python will do with the following logical operator. Then run the cell to check your predictions. Try changing the value of `age` and `day` and see what happens.

```
[ ]: age = 65
day = "Tuesday" # Tuesdays are senior nights!
if (age >=65) and (day == 'Tuesday'):
    print("You get a senior discount!")
else:
    print("Sorry, you do not quality for a senior discount.")
```

11.1 Using Conditional Statements to Select Dataframe rows

We learned above how to pull out data from an entire column (or columns). Now, we want to pull out data from rows (across multiple columns) that meets a certain condition. The logic is similar, but we need to add a bit more to our code. Remember that our data is similar within a column, but each row contains a mix of data across the columns. For example, some values are text (strings) and some are numbers. To filter a `DataFrame` based on the contents by rows in `pandas`, we need to use boolean (`True` or `False`) expressions in order to select the rows we want to keep. Conditional expressions (`>`, `<`, `==`, `!=`, `<=`, ...) enable us to test whether our conditions of interest would evaluate to `True` or `False`. Typically, we want to keep data in our rows that meets a certain criteria (`True`). Let’s walk through an example.

Task: Run this code. Predict what you think you will see once it runs.

```
[ ]: ### RUN THIS CELL ###
spotify['in_apple_playlists'] > 50
```

We put in the name of our column `in_apple_playlists` and asked Python to find where this value is greater than `> 50`. You can see from the output that sometimes values in this column are `True`, meaning this corresponds to a row that the song was in more than 50 apple playlists. If you see `False`, this indicates that the value in the row did *NOT* meet our condition (greater than 50). This is helpful to illustrate the values we would see if we *only* look at the column we specified `in_apple_playlists`.

We want to take this a step further, because we don't want to work with just one column in our dataset! It's important to see what Python is doing and how it interprets the code, since this is an intermediate step in the code we will run below. We want to use the code we've written above but now apply this filter across the entire dataset. So, we want to pull out **all rows** where data was in more than 50 apple playlists, and also keep all columns in our dataset. To do this, we have to specify that we want to do this filtering in the context of our dataframe `spotify`. The line of code below will reference the dataframe first and then put the code we ran above inside brackets to subset the entire dataset. We will get out only rows that meet our condition `>50 apple playlists` and also still have all of the columns in our dataframe to work with.

Task: Review this code and compare it to above. Also review the output.

```
[ ]: ### RUN THIS CELL ###
apple_subset = spotify[spotify['in_apple_playlists'] > 50]
apple_subset
```

```
[ ]: ### RUN THIS CELL ###
# How many rows met this condition?
print(len(apple_subset))
```

11.2 8.0 Coding Exercise

Choose your own adventure! Pick something you would like to filter the dataset by and test out your code. If you need a reminder of all the columns, run the cells below to preview some more info about the data. Alternatively, type the name of the dataset so you can scroll through it to pick something you are interested in.

If all else fails and you aren't feeling creative (or are stuck and don't know where to start), use the code above and adjust it so that you find the rows of data for Queen (`artist_name`).

```
[ ]: ### YOUR CODE HERE ###
```

```
[ ]: #@title Example Solution
# try filtering by an artist name
spotify[spotify['artist_name'] == 'Queen']
```

11.3 8.1 Coding Exercise

Using a for loop and an if statement, in combination with the other concepts you have learned, build some code to count the number of songs in the dataframe that are over 20 years old.

```
[ ]: ### YOUR CODE HERE ###
```



```
[ ]: #@title Example Solution
songs_over_20_counter = []
songs_under_20_counter = []

for i in spotify.index:
    if spotify.age[i] >= 20:
        songs_over_20_counter.append(1)
    else:
        songs_under_20_counter.append(1)

print('# of songs over 20:', sum(songs_over_20_counter))
print('# of songs under 20:', sum(songs_under_20_counter))
print('Check of total # of songs:', sum(songs_over_20_counter) +
↪sum(songs_under_20_counter))
```

```
[ ]: #@title Example Solutions - Other Options
#for i in range(len(df)):
#for index, row in df.iterrows():
```

12 Section 9: Reviewing the Dataset

So far we've learned about how to use Python and worked with parts of the dataset. Maybe you feel like you've started to know the dataset, but there's a lot of data! Let's take stock of the data we have available to us. I will demonstrate some of my most commonly used functions and methods. These will help to get at what data we have and how find out the info we want to describe the data from our dataframe.

Task: Let's start by reviewing the dataframe like we already have above. We will use `.info()` to remind ourselves what info we have. Run this cell.

```
[ ]: ### RUN THIS CELL ###
spotify.info()
```

12.1 Unique values

Luckily, we don't have to do everything by eye and scroll the whole way through the table (although I do recommend doing this to get a feel for the data!). We can use the function `unique` in `pandas` to help us out. It will return the unique values from the column we specify based on the order of appearance (it does not sort them in any way).

Helpful info can also be found [here](#).

Below, we will first call the name of our dataframe (`spotify`) then reference the column name (`.column_name`) and finally use `.unique()` to figure out the possible values in our column. The code we will use looks like this: `dataframe.column_name.unique()`

12.2 9.0 Coding Exercise

Until now, we don't actually know what all the options are for `artist_count` in the dataset - did you manually count above ?

Task: Use the example syntax above to write code that will show the `unique` numbers contained in the `artist_count` column. This will give us more info about artist collaborations in the dataset.

```
[ ]: ### YOUR CODE HERE ###
```

```
[ ]: #@title Example Solution
print('Counts of artists:', spotify.artist_count.unique())
```

12.3 Sorting

As you can see, this isn't sorted in order, but the numbers appear based on how they appeared in the dataset. Many times it helps us to arrange our data in logical way that makes sense for our brain . Sorting is particularly helpful for this!

If you want to `sort` them, can you find a way to do so?

Hint: There are many ways to do so! If you are ever stuck, you can try googling the combination of the package you are using (`numpy`) and what you'd like to do with the data (`sort`).

```
[ ]: ### YOUR CODE HERE ###
```

```
[ ]: #@title Example Solution
print('Counts of artists using np.sort:', np.sort(spotify.artist_count.
↪unique()))
```

```
[ ]: #@title Example Solution- Another Way
print('Counts of artists using sorted:', sorted(spotify.artist_count.unique()))
```

12.4 9.1 Coding Exercise

It's also possible to sort the entire dataframe. `pandas` uses `sort_values` to arrange the dataframe depending on what you'd like to sort by.

Let's take a look at the dataframe sorted by `released_year`. Just how old are the oldest songs in our dataset?

Task: Use the skeleton code here and add it to a cell below. Sort based on `released_year`. `sorted_df = dataframe.sort_values(by=['...'])`

Note: `sort_values` defaults to the order of sorting as `ascending = True`. If you want to change this, add this parameter and set it to false to sort from most recent to oldest instead.

```
[ ]: ### YOUR CODE HERE ###
```

```
[ ]: #@title Example Solution
# Surpring insight- christmas songs in our list?
```

```
sorted_year = spotify.sort_values(by=['released_year']) # default is ascending
↳ = True
sorted_year
```

Interesting! Just scrolling through the top of the list, there are some Christmas classic songs in there that still seem to be widely streamed!

12.5 Descriptive Statistics

12.6 9.2 Coding Exercise

We will now examine descriptive statistics of our different variables. We are typically interested in different parameters, such as: counts, means, and standard deviation. This will tell us more about our sample – it will describe the data we have available to work with.

There is a built in method to do just this with `pandas` called `describe`.

The code looks like this: `dataframe.describe()`

Task: Write code below using our dataframe name to describe the data.

```
[ ]: ### YOUR CODE HERE ###
```

```
[ ]: #@title Example Solution
spotify.describe()
```

Notice that by default, the information is calculated for only the columns that contain **numerical** data. It is possible to specify additional parameters for the `describe` method that will ask for **categorical** info to be displayed as well.

Task: Run the cell below to see the output that includes **all** datatypes. Notice how we will see NaNs for our numerical columns where the descriptives don't apply.

```
[ ]: ### RUN THIS CELL ###
spotify.describe(include = 'all')
```

12.7 Groupby

We could instead create our own descriptive table based on a subset of information we are interested in. Grouping allows us to arrange our data in a way we want to review it. You can think of grouping as splitting the dataset data into buckets . Then you can call “aggregate” functions (`mean`, `sum`, `max`, `min`, etc) on these buckets to find these values per bucket (which can lead to interesting analysis)!

We will use `groupby` in `pandas` to help us out.

A group by operation consists of two parts: 1. We use `.groupby` to group rows together that have the same value for the column(s) we are interested in. Typically, you would always group by categorical columns, not number columns. E.g. If you are interested in analyzing the major versus minor mode of songs, we might do: `spotify.groupby('mode')`

2. An aggregation function. This is the computation you want to perform across the sets of rows that were grouped together during step 1. `.count()`, `.mean()`, and `.sum()` are common aggregation functions, but you can also pass your own as an argument to `.aggregate()`.

Here are examples of using `groupby` to examine our data further

```
[ ]: # Calculate summary statistics
print("Summary statistics:")

# group and aggregate the values
summary_stats = spotify.groupby(by=['artist_name']).agg(
    {'artist_name': 'count', 'bpm': ['mean', 'std']}).round(2)

# display
display(summary_stats)
```

Reviewing the above code:

In this example, we wanted to see the **count** of `artist_name` and **mean** and **standard deviation** of `bpm` for each artist `artist_name`.

What we did with the code was:

1. `groupby` one column name. We only used one column above, but you can add more than one.
2. After we use `groupby`, we apply the aggregation method `.agg()` and specify a dictionary in a following way: `{column_name: aggregation function}`. We applied multiple functions at once on different columns.

Notice: We also applied multiple functions on the same column `bpm` by specifying a list that contained multiple functions in brackets like `["mean", "std"]`. Also, we have NaNs in our standard deviation column. This is why I also included the count column. It makes sense that if we only have 1 song (1 row of data), then we cannot calculate any deviation. This is a good way to check that the values you get from your data make sense!

12.8 9.3 Coding Exercise

Now it's your turn to practice using `groupby`. You can start by copy and pasting the code from above. Here we will `groupby` `artist_name` and then aggregate using both `count` (same as above), but instead finding the `min` and `max` of `released_year`. Adjust your code and test it out to review which artists have been releasing songs across multiple years, or to determine if some were one hit wonders!

```
[ ]: ### YOUR CODE HERE ###
```

```
[ ]: #@title Example Solution
# Calculate summary statistics
print("Summary statistics:")

# group and aggregate the values
summary_stats = spotify.groupby(by=['artist_name']).agg(
    {'artist_name': 'count', 'released_year': ['min', 'max']})
```

```
# display
display(summary_stats)
```

12.9 Bonus: Dictionaries

Here we have a new type of data structure that I likely won't go into detail in this workshop. If you want more information, read on, otherwise feel free to skip this extra bit. There is a datatype in python called a dictionary that we used above. Dictionaries store data in key:value pairs. Also note that we have a new type of brackets or braces that are curly { }

```
new_type = {'Number of petals': [8, 34, 5], 'Name': ['lotus', 'sunflower', 'rose']}
type(new_type)
dict
```

For example, here are key: value pairs: 'Number of petals': [8,34,5] 'Name': ['lotus','sunflower', 'rose'] The keys are the strings for the column names, and the values are the data that go in the rows of the table.

Dictionaries are structures which can contain multiple data types. They contain key-value pairs. For each unique key, the dictionary has one value. Keys can be various data types: strings, numbers, or tuples, while the corresponding values can be any Python object.

You cannot access values of the dictionary by the indexes (like you can in lists or arrays). But you can access them by the key. Due to this feature dictionaries don't allow duplicated keys.

You can also access just the keys or just the indexes by `.keys()` and `.values()` methods. To find out even more, read the Python [documentation](#).

Other examples and [tutorial](#)

13 Section 10: Plotting with Seaborn (and Matplotlib)

[Seaborn](#) is a library for data visualization and figure making. Seaborn works very well with **pandas** dataframes, and you can modify the appearances of your plots using similar settings to how we did it in **numpy**. It can also work together with **matplotlib**. In **seaborn**, we: (1) decide what plotting function to use, (2) pass the function a **full dataframe**, (3) specify which columns in our dataframe we want to use as our x-axis, y-axis, colors, etc. and (4) specify any additional parameters specific to the plotting function.

```
sns.PLOT_FUNCTION(
    data=YOUR_DATA,
    x='COL TO USE FOR X',
    y='COL TO USE FOR Y',
    hue = 'COL YOU WANT TO SET COLOR ACCORDING TO',
    additional_params=...)
```

Task: Review and run the cells below to see examples.

Let's create a plot to examine if there is a relationship between bpm and energy %? Does this vary by mode?

```
[ ]: ### RUN THIS CELL ###

# Set this text for the plot- easiest to edit here as needed
x_label_name = 'Beats per minute (bpm)' #
    ↳ x-axis label text
y_label_name = 'Energy %' #
    ↳ y-axis label text
leg_title = 'Mode' #
    ↳ legend title - column name for hue
plot_title = 'Relationship between BPM and Energy % by Mode' #
    ↳ plot title text
fig_name_scatter1 = "Scatter_BPM_Energy_Mode.png" #
    ↳ text to name figure for saving

# Example of a scatter plot
sp = sns.scatterplot(#
    ↳ plotting option to use in seaborn
    data=spotify, #
    ↳ dataframe
    x='bpm', #
    ↳ column name to use values for x-axis
    y='energy_percent', #
    ↳ column name to use values for y-axis
    hue='mode') #
    ↳ color code our dots by mode column values

# Adjustment + aesthetics of plot
sp.figure.set_size_inches(6.5, 4.5) #
    ↳ this sets the size and we will use to be consistent across all plots below
sns.despine() #
    ↳ remove top and right axes (w/o this, full box border)
plt.xlabel(x_label_name) #
    ↳ use matplotlib (plt) to adjust the x axis on this one
plt.ylabel(y_label_name) #
    ↳ use matplotlib (plt) to adjust the y axis on this one
plt.title(plot_title, y=1.05) #
    ↳ add a title so we know what we plotted!
plt.ylim(0,100) #
    ↳ use matplotlib (plt) to adjust the y axis limits
plt.legend(bbox_to_anchor=(1.1, 0.05), loc='lower right', title=leg_title); #
    ↳ use matplotlib (plt) to adjust the legend location and label

# Save our figure
```

```
plt.savefig(fig_name_scatter1, dpi=300, bbox_inches='tight') #_
↳ save w/figure name
```

Let's create a boxplot to examine if there are differences between number of **streams** depending on **release_month** and by **mode**.

```
[ ]: ### RUN THIS CELL ###

# Set this text for the plot- easiest to edit here as needed
x_label_name = 'Release Month' #_
↳ x-axis label text
y_label_name = 'Streams' #_
↳ y-axis label text
leg_title = 'Mode' #_
↳ legend title - column name for hue
plot_title = 'Streams of Songs Released Over Month by Mode' #_
↳ plot title text
fig_name_box = "Boxplot_ReleaseMonth_Streams_Mode.png" #_
↳ text to name figure for saving

# Example of a scatter plot
bp = sns.boxplot( #_
    ↳ plotting option to use in seaborn
    data=spotify, #_
    ↳ dataframe
    x='released_month', #_
    ↳ column name to use values for x-axis
    y='streams', #_
    ↳ column name to use values for y-axis
    hue='mode') #_
↳ color code our dots by mode column values

# Adjustment + aesthetics of plot
bp.figure.set_size_inches(6.5, 4.5) #_
↳ this sets the size and we will use to be consistent across all plots below
sns.despine() #_
↳ remove top and right axes (w/o this, full box border)
plt.xlabel(x_label_name) #_
↳ use matplotlib (plt) to adjust the x axis on this one
plt.ylabel(y_label_name) #_
↳ use matplotlib (plt) to adjust the y axis on this one
plt.title(plot_title, y=1.1) #_
↳ add a title so we know what we plotted!
plt.ylim(0,4000000000) #_
↳ use matplotlib (plt) to adjust the y axis limits
```

```
plt.legend(bbox_to_anchor=(1.2, 1.05), loc='upper right', title=leg_title); #
↳ use matplotlib (plt) to adjust the legend location and label

# Save our figure
plt.savefig(fig_name_box, dpi=300, bbox_inches='tight') #
↳ save w/figure name
```

13.1 A Few Tips on Data Visualization

Although there are many reviews and studies discussing [best practices for figures](#), we will list a couple of pandas plots you can utilize when you wish to visualize various relationships:

Visualizing a single variable - Histogram (`sns.hist()`) - Barplot, i.e. for counts of a single variable (`sns.bar()`)

Visualizing multiple numeric variables - Scatter plot/line graph (`sns.scatter`) - 2d histogram plot (`sns.jointplot()`)

Visualizing categorical & numerical variable - Violinplot/Boxplot (`sns.violinplot()`, `sns.boxplot()`) - Stripplot or swarm plot (`sns.stripplot()`, `sns.swarmplot()`) - Barplot (`sns.barplot()`, `sns.boxplot()`)

Visualizing two categorical variables - Heatmap (`sns.heatmap()`)

To visualize 3 variables, consider encoding one variable as the color. To visualize 4+ variables, consider using [facetgrid](#).

13.2 10.1 Coding Exercise

Let's look at the top 15 artists with the most songs. We will create a barplot to do this. In this example, we are going to create a smaller subset of the data and reference that instead of the entire dataframe.

Task: Run the first 3 cells to get a feel for the data and then to create an initial plot. Notice that the plot is missing lots of details! Skeleton code is provided to adjust the labels and aesthetics. Adjust the provided code below to create the plot and see which artists fall within the top 15!

```
[ ]: ### RUN THIS CELL ###
artist_counts = spotify['artist_name'].value_counts()
artist_counts.head(50)

[ ]: ### RUN THIS CELL - THIS IS THE DATA OF THE TOP 15 TO USE ###
artist_plot = artist_counts.head(15)
artist_plot

[ ]: ### RUN THIS CELL ###
ax = sns.barplot(                                     # plotting
    ↳ option to use in seaborn
    x = artist_plot.index,                             # column name to
    ↳ use values for x-axis
```



```

    y = artist_plot,                                # column name to
    ↪use values for y-axis
    hue = artist_plot,                              # color code our
    ↪dots by mode column values
    palette = 'flare')                             # set the color
    ↪palette we want to use

```

Blah! That doesn't look very nice. Who are the artists?

Task: In the code cell below, add all of the labels at the beginning and then adjust the aesthetics. Use the code examples for the initial plots to see what you should do.

```

[ ]: ### EDIT THIS CELL ###
# Set this text for the plot- easiest to edit here as needed
# My solution included 4 lines here
...

# Barplot
ax = sns.barplot(                                  # plotting
    ↪option to use in seaborn
    x = artist_plot.index,                          # column name to
    ↪use values for x-axis
    y = artist_plot,                                # column name to
    ↪use values for y-axis
    hue = artist_plot,                              # color code our
    ↪dots by mode column values
    palette = 'flare')                             # set the color
    ↪palette we want to use

# Adjustment + aesthetics of plot
# My solution included 6 lines here
...
plt.xticks(rotation=45,ha='right')                 # Rotate x-axis
    ↪labels for better readability

# My solution included code to save the plot
...

```

```

[ ]: #@title Solution
# Set this text for the plot- easiest to edit here as needed
x_label_name = 'Artist Name'                       #
    ↪x-axis label text
y_label_name = 'Number of Songs'                   #
    ↪y-axis label text
plot_title = 'Top 15 Artists With Most Songs'      # plot
    ↪title text

```

```

fig_name_bar = 'Top15_artists_bar.png' # text
    ↳to name figure for saving

# Example of barplot
ax = sns.barplot( # plotting
    ↳option to use in seaborn
    x = artist_plot.index, # column name to
    ↳use values for x-axis
    y = artist_plot, # column name to
    ↳use values for y-axis
    hue = artist_plot, # color code our
    ↳dots by mode column values
    palette = 'flare') # set the color
    ↳palette we want to use

# Adjustment + aesthetics of plot
ax.figure.set_size_inches(6.5, 4.5) # this sets the
    ↳size and we will use to be consistent across all plots below
sns.despine() # remove top and
    ↳right axes (w/o this, full box border)
plt.xlabel(x_label_name) # use matplotlib
    ↳(plt) to adjust the x axis on this one
plt.ylabel(y_label_name) # use matplotlib
    ↳(plt) to adjust the y axis on this one
plt.xticks(rotation=45,ha='right') # Rotate x-axis
    ↳labels for better readability
plt.title(plot_title, y=1.1) # add a title so
    ↳we know what we plotted!
plt.ylim(0,40); # use matplotlib
    ↳(plt) to adjust the y axis limits

# Save our figure
plt.savefig(fig_name_bar, dpi=300, bbox_inches='tight') #
    ↳save w/figure name

```

13.3 10.2 Coding Exercise

Let's look at the number of songs released per month. We will create a barplot to do this. In this example, we are going to create a smaller dataframe first.

Task: Run the first cell to get the data we want to work with and then create an initial plot. Notice that the plot is missing lots of details! Skeleton code is provided to adjust the labels and aesthetics. Adjust the provided code below to create the plot and see which months have the most songs released.

```

[ ]: ### RUN THIS CELL ###
songs_over_time = spotify['released_month'].value_counts().reset_index()

```

```
songs_over_time.columns = ['released_month', 'count']
songs_over_time
```

```
[ ]: ### RUN THIS CELL ###
b2 = sns.barplot(                                     # plotting option to
    ↪ use in seaborn
    data = songs_over_time,                           # dataframe
    x = 'released_month',                             # column name to use
    ↪ values for x-axis
    y = 'count',                                       # column name to use
    ↪ values for y-axis
    hue = 'released_month',                           # color code our bars
    palette = 'viridis')                             # set the color
    ↪ palette we want to use
```

Again we've generated an initial plot that looks okay, but we can improve! We also need to move (remove) that legend.

Task: Edit the code below to beautify the plot Hint: To remove the legend, set `legend = False` in the part where `b2` is defined within seaborn.

```
[ ]: ### EDIT THIS CELL ###
# My solution included 4 lines here
...

b2 = sns.barplot(                                     # plotting option to
    ↪ use in seaborn
    data = songs_over_time,                           # dataframe
    x = 'released_month',                             # column name to use
    ↪ values for x-axis
    y = 'count',                                       # column name to use
    ↪ values for y-axis
    hue = 'released_month',                           # color code our bars
    palette = 'viridis')                             # set the color
    ↪ palette we want to use
# My solution added a line here to remove the legend

# My solution included 6 lines here
....

# My solution included code to save the plot
...
```

```
[ ]: #@title Example Solution
x_label_name = 'Release Month'                       #
    ↪ x-axis label text
```

```

y_label_name = 'Number of Songs'                                #
    ↳y-axis label text
plot_title = 'Number of Songs by Release Month'                #
    ↳plot title text
fig_name_bar2 = 'NumSongs_ReleaseMonth.png'                    #
    ↳text to name figure for saving

b2 = sns.barplot(                                                # plotting option to
    ↳use in seaborn
    data = songs_over_time,                                     # dataframe
    x = 'released_month',                                       # column name to use
    ↳values for x-axis
    y = 'count',                                                # column name to use
    ↳values for y-axis
    hue = 'released_month',                                     # color code our bars
    palette = 'viridis',                                       # set the color
    ↳palette we want to use
    legend = False)                                            # remove the legend

# Adjustment + aesthetics of plot
b2.figure.set_size_inches(6.5, 4.5)                             # this sets the
    ↳size and we will use to be consistent across all plots below
sns.despine()                                                    # remove top and
    ↳right axes (w/o this, full box border)
plt.xlabel(x_label_name)                                         # use matplotlib
    ↳(plt) to adjust the x axis on this one
plt.ylabel(y_label_name)                                         # use matplotlib
    ↳(plt) to adjust the y axis on this one
plt.title(plot_title, y=1.1)                                    # add a title so
    ↳we know what we plotted!
plt.ylim(0,140);                                                # use matplotlib
    ↳(plt) to adjust the y axis limits

# Save our figure
plt.savefig(fig_name_bar2, dpi=300, bbox_inches='tight')        # save w/figure
    ↳name

```

13.4 10.3 Coding Exercise

Now let's look at another type of plot to examine correlation values. We will use a heatmap to correlate the following variables: streams bpm danceability_percent valence_percent energy_percent acousticness_percent instrumentalness_percent liveness_percent speechiness_percent

First we will create a list of the column names we want to correlate. Then we will use `.corr()` to

get the correlation matrix. Example call: `corr_matrix = dataframe[columns_to_corr].corr()`

Once we have these values, we can create our plot in `seaborn`.

Task: Edit the code below to create our heatmap.

```
[ ]: ### EDIT THIS CELL ###
# Setup dataframe
columns_to_correlate = ...
corr_matrix = ...

# Title and saving name
# My solution had 2 lines here
....

# Plotting code
hm = sns.heatmap(corr_matrix,          # corr matrix as df
                  annot=True,          # include values on map
                  cmap='crest',        # change the color if you want
                  fmt=".2f")           # 2 decimals for displayed values

# Adjustment + aesthetics of plot (size and title only)
# My solution had 2 lines here

# Save the fig
# My solution had one line to save the fig
```

```
[ ]: #@title Example Solution

# Setup dataframe
columns_to_correlate = ['streams', 'bpm', 'danceability_percent',
                        ↪ 'valence_percent', 'energy_percent', 'acousticness_percent',
                        ↪ 'instrumentalness_percent', 'liveness_percent', 'speechiness_percent']
corr_matrix = spotify[columns_to_correlate].corr()

# Labels and title
plot_title = 'Correlation Heatmap'          # plot title text
fig_name_hmp = 'Corr_heatmap.png'          # text to name ↵
                        ↪ figure for saving

# Plotting code
hm = sns.heatmap(corr_matrix,              # corr matrix as ↵
                  ↪ df
                  annot=True,              # include values ↵
                  ↪ on map
                  cmap='crest',            # change the ↵
                  ↪ color if you want
```

```

        fmt=".2f") # 2 decimals for
    ↪ displayed values

# Adjustment + aesthetics of plot (size and title only)
hm.figure.set_size_inches(10, 8) # this sets the
    ↪ size and we will use to be consistent across all plots below
plt.title(plot_title, y=1.1) # add a title so
    ↪ we know what we plotted

# Save our figure
plt.savefig(fig_name_hmp, dpi=300, bbox_inches='tight') # save w/figure
    ↪ name

```

13.5 Bonus: Plotting a Relationship

Do you think **acousticness** and **energy** have a relationship with each other? If you think so, what direction would you expect (positive or negative)? If you wanted to visualize this relationship, which plot would you use?

Task: Create your plot below

```
[ ]: ### YOUR CODE HERE ###
```

```

[ ]: #@title Example Solution
# Set this text for the plot- easiest to edit here as needed
x_label_name = 'Acousticness %'
y_label_name = 'Energy %'
leg_title = 'Mode'
plot_title = 'Relationship between Acousticness % and Energy %'
fig_name_scatter2 = 'Scatter_Acoustic_Energy_.png'

# Example of a scatter plot
sp2 = sns.scatterplot(
    data=spotify,
    x='acousticness_percent',
    y='energy_percent')

# Adjustment + aesthetics of plot
sp2.figure.set_size_inches(6.5, 4.5)
sns.despine()
plt.xlabel(x_label_name)
plt.ylabel(y_label_name)
plt.title(plot_title, y=1.05)
plt.ylim(0,100);

# Save our figure
plt.savefig(fig_name_scatter2, dpi=300, bbox_inches='tight') # save

```

14 Post Workshop Survey

If you could take a few minutes to complete the survey here, we would appreciate your time and feedback!

15 Extras

15.1 Broadcasting

Broadcasting involves performing a computation using two or more arrays that share **one** dimension. Perhaps you want to divide every row of array A element-by-element by array B. Here is a plausible real life example where we might want to broadcast. Many numpy functions will be able to figure it out

```
[ ]: # An array with the number of hours we worked on
# three different projects throughout the workweek.
hours_on_each_project = np.array([[2,4,2],[1,7,0],[5,2,1],[8,0,0],[3,3,2]])
billing_rate_per_project = np.array([50,60,25]) # $/hour

# Money billed to each project for each day of the workweek.
money_on_each_project = hours_on_each_project*billing_rate_per_project
print("money billed to each project:\n", money_on_each_project)
```

money billed to each project:

```
[[100 240 50]
 [ 50 420  0]
 [250 120 25]
 [400  0  0]
 [150 180 50]]
```

15.2 Aggregating Across an Axis

We may want to perform an operation across one dimension of an array (row, column, or a third/fourth dimension). Here we can use `np.apply_along_axis()` function to specify the array, function, and axis we want to apply a function on. Here is an example. Notice how `axis=0` and `axis=1` perform the function on different dimensions of the array.

```
[ ]: price_per_meal = np.array([[8,12,23], # Bfast, lunch, dinner - day 1
                               [6,12,32], # Bfast, lunch, dinner - day 2
                               [8,15,19], # Bfast, lunch, dinner - day 3...
                               [3,9,40],
                               [4,18,22]])

average_meal_cost = np.apply_along_axis(arr=price_per_meal, func1d = np.mean,
↪axis=0)
```

```
print('average_meal_cost:\n', average_meal_cost) # Avg meal cost for bfast,
↳ lunch, and dinner.

total_cost_each_day= np.apply_along_axis(arr=price_per_meal, func1d = sum,
↳ axis=1) # python comes with a native sum function.
print('\ntotal_cost_each_day:\n', total_cost_each_day) # Total meal cost per
↳ each day.
```

```
average_meal_cost:
[ 5.8 13.2 27.2]
```

```
total_cost_each_day:
[43 50 42 52 44]
```

Here is an example (from a different dataset), but I like this image to illustrate how the axes work.

What if we need the maximum inflammation for each patient over all days (as in the diagram on the left) or the average for each day (as in the diagram on the right)? As the diagram below shows, we want to perform the operation across an axis.

Image [source](#)

15.3 Functions

The “commands” or “instructions” we have been using have a name in Python: they are called **functions**.

A function is a block of code which only runs when it is called. The commands we were using that typically involved `()` were actually functions! Example: `np.random.choice()`

You can pass data, known as arguments or parameters, into a function. A function can return nothing at all. A function can also return data as a result.

15.3.1 Writing a function

Not only can you use functions included in the Python libraries you imported, or native to Python, you can also write your own functions.

Defining, or creating a function, always uses the following syntax.

```
def FunctionName(parameter_name1, parametername2, ...):
    FUNCTION BODY
    return ReturnValues
```

- **def:** This keyword tells Python that you are about to define a function. Don’t forget to also have parentheses and a colon.
- **Function name:** The name of your function that you are going to use to call it whenever you need it to run!
- **Function body:** The set of commands or operations you want your function to execute whenever it is called. Note that the function body (which can be multiple lines) and return statement need to be indented relative to the `def` line!

- **Arguments** (optional): The variable names for optional input values you give the function. When writing your function body you can reference the argument variable name to refer to the value of the corresponding input.
- **return** and Return values (optional): The keyword **return**, followed by any data or values you want your function to return when called.

Note that you will need to run the code that defines a function before you are able to call it.

15.3.2 Coding Exercise

This exercise combines what you've learned about functions, for loops and if statements. Use a for loop to go through the rows of our dataset. On every iteration of the loop, use the function defined below called `roll_the_dice()` . - If the roll of the dice is even, store the `bpm` data for the current row in `evens`. - If the roll of the dice is odd, store the `bpm` data for the current row in `odds`.

After the loop is complete, take the average of the evens, as well as the odds. Print the two means.

```
[ ]: ### RUN THIS CELL ###
def roll_the_dice():
    return np.random.choice([1,2,3,4,5,6])

[ ]: ### YOUR CODE HERE ###

[ ]: #@title Example Solution
evens = []
odds = []

for i in range(len(spotify)):
    dice_roll = roll_the_dice()
    #print('dice_roll: ', dice_roll)
    if (dice_roll==1) or (dice_roll==3) or (dice_roll==5):
        odds.append(spotify.bpm[i])
    else:
        evens.append(spotify.bpm[i])

print('Average of evens:', np.average(evens).round(2))
print('Average of odds:', np.average(odds).round(2))
```

Average of evens: 123.29

Average of odds: 121.74

15.4 Plotting with Matplotlib

Matplotlib is a graphics library built to play nicely with numpy. Matplotlib was also conceived as an alternative to using matlab's plotting tools, so those familiar with matlab may notice some syntax overlap. Nowadays, there are certainly more advanced plotting libraries than Matplotlib with various beautifying features, but Matplotlib remains popular because of its simplicity and ability to work with nearly every graphics engine.

Recall that previously, we imported matplotlib library pyplot as plt (See “Imports & Libraries”). Optionally, we can also set a default style for our matplotlib plots. Some available styles are “ggplot”, “classic”, “fivethirtyeight”, and “tableau-colorblind10”. We’ll just use the “classic” style for now, but feel free to experiment with different styles by running the following code.

```
[ ]: # Run this to set the default matplotlib style (optional)
plt.style.use('fivethirtyeight')

# Example plot
plt.figure(figsize=(3,3))
plt.plot([1,2,3,4], [2,3,4,5], '-x')
plt.show()
```

15.4.1 Plotting, Rendering, and Saving Figures

The cleanest way to save a figure is:

- (1) Initialize a figure object. If you are combining multiple plots onto one figure object you may want to initialize “ax” (axis) objects for each plot as well, so that you can have more control over the settings of each graph later.
- (2) Plot your data on your figure. Matplotlib lets you do this directly from the pyplot module (`plt.plot(x,y)`), or you can apply the plot function to the specific axes (subplot) you wish to graph to. (`ax[0].plot(x,y)`).
- (3) Save your figure to a filename.
- (4) Use `plt.show()` to render your figure.

This strategy should work in colab, jupyter notebook, or within a Python script.

Here are a two different example workflows for creating images. Feel free to look in your Colab files for the saved figure. This same workflow should work in a jupyter notebook, a Python script, or a Python command line as well.

```
[ ]: ### EXAMPLE 1 ####
print("Example 1...")
x = np.array([1,2,3,4,5,6])
y = np.array([1,4,5,6,7,19])

# Initialize figure
# Note that we can set a figsize parameter
# to specify the window size.
fig = plt.figure(figsize=(4,3))

# Simple scatter plot
plt.plot(x,y, '-x')
plt.xlabel('x')
plt.ylabel('y')

# Save figure
```

```

plt.savefig('example1.png')

# Render figure in colab window
plt.show()

print('Example 2...')

### EXAMPLE 2 ###

# Initialize figure
# This time we will initialize a figure with subplots,
# consisting of 3 graphs, each next to the other.
fig, ax = plt.subplots(figsize=(12,3), ncols=3)

# One scatter plot and one line plot.
ax[0].plot(x,y, 'x')
ax[1].plot(x,y, '-')
ax[2].plot(x,y, 'o')

for i in range(3):
    ax[i].set_xlabel('x')
    ax[i].set_ylabel('y')

# Save figure
plt.savefig('example2.png')

# Render figure in colab window
plt.show()

```

15.4.2 Multiple Lines on a graph

There are many cases where we may want to show 3 or more dimensions on a graph, using color or shape as the third dimensions. There are two strategies for doing this.

(1) Plot each line/distribution on the graph at a time, specifying the label and color for each line
or

(2) Plot a multidimensional numpy array, specifying which axis should be the x, y, color, or shape axes.

Here is an example, build a histogram of different candy costs in 50 states (randomly generated).

```

[ ]: sour_patch_kids_cost = np.random.rand(50)+.5
    hershey_bar_cost = np.random.rand(50)+.25
    reeses_cost = np.random.rand(50)+1

```

```

# We will use 10 cent increments from 0-3$
# for our histogram buckets (optional argument).
cost_bins = [i*.1 for i in range(30)]

##### Approach 1 #####
f = plt.figure(figsize=(5,4))

# The alpha parameter refers to how opaque each color is.
print("Example 1...")
plt.hist(sour_patch_kids_cost, color='green', label='sour patch', alpha=.7,
        ↪bins=cost_bins, histtype='stepfilled')
plt.hist(hershey_bar_cost, color='brown', label='hershey', alpha=.7,
        ↪bins=cost_bins, histtype='stepfilled')
plt.hist(reeses_cost, color='orange', label='reeses', alpha=.7, bins=cost_bins,
        ↪histtype='stepfilled')
plt.legend()
plt.xlabel('cost per bar')
plt.ylabel('# of stores')
plt.show()

##### Approach 2 #####
print("Example 2...")
f = plt.figure(figsize=(5,4))
all_candy_bars = np.array([sour_patch_kids_cost,
                           hershey_bar_cost,
                           reeses_cost]).transpose()
plt.hist(all_candy_bars, alpha=.8,
        color=['green', 'brown', 'orange'],
        label=['sour patch', 'hershey', 'reeses'],
        histtype='stepfilled', bins=cost_bins)
plt.xlabel('cost per bar')
plt.ylabel('# of stores')
plt.legend()
plt.show()

```

15.4.3 Beautifying your plots

A complaint many have about Matplotlib is that the plots are not very aesthetically appealing by default. However, Matplotlib offers numerous settings you can control in order to have precise control over what your plots look like. Here are a few examples of what you can control and how to go about changing these settings.

Let's start with the following plots, and experiment with changing some of the parameters.

Colors and Shapes

Typically, we pass in arguments for what we want the colors and shapes of a plot to look like, when

we call the plotting function (`.plot()`, `.hist()`, etc.). You can also set a specific color palette for Matplotlib to select colors from.

```
[ ]: f, ax = plt.subplots(ncols=2, nrows=2, figsize=(10,10))

# Some random distributions for variables.
x = np.random.random(500)
y = x + .2*np.random.randn(500)
i = np.random.randn(500)
j = np.random.randn(500)

# Plotting different types of plots.
# Sometimes if you want to edit a figure,
# it can be helpful to return an object from the plotting function
# and then you can mess around with properties of that object.
# (See plt3 for an example)
plt1 = ax[0,0].plot(x,y,
                    marker='x', linestyle='none',
                    markersize=15, color='red', label='dist1')
plt11 = ax[0,0].plot(x,i,
                     marker='.', linestyle='none',
                     markersize=8, color='blue', label='dist2')

# We can also specify our colors in hexcode.
plt2 = ax[0,1].hist(x, color='#a032a8')

# patch_artist=True lets us color in the boxes.
plt3 = ax[1,0].boxplot([x,y,i,j], patch_artist=True)
# We can reference the object output by our plotting function
# in order to change various parameters.
plt3['boxes'][0].set_color('red')
plt3['boxes'][1].set_color('orange')
plt3['boxes'][2].set_color('yellow')
plt3['boxes'][3].set_color('pink')

# See matlab documentation for different color maps
plt4 = ax[1,1].hist2d(i,j, bins=20, cmap='spring')

plt.show()
```

Plot Background

We can also edit the backgrounds of the plots - this might mean adding/removing gridlines, changing the background color, or even adding additional text or figures to the plots. Here are some examples for how we might edit the background attributes of our figures.

```
[ ]: # We can turn on/off legends, colorbars, etc.
ax[0,0].legend()

# We can add text, and shapes to plots.
ax[1,0].text(s='this is \na boxplot', x=1, y=-2, fontsize=20)

# We can turn gridlines on / off.
ax[1,1].grid(visible=True, axis='both', color='black')

# We can set the background color.
ax[0,1].set_facecolor('grey')

# Show updated figure.
f
```

Axes

We can also edit the x- and y-axes, ticks, and labels.

(Note: in python we unfortunately often refer to subplots as “axes”. These are **different** from the x- and y- axes, which each subplot contains.)

Here are a few examples for how you might want to modify your x and y axes.

```
[ ]: # We can set the labels for our axes.
ax[0,0].set_xlabel('x', fontsize=20)
ax[0,0].set_ylabel('y', fontsize=20)

# We can remove ticks, or change the scale of the ticks.
ax[0,1].set_xticks([])
ax[0,1].set_yscale('log')

# We can set different tick labels.
ax[1,0].set_xticklabels(['x', 'y', 'i', 'j'], fontsize=14)

# We can rotate the labels,
# while keeping the tick labels the same.
ax[1,1].set_xticklabels(ax[1,1].get_xticks(), rotation=45)

# Show updated figure.
f
```

16 Just for fun

Read about some of the top [trends from 2023 on Spotify](#). Do you recognize any of these names from our dataset?

17 Technical Notes and Credits

The exercises for this notebook were adapted from resources available from the [Python for Supervised Machine Learning Bootcamp](#) and Dr. Brianna Chrisman. Exercises were also adapted from [Programming in Python Data Carpentry Resources](#).

Thanks to the Data Science Initiative at UNR, supported by a Nevada INBRE supplemental award for Building Data Science Capacity, along with Research & Innovation for supporting this workshop during the first [Data Science Conference](#). Thanks also to the sponsors including AWS and IBM. Shout out and special thanks to Dr. Juli Petereit (Director of Nevada Bioinformatics Center) for her support and hard work on putting all of this together.