09-04-22
JOVIAN - Data Analysis - Introduction to Programming with Python
Lecture Video: https://youtu.be/-2kMKVtCHVg
Python (Code): *#path: jovian/data_analysis/other_files/lesson_02_while_loop_stars.py*
NOTEBOOKS: first steps with Python, Python variables, and Python branching and loops

COURSE OVERVIEW:

➜ Lessons 1 & 2 - programming in Python geared toward data analysis
➜ Lesson 3 - numerical computation with Numpy
➜ Lesson 4 - analyzing tabular data with Pandas
➜ Lesson 5 - data visualization with MatPlotLib and Seaborn
➜ Lesson 6 - exploratory data analysis with a real world data set

New things I learned about Python:

- Assigning multiple variables to multiple values at once:
  - color1, color2, color3 = "red", "green", "blue"
- Assigning multiple variables to the same value at once:
  - color4 = color5 = color6 = "magenta"
- A variable's name must start with a letter or the underscore character _. It cannot begin with a number.
  - A variable name can **only contain** lowercase (small) or uppercase (capital) letters, digits, or underscores (a-z, A-Z, 0-9, and _).
  - Variable names are case-sensitive, i.e., a_variable, A_Variable, and A_VARIABLE are all different variables.
- Floating point numbers can also be written using the scientific notation with an "e" to indicate the power of 10. → one_hundredth = 1e-2
- Booleans are automatically converted to ints when used in arithmetic operations. True is converted to 1 and False is converted to 0.
  - 5 + False = 0, 3. + True = 4.0
- Any value can be converted to a Boolean using the bool function.
- Only the following values evaluate to False (they are often called *falsy* values):
  - The value False itself
  - The integer 0
  - The float 0.0
  - The empty value None
  - The empty text ""
  - The value False itself
  - The integer 0
  - The empty dictionary { }
  - The empty set set( )
  - The empty range range(0)
- Everything else evaluates to True (a value that evaluates to True is often called a *truthy* value).
  - Ex. bool(False) returns False

    - The float 0.0
    - The empty value None
    - The empty text ""
    - The empty list [ ]
    - The empty tuple ( )
- Ex. bool(True), bool(1) return True

- **None -** The None type includes a single value None, used to indicate the absence of a value. None has the type NoneType. It is often used to declare a variable whose value may be assigned later.
- A string can be converted into a list of characters using list function.
- **Methods**: Methods are functions associated with data types and are accessed using the . notation e.g. variable_name.method() or "a string".method(). Methods are a powerful technique for associating common operations with values of specific data types.
- The .replace method replaces a part of the string with another string. It takes the portion to be replaced and the replacement text as *inputs* or *arguments*.
- The .format method combines values of other data types, e.g., integers, floats, booleans, lists, etc. with strings. You can use format to construct output messages for display.
    - output_template = """If a grocery store sells ice bags at $ {} per bag, with a profit margin of {} %, then the total profit it makes by selling {} ice bags is $ {}."""
    - total_profit = cost_of_ice_bag * profit_margin * number_of_bags
    - output_message = output_template.format(cost_of_ice_bag, profit_margin*100, number_of_bags, total_profit)
- **Lists** - ordered sequence of data
- tuple(['one', 'two', 'three']) - immutable = ('one', 'two', 'three')
- list(('Athos', 'Porthos', 'Aramis')) - mutable = ['Athos', 'Porthos', 'Aramis']
- Two ways to make a dictionary:

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

and

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

- dictionary.get() - to avoid key errors
    - person2.get(address, Unknown) - if the key 'address' does not exist in the dictionary, it will be created with the value 'Unknown'
- Person1["address"] = 1 Penny Lane - will either add it if not there or update it if it was there.
- dictionary.keys(), .values(), .items() - to get all keys, values, or key-value pairs

.....
JOVIAN - Data Analysis - Introduction to Programming with Python
Lecture Video: https://youtu.be/_a95RaIZyf0

NOTEBOOKS: branching, conditionals, loops; functions and scope; os and filesystem

BRANCHING using Conditionals Statements and Loops:

- If statements:
  - *if a_number % 2 == 0:*
    print("We're inside an if block")
    print('The given number {} is even.'.format(a_number))
- ELIF: The conditions are evaluated one by one. For the first condition that evaluates to True, the block of statements below it is executed. The remaining conditions and statements are not evaluated. So, in an if, elif, elif... chain, at most one block of statements is executed, the one corresponding to the first condition that evaluates to True.
- This is the key difference between using a chain of if, elif, elif... statements vs. a chain of if statements, where each condition is evaluated independently. Multiple ifs will all get checked, elif quits after one is true.
- Using if, elif, and else together: You can also include an else statement at the end of a chain of if, elif... statements. This code within the else block is evaluated when none of the conditions hold true.
- Non-Boolean Conditions: conditions do not necessarily have to be booleans. In fact, a condition can be any value. The value is converted into a boolean automatically using the bool operator. This means that falsy values like 0, '', {}, [], etc. evaluate to False and all other values evaluate to True.
- Shorthand if conditional expression: Python allows writing conditions in a single line of code. It is known as a ***conditional expression***, sometimes also referred to as a ***ternary operator***.

```
a_number = 13
if a_number % 2 == 0:
    parity = 'even'
else:
    parity = 'odd'
```

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

  - **Statements**: A statement is an instruction that can be executed.
  - **Expressions**: An expression is some code that evaluates to a value..
  - Most expressions can be executed as statements, but not all statements are expressions. For example, the regular if statement is not an expression since it does not evaluate to a value.
  - An expression is anything that can appear on the right side of the assignment operator =.

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

```
# if expression

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

- **BREAK statement** = use to break out of a while loop and exit completely, when you have reached a goal and do not want to run the rest of the code in the loop
- **CONTINUE statement** = any remaining statements in the loops are skipped, and the next iteration begins

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

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

- FOR loop over DICTIONARY:

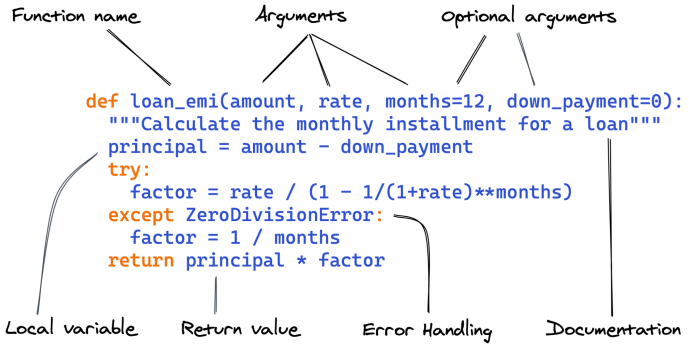
  for key in dictionary:
  print("Key:", key, ",", "Value:", dictionary[key])
  - Iterate over values only:

    for value in dictionary..values():
    print(value)
  - Iterate over keys and values:

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

.....

WRITING FUNCTIONS:



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

Optional arguments - not required, but can be set with a default value in the function definition. (Must come after required arguments in the definition.)

Named arguments - Invoking a function with many arguments can often get confusing. Python provides the option of invoking functions with named arguments for better clarity. You can also split function def into multiple lines.

- Home Loan problems:

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

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

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

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

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

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

print(f'Eight year: ${eight_year:,.2f}')

ten_year = loan_emi(1260000, 10*12, 0.08/12)

print(f'Ten year: ${ten_year:,.2f}')
```

.FORMAT()

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

```
>> Option 1 is the lower EMI: $14,568
```

TRY and EXCEPT:

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

   return emi_100k

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

print("EMI for $100,000 loan without interest (try-except) ${:,}".format
      (math.ceil(loan_emi(100000, 10 * 12, rate = 0))))

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

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

09-05-22
JOVIAN - Data Analysis - Numerical Computing with Numpy

Lecture Video: https://youtu.be/d0E0_87CrFA Office Hours: https://youtu.be/egkw8xpIs60

Python (Code): *path = '/Users/jovian/data_analysis/Other Files/lecture03.py'*

NOTEBOOKS: numerical-computing-with-numpy, 100-numpy-exercises, os-and-file system

▶ The Numpy library provides specialized data structures, functions, and other tools for numerical computing in Python.

▶ We use vectors, or lists of values, to organize inputs grouped by category, etc., as well as for weights on those inputs: weights = [w1, w2, w3]

```
REGIONS:
kanto = [73, 67, 43]
johto = [91, 88, 64]
hoenn = [87, 134, 58]
sinnoh = [102, 43, 37]
unova = [69, 96, 70]
WEIGHTS:
weights = [w1, w2, w3]
```

```
def crop_yield(region, weights):
    result = 0
    for x, w in zip(region, weights):
        result += x * w
    return result
```

```
for item in zip(kanto, weights):
    print(item)

>> (73, 0.3)  # zip creates pairs
   (67, 0.2)  # become x, w
   (43, 0.5)
```

▶ The calculation performed by the crop_yield (element-wise multiplication of two vectors and taking a sum of the results) is also called the ***dot product***. The Numpy library provides a built-in function to compute the dot product of two vectors. However, we must first convert the lists into Numpy arrays.

- kanto = np.array([73, 67, 43])
- print("Kanto: ", np.dot(kanto, weights)) *# same as (kanto * weights).sum()*

▶ Numpy arrays can be indexed like Python lists, but they also have differences from lists:

- **Ease of use**: You can write small, concise, and intuitive mathematical expressions like (kanto * weights).sum() rather than using loops & custom functions like crop_yield.
- **Performance**: Numpy operations and functions are implemented internally in C++, which makes them much faster than using Python statements & loops that are interpreted at runtime

Multi-dimensional Numpy arrays: lists of lists

```
climate_data = np.array([[73, 67, 43],
                         [91, 88, 64],
                         [87, 134, 58],
                         [102, 43, 37],
                         [69, 96, 70]])
```

- One-dimensional = a vector
- Two-dimensional = more than one row
- Three-dimensional = more lists inside those elements (number of dimensions keeps going depending on levels of nesting lists

▶ Numpy arrays can have any number of dimensions and different lengths along each dimension. We can inspect the length along each dimension using the **.shape property** of an array. (Outputs a tuple representing the structure.)

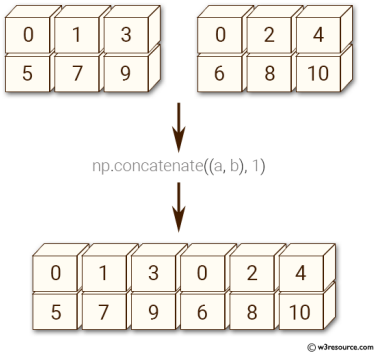
▹ ***2d arrays*** *are like two 1d arrays stacked on top each other, making a rectangle shape.*
▹ ***3d arrays*** *are like two 2d arrays stacked beside each other, making a cuboid shape.*
▹ All the elements in a numpy array have the same data type. You can check the data type of an array using the **.dtype** property.

$$\begin{bmatrix} 73 & 67 & 43 \\ 91 & 88 & 64 \\ \vdots & \vdots & \vdots \\ 69 & 96 & 70 \end{bmatrix} \times \begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \end{bmatrix}$$

▹ When these two are multiplied, we get back a vector where each value in lines of the vector on the left are multiplied by the corresponding values in the weights vector: 73*w11, 67*w12, 43*w13, etc.
▹ We can use the **np.matmul function** or the @ operator to perform matrix multiplication.

 # ..... #

WORKING WITH CSV FILES:



▶ Numpy also provides helper functions reading from & writing to files.
(Using URLLIB to import files, such as CSV, look into more.)
▶ np.genfromtxt - args = filename (or path), data separator, what to skip
    ▹ reads file into numpy array
▶ np.concatenate - add a new column (axis = 0 for rows, axis = 1 for columns)
▶ .reshape() = changes the shape of an array without changing its data.
climate_results = np.concatenate((climate_data, yields.reshape(10000, 1)), axis=1)
▶ np.savetxt - write the final results from our computation above back to a file

```
np.savetxt('climate_results.txt',
        climate_results,
        fmt='%.2f',
        delimiter=',',
        header='temperature,rainfall,humidity,yeild_apples', comments='')
```

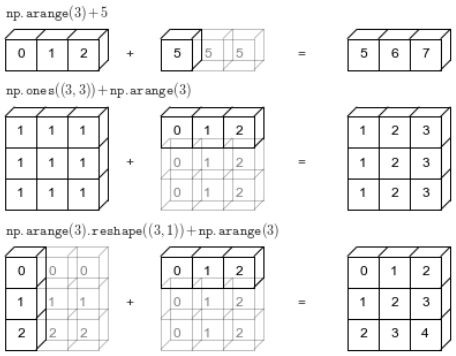
▶ Numpy provides hundreds of functions for performing operations on arrays.
▹ Here are some commonly used functions:
    ▹ Mathematics: np.sum, np.exp, np.round, arithmetic operators
    ▹ Array manipulation: np.reshape, np.stack, np.concatenate, np.split
    ▹ Linear Algebra: np.matmul, np.dot, np.transpose, np.eigvals
    ▹ Statistics: np.mean, np.median, np.std, np.max

Arithmetic operations, broadcasting and comparison

▶ Numpy arrays support arithmetic operators like +, -, *, etc. You can perform an arithmetic operation with a single number (also called scalar) or with another

array of the same shape. Operators make it easy to write mathematical expressions with multi-dimensional arrays.

- ▻ The operations are applied to all the items of the array.
- ▻ See Python file or Jupyter notebook for more.



▶ If the arrays are not the same shape, Numpy will broadcast the smaller array to match the larger array's shape. This is called broadcasting.

▶ Broadcasting only works if one of the arrays can be replicated to match the other array's shape.

▶ Array Comparison - Numpy arrays also support comparison operations like ==, !=, > etc. The result is an array of booleans.

▶ Array indexing and slicing: Numpy extends Python's list indexing notation using [ ] to multiple dimensions in an intuitive fashion. You can provide a comma-separated list of indices or ranges to select a specific element or a subarray (also called a slice) from a Numpy array.

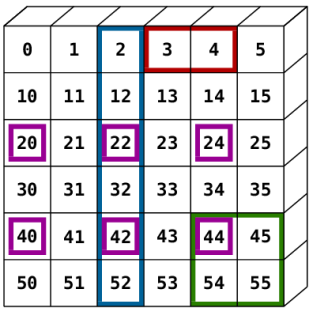
- ▻ Using whole numbers on index selection does not retain dimensions, but using ranges does.

```
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 55],
       [54, 55]])

>>> a[:, 2]
a([2, 12, 22, 32, 42, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



▶ Other ways of creating Numpy arrays:

Numpy also provides some handy functions to create arrays of desired shapes with fixed or random values.

- ▻ .zeros() = Create an array of zeros
- ▻ .ones() = Create an array of ones
- ▻ .eye() = Identity matrix - Create a square N x N identity matrix (1s on the diagonal and 0s elsewhere)
- ▻ random.random() - Create an array of the given shape and populate it with random samples from a uniform distribution over [0, 1) (every point has equal probability, FLAT)
- ▻ random.randn - Create an array of the given shape and populate it with random samples from a standard normal" distribution (BELL CURVE, varying probability, gaussian). Unlike random.random, this function takes a tuple as its argument
- ▻ .full() = fixed value - Create an array of the given shape and populate it with the given value
- ▻ .arange() = Range with a start, end, and step - Create an array of evenly spaced values within a given interval
- ▻ .linespace() = Range with a start and end - Create an array of evenly spaced values within a given interval

----

Reading from and Writing to Files using Python

Notebook: https://jovian.ai/evanmarie/python-os-and-filesystem-a0b4d

Interacting with the OS and filesystem

▶ The os module in Python provides many functions for interacting with the OS and the filesystem. Let's import it and try out some examples.

▶ We can check the **present working directory** using the os.getcwd function.

▶ To **get the list of files** in a directory, use os.listdir. You pass an absolute or relative path of a directory as the argument to the function.

▶ You can **create a new directory** using os.makedirs. Let's create a new directory called data, where we'll later download some files. os.makedirs('./data', exist_ok=True)

- ▹ verify that the directory was created and is currently empty - 'data' in os.listdir('.')

▶ **download** some files into the `data` directory using the `urllib` module.

- ▹ *from urllib.request import urlretrieve*
- ▹ *urlretrieve(url1, './data/loans1.txt')*
- ▹ *urlretrieve(url2, './data/loans2.txt')*
- ▹ *urlretrieve(url3, './data/loans3.txt')*

▶ **Reading from a file**: To read the contents of a file, we first need to open the file using the built-in `open` function. The `open` function returns a file object and provides several methods for interacting with the file's contents.

- ▹ The **open function** also accepts a `mode` argument to specifies how we can interact with the file. The following options are supported:
  - ▹ *file1 = open('./data/loans1.txt', mode='r')*
  - ▹ *print(file1_contents)*

▶ **Closing a file**: file1.close()

- ▹ Close files automatically using ***with*** - To close a file automatically after you've processed it, you can open it using the with statement.
- ▹ Once the statements within the with block are executed, the .close method on file2 is automatically invoked.

▶ **Reading a file line by line**: File objects provide a readlines method to read a file line-by-line. (use .strip() to get rid of the \n when you read lines from file.)

```
with open('./data/loans3.txt', 'r') as file3:
    file3_lines = file3.readlines()
```

▶ Processing data from files: Before performing any operations on the data stored in a file, we need to convert the file's contents from one large string into Python data types.

- Read the file line by line
- Parse the first line to get a list of the column names or headers
- Split each remaining line and convert each value into a float
- Create a dictionary for each loan using the headers as keys
- Create a list of dictionaries to keep track of all the data
-

▹ Since we will perform the same operations for multiple files, it would be useful to define a function read_csv.
▹ We'll also define some helper functions to build up the functionality step by step.

▹ Let's start by defining a function ***parse_header*** that takes a line as input and returns a list of column headers.

```
def parse_headers(header_line):
    return header_line.strip().split(',')
```

▹ Define a function ***parse_values*** that takes a line containing some data and returns a list of floats (also takes care of edge cases such as empty fields)

```
def parse_values(data_line):
    values = []
    for item in data_line.strip().split(','):
        if item == '':
            values.append(0.0)
        else:
            try:
                values.append(float(item))
            except ValueError:
                values.append(item)
    return values
```

▹ Define a function ***create_item_dict*** that takes a list of values and a list of headers as inputs and returns a dictionary with the values associated with their respective headers as keys.

```
def create_item_dict(values, headers):
    result = {}
    for value, header in zip(values,
headers):
        result[header] = value
    return result
```

What zip is doing:

```
for item in zip([1,2,3], ['a', 'b', 'c']):
    print(item)

>> (1, 'a')
   (2, 'b')
   (3, 'c')
```

▹ As expected, the values & header are combined to create a dictionary with the appropriate key-value pairs.

▹ Put it all together and define the ***read_csv*** function.

```
def read_csv(path):
    result = []
    with open(path, 'r') as f:                    # Open the file in read mode
        lines = f.readlines()                     # Get a list of lines
        headers = parse_headers(lines[0])         # Parse the header
        for data_line in lines[1:]:               # Loop over the remaining lines
            values = parse_values(data_line)      # Parse the values
            item_dict = create_item_dict(values, headers) # Create dicti w/ values & headers
            result.append(item_dict)              # Add the dictionary to the result
    return result
```

▶ <u>Writing to files</u> - create/open a file in w mode using open and write to it using the ***.write*** method. The string format method will come in handy here.

```
def write_csv(items, path):
    with open(path, 'w') as f:
        if len(items) == 0:
            return

        headers = list(items[0].keys())
        f.write(','.join(headers) + '\n')

        for item in items:
            values = []
            for header in headers:
                values.append(str(item.get(header, "")))
            f.write(','.join(values) + "\n")
```

▹ Open the file in write-mode
▹ If a line is found to have nothing in it, just return

▹ Write the headers in the first line

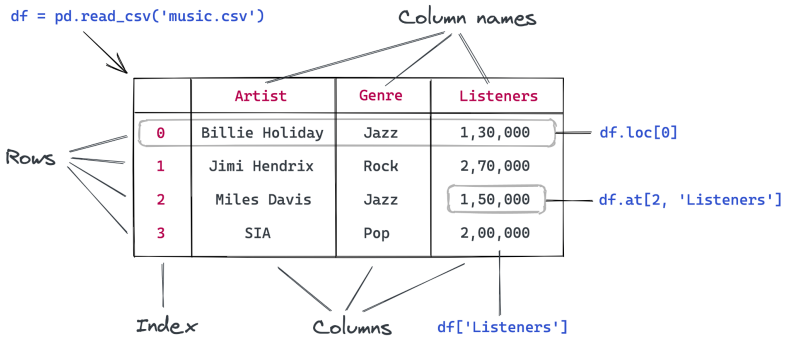
▹ Loop through data getting individual fields, writing one item per line

▹ With just four lines of code, we can now read each downloaded file, calculate the EMIs, and write the results back to new files:

```
for i in range(1,4):
    loans = read_csv('./data/loans{}.txt'.format(i))
    compute_emis(loans)
    write_csv(loans, './data/emis{}.txt'.format(i))
```

09-07-22
JOVIAN - Data Analysis - Analyzing Tabular Data with Pandas
Lecture Video: https://www.youtube.com/watch?v=MMirDY9AUEg
Lecture Notebook: https://jovian.ai/evanmarie/python-pandas-data-analysis
Python: */Users/evancarr/Code/puppy_lessons/Jovian/data_analysis/lecture_04.py*

➤ Reading a CSV file using Pandas
- Pandas is a popular Python library used for working in tabular data (similar to the data stored in a spreadsheet). Pandas provides helper functions to read data from various file formats like CSV, Excel spreadsheets, HTML tables, JSON, SQL, and more



Methods explored in my Python file include (*these are my comments*):
⇒ use urllib.request to download the dataset
⇒ urlretrieve is a function that downloads a remote file from a URL to a local file
⇒ pd.read_csv() is a function that reads a CSV file and returns a DataFrame object (covid_df = creating a dataframe)
covid_df = pd.read_csv('italy-covid-daywise.csv')
⇢ Data from the file is read and stored in a DataFrame object - one of the core data structures in Pandas for storing and working with tabular data. We typically use the _df suffix in the variable names for dataframes.
⇒ covid_df.head() - Let's look at the first few rows of the dataset
⇒ covid_df.info() - this method prints information about the dataframe, including the number of rows and columns, the column names, and the data types of the columns.
⇒ covid_df.describe() - this method prints some summary statistics for numerical columns in the dataframe.
⇒ covid_df.tail() - Let's look at the last few rows of the dataset
⇒ covid_df.columns - this attribute returns the column names of the dataframe as a list.
⇒ covid_df.shape - this attribute returns the number of rows and columns in the dataframe as a tuple.

➤ Retrieving data from a data frame: to retrieve data from this data frame, e.g., the counts of a specific day or the list of values in a particular column, it might help to understand the internal representation of data in a data frame. Conceptually, you can think of a dataframe as a dictionary of lists: keys are column names, and values are lists/arrays containing data for the respective columns.

⇒ Representing data as a list of dictionaries of data has a few benefits:

- All values in a column typically have the same type of value, so it's more efficient to store them in a single array.
- Retrieving the values for a particular row simply requires extracting the elements at a given index from each column array.
- The representation is more compact (column names are recorded only once) compared to other formats that use a dictionary for each row of data (see the example below).

⇒ To retrieve data from a data frame, we can get a list of values from a specific column using the **[ ] indexing notation**.
➪ We can retrieve a column from a dataframe using the column name as an attribute ⇒ **covid_df.date**

⇒ **covid_df.date** - this returns the date column as a Pandas Series object.

⇒ A **Pandas Series** is a one-dimensional array of values with an index.
⇒ We can retrieve a column from a dataframe using the column name as a key in a dictionary.
⇒ **covid_df['new_cases']** - this returns the date column as a Pandas Series object.

➪ Instead of using the indexing notation [ ], Pandas also allows accessing columns as properties of the dataframe using the . notation. However, this method only works for columns whose names do not contain spaces or special characters. ⇒ **covid_df.new_cases**
⇒ you can also pass a list of columns within the indexing notation [ ] to access a subset of the data frame with just the given columns, as a list.

⇒ **covid_df[['date', 'new_cases']]**

➪ Pandas also provides the **.at** method to retrieve the element at a specific row & column directly.
➪ **.at** is faster than using the indexing notation []. It's useful when you need to retrieve a specific value repeatedly.

⇒ The distinction between 0 and NaN is subtle but important. In this dataset, it represents that daily test numbers were not reported on specific dates.

⇒ We can find the first index that doesn't contain a NaN value using a column's: **covid_df.new_tests.first_valid_index()**

➪ **.loc** is a method that allows you to retrieve rows and columns by label or condition.

⇒ **covid_df.loc[0]** - returns first row of dataframe as a Series object.
⇒ **covid_df.loc[246]** - returns last row of dataframe as a Series object.
⇒ **.loc** uses index of dataframe by default - square brackets [] to specify the column names.

➪ **.iloc** is a method that allows you to retrieve rows and columns by position.

⇒ **covid_df.iloc[0]** - returns first row of dataframe as a Pandas Series object.

⇒ .iloc is similar to .loc, but uses the position instead of the label to retrieve rows and columns.

➛ Instead of using the indexing notation [], Pandas also allows accessing columns as properties of the dataframe using the . notation. However, this method only works for columns whose names do not contain spaces or special characters.

➛ you can also pass a list of columns within the indexing notation [ ] to access a subset of dataframe with the given columns, as a list.

➪ .copy() - this method returns a copy of the dataframe. This is useful when you want to modify a dataframe without changing the original dataframe.

➪ .drop() - this method drops the specified rows or columns from the dataframe.

⇒ covid_df.drop('new_cases', axis=1) - this drops the new_cases column from the dataframe.

➪ The new data frame cases_df is simply a "view" of the original data frame covid_df. Both point to the same data in the computer's memory. Changing any values inside one of them will also change the respective values in the other. Sharing data between data frames makes data manipulation in Pandas blazing fast. You needn't worry about the overhead of copying thousands or millions of rows every time you want to create a new data frame by operating on an existing one.

➪ Sometimes you might need a full copy of the data frame, in which case you can use the copy method. ⇒ covid_df_copy = covid_df.copy()

⇒ The data within covid_df_copy is completely separate from covid_df, and changing values inside one of them will not affect the other.

➪ .sample() - this method returns a random sample of items from the dataframe.

⇒ covid_df.sample(5) - this returns a random sample of 5 rows from the dataframe.

➤ Here's a summary of the functions & methods we looked at in this section:

- ★ covid_df['new_cases'] - Retrieving columns as a Series using the column name
- ★ new_cases[243] - Retrieving values from a Series using an index
- ★ covid_df.at[243, 'new_cases'] - Retrieving a single value from a data frame
- ★ covid_df.copy() - Creating a deep copy of a data frame
- ★ covid_df.loc[243] - Retrieving a row or range of rows of data from the data frame
- ★ head, tail, and sample - Retrieving multiple rows of data from the data frame
- ★ covid_df.new_tests.first_valid_index - Finding the first non-empty index in a series

----

➤ ANALYZING DATA from DATE FRAMES:

➪ .sum() - this method returns the sum of the values for the requested axis.

⇒ covid_df.new_cases.sum() - this returns the sum of the values in the new_cases column.

➪ The overall death rate is the total number of deaths divided by the total number of cases.

⇒ covid_df.new_deaths.sum() / covid_df.new_cases.sum() - this returns the overall death rate.

➤ QUERYING and SORTING ROWS:

➪ high_new_cases = covid_df.new_cases > 1000 - this returns a Pandas Series of True/False values.

⇒ rows that return True are included in the dataframe, and rows that return False are excluded.

➪ covid_df[high_new_cases] - this returns a dataframe with only the rows where new_cases > 1000.

➪ The data frame contains 72 rows, but only the first & last five rows are displayed by default brevity. We can change display options to view all the rows.

⇒ pd.set_option('display.max_rows', 72) - this sets the maximum number of rows to display to 72.

➪ We can also formulate more complex queries that involve multiple columns. As an example, let's try to determine the days when the ratio of cases reported to tests conducted is higher than the overall positive_rate.

⇒ covid_df.positive_rate > covid_df.new_cases / covid_df.total_tests - this returns a Pandas Series of True/False values.

➪ We can use this series to add a new column to the dataframe for positive rate.

⇒ covid_df['positive_rate'] = covid_df.new_cases / covid_df.total_tests - this adds a new column to the data frame.

➪ Add a column for the ratio of cases to tests:

⇒ covid_df['ratio'] = covid_df.new_cases / covid_df.new_tests - this adds a new column to the data frame.

➪ Print column names, where we now have our two new columns.

⇒ covid_df.columns - returns a list of the column names in the data frame.

➪ Now, we can remove those two newly added columns from the data frame.

⇒ covid_df.drop(['positive_rate', 'ratio'], axis=1, inplace=True) - this drops the positive_rate and ratio columns from the data frame, and updates the original data frame.

⇒ inplace=True - this is an optional parameter that specifies whether the changes should be made to the original dataframe. If inplace=True, the changes are made to the original dataframe. If inplace=False, the changes are made to a copy of the original dataframe.

➤ SORTING ROWS USING COLUMN VALUES:

➪ covid_df.sort_values('new_cases') - this sorts the rows by the values in the new_cases column.

⇒ covid_df.sort_values('new_cases', ascending=False) - this sorts the rows by the values in the new_cases column in descending order.

➪ You can chain multiple sort operations together. For example, let's sort the rows by the new_cases column in descending order, and then sort the rows by the new_deaths column in ascending order and display the first 10 rows using the head() method.

⇒ covid_df.sort_values('new_cases', ascending = True).sort_values('new_deaths').head(10)

➪ Some data came back with negative values. This will average the values for the day before and the day after and replace the negative value with the average.

⇒ with covid_df.at[172, 'new_cases'] = (covid_df.at[171, 'new_cases'] + covid_df.at[173, 'new_cases']) / 2, we are using the .at[ ] method to access the value at the specified row and column, viewable with covid_df.at[172, 'new_cases'] - this returns the value at row 172 and column new_cases.

Summary of the functions & methods we looked at in this section:

- covid_df.new_cases.sum() - Computing the sum of values in a column or series
- covid_df[covid_df.new_cases > 1000] - Querying a subset of rows satisfying the chosen criteria using boolean expressions
- df['pos_rate'] = df.new_cases/df.new_tests - Adding new columns by combining data from existing columns
- covid_df.drop('positive_rate') - Removing one plus columns from the data frame
- sort_values - Sorting the rows of a data frame using column values
- covid_df.at[172, 'new_cases'] = ... - Replacing a value within the data frame

----

➤ WORKING with DATES:

➪ covid_df.date - this returns a Pandas Series of the date column.

⇒ The data type of date is currently object, so Pandas does not know that this column is a date. We can convert it into a datetime column using the pd.to_datetime method.

⇒ covid_df['date'] = pd.to_datetime(covid_df.date) - this converts the date column to a datetime column.

⇒ We can now extract different parts of the data into separate columns, using the DatetimeIndex class

- covid_df['year'] = pd.DatetimeIndex(covid_df.date).year
- covid_df['month'] = pd.DatetimeIndex(covid_df.date).month
- covid_df['day'] = pd.DatetimeIndex(covid_df.date).day
- covid_df['weekday'] = pd.DatetimeIndex(covid_df.date).weekday

➪ Now, the data from specific months can be extracted using the month column, for example: covid_df[covid_df.month == 3] - this returns all rows where the month column is equal to 3.

➪ This can also be done by defining variables for the months, and then using those variables in the comparison, for example:

⇒ Query the rows for May: covid_df_may = covid_df[covid_df.month == 5]
⇒ Extract the subset of columns to be aggregated: covid_df_may_metrics = covid_df_may[['new_cases', 'new_deaths', 'new_tests']]
⇒ Get column-wise sum: covid_may_totals = covid_df_may_metrics.sum()
⇒ We can also combine the above operations into a single statement.
covid_df[covid_df.month == 5][['new_cases', 'new_deaths', 'new_tests']].sum()

➪ Check if the number of cases reported on Sundays is higher than the average number of cases reported every day by aggregate columns using the .mean() method

⇒ covid_df[covid_df.weekday == 6][['new_cases', 'new_deaths', 'new_tests']].mean()

➤ GROUPING and AGGREGATING:

➪ covid_df.groupby('month')[['new_cases', 'new_deaths', 'new_tests']].sum()

⇒ This groups the data by the month column and then aggregates the new_cases, new_deaths, and new_tests columns using the sum method.
⇒ The result is a new data frame that uses unique values from the column passed to groupby as the index.

➪ Grouping and aggregation is a powerful method for progressively summarizing data into smaller data frames.

➪ You can also aggregate by other measures like mean. Let's compute the average number of daily new cases, deaths, and tests for each month.

⇒ covid_df.groupby('month')[['new_cases', 'new_deaths', 'new_tests']].mean()

➪ We can use the cumsum method to compute the cumulative sum of a column as a new series. Let's add three new columns: total_cases, total_deaths, and total_tests.

➪ The following code adds the cumulative sum of the new_cases, new_deaths, and new_tests columns to the data frame as new columns:

⇒ covid_df['total_cases'] = covid_df.new_cases.cumsum()
⇒ covid_df['total_deaths'] = covid_df.new_deaths.cumsum()
⇒ covid_df['total_tests'] = covid_df.new_tests.cumsum()

➤ MERGING DATA from MULTIPLE SOURCES:

➪ The following document contains data from other countries: (see Python file)
locations_df = pd.read_csv('locations.csv')

➪ We can merge this data into our existing data frame by adding more columns. However, to merge two data frames, we need at least one common column. Let's insert a location column in the covid_df dataframe with all values set to "Italy".

⇒ covid_df['location'] = 'Italy'

➪ Now, we can merge the two data frames using the pd.merge function. The function takes two data frames as arguments and the name of the column that is common to both data frames.

⇒ merged_df = covid_df.merge(locations_df, on="location")

➪ The location data for Italy is appended to each row within covid_df. If the covid_df data frame contained data for multiple locations, then the respective country's location data would be appended for each row.

➪ We can now calculate metrics like cases per million, deaths per million, and tests per million:

⇛ merged_df['cases_per_million'] = merged_df.total_cases * 1e6 / merged_df.population
⇛ merged_df['deaths_per_million'] = merged_df.total_deaths * 1e6 / merged_df.population
⇛ merged_df['tests_per_million'] = merged_df.total_tests * 1e6 / merged_df.population

----

➤ WRITING DATA to a FILE:

➪ Before we write the data to a file, let's drop the columns that we don't need.

⇛ We can choose the columns we want in our new file by passing a list of column names to the new data frame. results_df will be the new data frame that we will write to a file.

```
result_df = merged_df[['date',
                       'new_cases',
                       'total_cases',
                       'new_deaths',
                       'total_deaths',
                       'new_tests',
                       'total_tests',
                       'cases_per_million',
                       'deaths_per_million',
                       'tests_per_million']]
```

➪ We can write the data frame to a CSV file using the .to_csv method.

⇛ The to_csv function also includes an additional column for storing the index of the dataframe by default. We pass index=None to turn off this behavior.

⇛ result_df.to_csv('results.csv', index=None)

➤ BASIC PLOTTING with PANDAS:

➪ We generally use a library like matplotlib or seaborn plot graphs , however, Pandas dataframes & series provide a handy .plot method for quick and easy plotting. Let's plot a line graph showing how the number of daily cases varies over time.

⇛.plot() is a method of the Pandas series class. It plots the values of the series on the y-axis and the index of the series on the x-axis.

⇛ result_df.new_cases.plot();

➪ While this plot shows the overall trend, it's hard to tell where the peak occurred, as there are no dates on the X-axis. We can use the date column as the index for the data frame to address this issue.

⇛ .set_index() is a method of the Pandas dataframe class. It sets the index of the dataframe to the column passed as an argument.
⇛ result_df.set_index('date', inplace=True)
⇛ Notice that the index of a data frame doesn't have to be numeric. Using the date as the index also allows us to get the data for a specific date using .loc.
⇛ **result_df.loc['2020-09-01']**

➪ We can now plot the data again, and the X-axis will show the dates.
result_df.new_cases.plot();

➪ Let's plot the new cases & new deaths per day as line graphs.
⇛ **result_df.new_cases.plot()**
⇛ **result_df.new_deaths.plot();**

➪ We can also compare the total cases vs. total deaths.
⇛ **result_df.total_cases.plot()**
⇛ **result_df.total_deaths.plot();**

➪ Let's see how the death rate and positive testing rates vary over time.
⇛ **death_rate = result_df.total_deaths / result_df.total_cases**
⇛ **death_rate.plot(title='Death Rate');**
⇛ **positive_rates = result_df.total_cases / result_df.total_tests**
⇛ **positive_rates.plot(title='Positive Rate');**

➪ Finally, let's plot some month-wise data using a bar chart to visualize the trend at a higher level.
⇛ **covid_month_df.new_cases.plot(kind='bar');**
⇛ **covid_month_df.new_tests.plot(kind='bar')**

09-08-22 LINKS: Seaborn gallery -- Matplotlib gallery -- Matplotlib tutorial

JOVIAN - Data Analysis - Visualization with MATPLOTLIB and SEABORN
Lecture Video: https://youtu.be/tuDcsAxxOR8
Lecture Notebook: https://jovian.ai/evanmarie/python-matplotlib-data-visualization-18a91
Dataviz CheatSheet: https://jovian.ai/evanmarie/dataviz-cheatsheet
Python: /Users/evancarr/Code/puppy_lessons/Jovian/data_analysis/python_lectures_assignments/lecture_05.py

NOTE: Plotting works best in Jupyter, so see the NOTEBOOK for all the info.

➤ INTRODUCTION:
➜ Data visualization is the graphic representation of data. It involves producing images that communicate relationships among the represented data to viewers. Visualizing data is an essential part of data analysis and machine learning. We'll use Python libraries Matplotlib and Seaborn to learn and apply some popular data visualization techniques. We'll use the words chart, plot, and graph interchangeably in this tutorial.

- ➜ **import matplotlib.pyplot as plt**
- ➜ **import seaborn as sns**
- ➜ **%matplotlib inline** - in Jupyter, keeps images in the frames not extra windows

➜ A LINE CHART displays information as a series of data points or markers connected by straight lines. You can customize the shape, size, color, and other aesthetic elements of the lines and markers for better visual clarity.

- ➢ Calling the plt.plot function draws the line chart as expected. It also returns a list of plots drawn [<matplotlib.lines.Line2D at 0x7ff70aa20760>], shown within the output.
- ➢ We can include a semicolon (;) at the end of the last statement in the cell to avoid showing the output and display just the graph.
  - ➻ yield_apples = [0.895, 0.91, 0.919, 0.926, 0.929, 0.931]
  - ➻ plt.plot(yield_apples);

➤ Customizing the X-axis: The X-axis of the plot currently shows list element indexes 0 to 5. The plot would be more informative if we could display the year for which we're plotting the data. We can do this by two arguments **plt.plot**.

- ➻ years = [2010, 2011, 2012, 2013, 2014, 2015]
- ➻ yield_apples = [0.895, 0.91, 0.919, 0.926, 0.929, 0.931]
- ➻ plt.plot(years, yield_apples);

➤ Axis Labels: We can add labels to the axes to show what each axis represents using the plt.xlabel and **plt.ylabel** methods.

- ➻ plt.plot(years, yield_apples)
- ➻ plt.xlabel('Year')
- ➻ plt.ylabel('Yield (tons per hectare)');

➢ **Plotting Multiple Lines:** We can add labels to the axes to show what each axis represents using the **plt.xlabel** and **plt.ylabel** methods.

```
plt.plot(years, apples)
plt.plot(years, oranges)
plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)');
```

➢ **Chart Title and Legend:** To differentiate between multiple lines, we can include a legend within the graph using the **plt.legend** function. We can also set a title for the chart using the plt.title function.

```
plt.xlabel('Year')
plt.ylabel('Yield (tons per hectare)')

plt.title("Crop Yields in Kanto")
plt.legend(['Apples', 'Oranges']);
```

➢ **Line Markers:** We can also show markers for the data points on each line using the marker argument of **plt.plot**. Matplotlib provides many different markers, like a circle, cross, square, diamond, etc. You can find the full list of marker types here: https://matplotlib.org/3.1.1/api/markers_api.html .

```
plt.plot(years, apples, marker='o')
plt.plot(years, oranges, marker='x')
```

➢ **Styling Lines and Markers** - The plt.plot function supports many arguments for styling lines and markers:

```
plt.plot(years, apples, marker='s', c='b', ls='-', lw=2, ms=8, mew=2, mec='navy')
plt.plot(years, oranges, marker='o', c='r', ls='--', lw=3, ms=10, alpha=.5)
```

- color or c: Set the color of the line (supported colors), also takes RGB hex
- linestyle or ls: Choose between a solid or dashed line
- linewidth or lw: Set the width of a line
- markersize or ms: Set the size of markers
- markeredgecolor or mec: Set the edge color for markers
- markeredgewidth or mew: Set the edge width for markers
- markerfacecolor or mfc: Set the fill color for markers
- alpha: Opacity of the plot

Check out the documentation for plt.plot to learn more:

➢ The **fmt** argument provides a shorthand for specifying the marker shape, line style, and line color. It can be provided as the third argument to plt.plot.

```
fmt = '[marker][line][color]'   # If no line style is specified in fmt, only markers appear.
plt.plot(years, apples, 's-b')  # Indicates square marker, solid line, blue color
plt.plot(years, oranges, 'o--r') # Indicates round marker, dashed line, red color
```

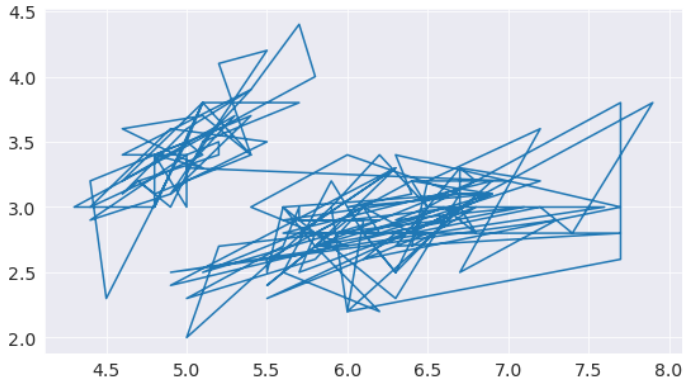
➢ Changing the Graph Size: You can use the plt.figure function to change the size of the figure. plt.figure(figsize=(12, 6)) (Give tuple for aspect ratio desired.)

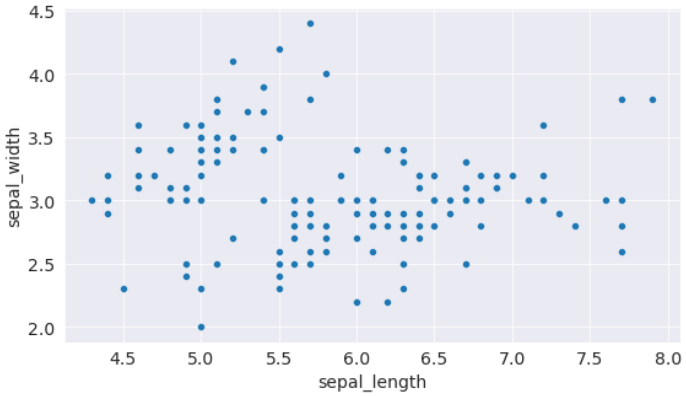
➢ Improving Default Styles using Seaborn: An easy way to make your charts look beautiful is to use some default styles from the Seaborn library. These can be applied globally using the sns.set_style function. You can see a full list of predefined styles here Ex: sns.set_style("whitegrid")

- ➢ the sns.set_style will set "whitegrid" style specifics to all plots when set
- ➢ sns.set_style("darkgrid") - nice, light gray style
- ➢ You can also edit default styles directly by modifying the matplotlib.rcParams dictionary. Learn more here.
  - matplotlib.rcParams['font.size'] = 14
  - matplotlib.rcParams['figure.figsize'] = (9, 5)
  - matplotlib.rcParams['figure.facecolor'] = '#00000000'

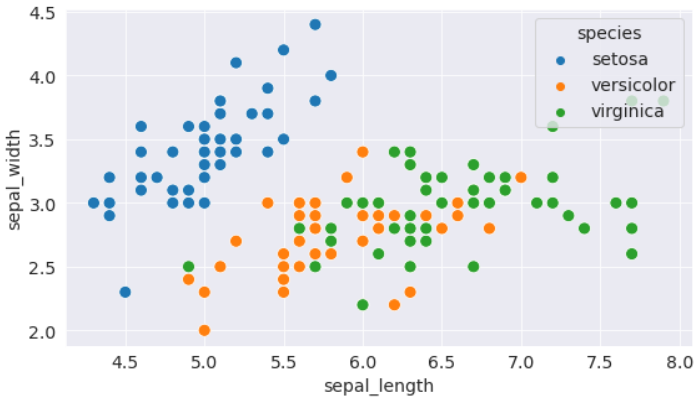
➢ SCATTER PLOT: In a scatter plot, the values of 2 variables are plotted as points on a 2-dimensional grid. Additionally, you can also use a third variable to determine the size or color of the points.

- ➢ Load data into a Pandas dataframe - flowers_df = sns.load_dataset("iris") (similar to using pd.read_csv and giving a csv file)
- ➢ plt.plot(flowers_df.sepal_length, flowers_df.sepal_width); matplotlib can take a series of columns as shown here. But this graph is messy with all the lines.
- ➢ A scatter plot will be more informative: sns.scatterplot(x=flowers_df.sepal_length, y=flowers_df.sepal_width);





- ➢ The scatter plot works much better with this info. And Seaborn's scatter plot automatically took the names of the two columns and inserted them as axis labels.
- ➢ Adding Hues: Notice how the points in the above plot seem to form distinct clusters with some outliers. We can color the dots using the flower species as a hue. We can also make the points larger using the s argument. sns.scatterplot(x=flowers_df.sepal_length, y=flowers_df.sepal_width, hue=flowers_df.species, s=100); (notice the flower species is where hue is set.)



➢ Adding **hues** makes the plot more informative. We can immediately tell that Setosa flowers have a smaller sepal length but higher sepal widths. In contrast, the opposite is true for Virginica flowers.

➢ **Customizing Seaborn Figures**: Since Seaborn uses Matplotlib's plotting functions internally, we can use functions like plt.figure and plt.title to modify the figure.

➢ In this way, matplotlib and seaborn can be used together to make completely customizable data representation

➢ This also adds a title onto the graph itself.

➳ **plt.figure(figsize=(12, 6))**

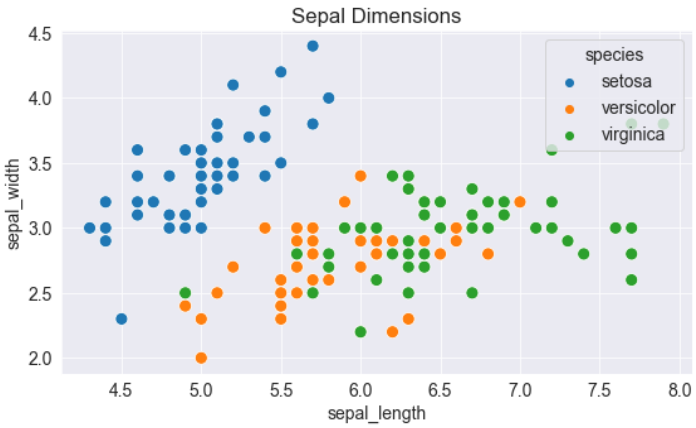
➳ **plt.title('Sepal Dimensions')**

➳ **sns.scatterplot(x=flowers_df.sepal_length,**
y=flowers_df.sepal_width,
hue=flowers_df.species,s=100);

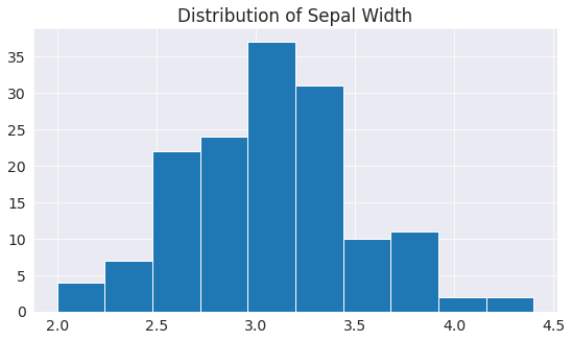
➢ **Plotting using Pandas Data Frames**: Seaborn has in-built support for Pandas data frames. Instead of passing each column as a series, you can provide column names and use the data argument to specify a data frame.

➳ **plt.title('Sepal Dimensions')**

➳ **sns.scatterplot(x='sepal_length',**
y='sepal_width',
hue='species',
s=100,
data=flowers_df);



➢ **Histogram**: represents the distribution of a variable by creating bins (intervals) along the range of values and showing vertical bars to indicate the number in each bin.
For example, let's visualize the distribution of values of sepal width in the flowers dataset. We can use the **plt.hist** function to create a histogram.



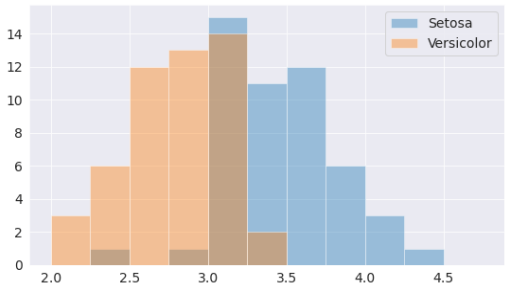
➢ Controlling the size and number of bins - We can control the number of bins or the size of each one using the bins argument.

➳ number of bins - **plt.hist(flowers_df.sepal_width, bins=5);**

➳ bin boundaries - **plt.hist(flowers_df.sepal_width, bins=np.arange(2, 5, 0.25));**

➳ bins of unequal sizes - **plt.hist(flowers_df.sepal_width, bins=[1, 3, 4, 4.5]);**

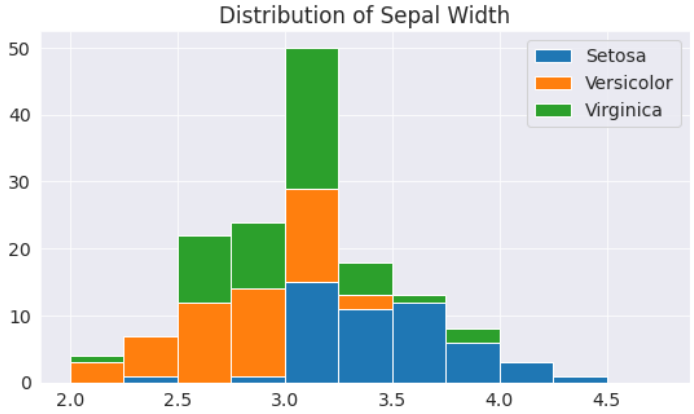
➢ **Multiple Histograms**: Similar to line charts, we can draw multiple histograms in a single chart. We can reduce each histogram's opacity so that one histogram's bars don't hide the others'.



➢ setosa_df = flowers_df[flowers_df.species == 'setosa']
➢ versicolor_df = flowers_df[flowers_df.species == 'versicolor']
➢ virginica_df = flowers_df[flowers_df.species == 'virginica']
➢ plt.hist(setosa_df.sepal_width, alpha=0.4, bins=np.arange(2, 5, 0.25));
➢ plt.hist(versicolor_df.sepal_width, alpha=0.4, bins=np.arange(2, 5, 0.25));
➢ plt.legend(['Setosa', 'Versicolor']);

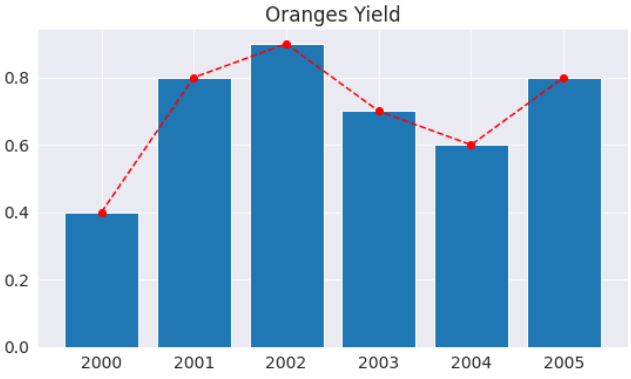
We can also stack multiple histograms on top of one another.

➢ plt.title('Distribution of Sepal Width')
➢ plt.hist([setosa_df.sepal_width,
versicolor_df.sepal_width,
virginica_df.sepal_width],
bins=np.arange(2, 5, 0.25),
stacked=True);
➢ plt.legend(['Setosa', 'Versicolor', 'Virginica']);



➢ **Bar Chart**: quite similar to line charts, i.e., they show a sequence of values. However, a bar is shown for each value, rather than points connected by lines. We can use the **plt.bar** function to draw a bar chart.

➢ Like histograms, we can stack bars on top of one another or stack with line graphs. We use the bottom argument of plt.bar to achieve this.



First graph listed will be at the back
plt.bar(years, oranges);

Will be stacked on top
plt.plot(years, oranges, 'o--r');

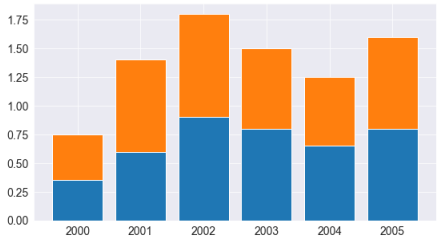
plt.title("Oranges Yield")

Like histograms, we can stack bars on top of one another. We use the **bottom** argument of **plt.bar** to achieve this.

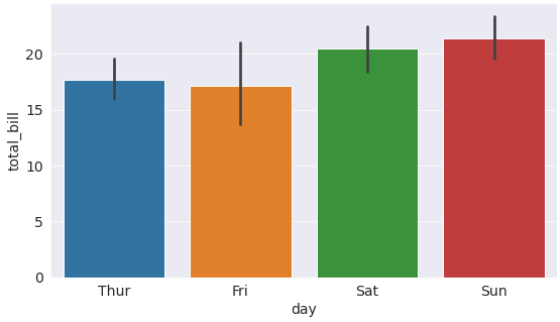
plt.bar(years, apples)

 # bottom = apples tells the graph to put apples behind oranges.

plt.bar(years, oranges, bottom=apples);



➢ **Bar Plots with Averages**: Let's look at another sample dataset included with Seaborn, called tips. The dataset contains information about the sex, time of day, total bill, and tip amount for customers visiting a restaurant over a week.

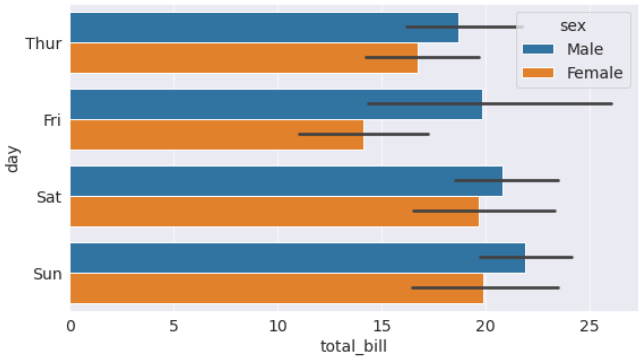


➢ The Seaborn library provides a **barplot** function which can automatically compute averages. **sns.barplot(x='day', y='total_bill', data=tips_df);**

➳ Here, we have grouped by the x axis, day, and said that we want to average the total_bill, the y axis.

➳ The lines cutting each bar represent the amount of variation in the values. For instance, it seems like the variation in the total bill is relatively high on Fridays and low on Saturday.

➳ We can also specify a hue argument to compare bar plots side-by-side based on a third feature, e.g., sex. ➳ **sns.barplot(x='day', y='total_bill', hue='sex', data=tips_df);**



➢ You can make the bars horizontal simply by switching the axes.

➳ **sns.barplot(x='total_bill', y='day', hue='sex', data=tips_df);**

➢ **Heatmap**: used to visualize 2-dimensional data like a matrix or a table using colors. (another sample dataset from Seaborn, called flights, to visualize monthly passenger traffic at an airport over 12 years.)

➢ flights_df is a matrix with one row for each month and one column for each year. The values show the number of passengers (in thousands) that visited the airport in a specific month of a year. We can use the **sns.heatmap** function to visualize the traffic at the airport.

➳ **flights_df = sns.load_dataset("flights").pivot("month", "year", "passengers")**

➳ **plt.title("No. of Passengers (1000s)")**

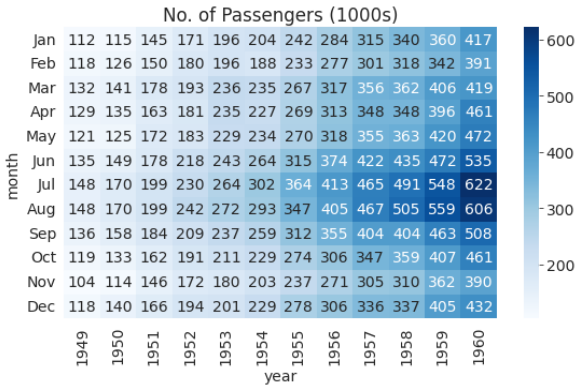
➳ **sns.heatmap(flights_df);**

Pivot is a function in Pandas that reshapes a given DataFrame organized by given index / column values. (args = y axis, moth, x axis, year, data within)

➢ The brighter colors indicate a higher traffic at the airport. By looking at the graph, we can infer two things:

- The traffic at the airport in any given year tends to be the highest around July & August.
- The traffic at the airport in any given month tends to grow year by year.

➢ We can also display the actual values in each block by specifying **annot=True** and using the **cmap** argument to change the color palette.



➳ plt.title("No. of Passengers (1000s)")

➳ sns.heatmap(flights_df, fmt="d", annot=True, cmap='Blues');

➢ **Images**: We can also use Matplotlib to display images.

➢ Before displaying an image, it has to be read into memory using the **PIL module, Python image library.**

➳ from PIL import Image

➳ img = Image.open('chart.jpg')

➢ An image loaded using PIL is simply a 3-dimensional numpy array containing pixel intensities for the red, green & blue (RGB) channels of the image. We can convert the image into an array using np.array.

➳ img_array = np.array(img)

➳ img_array.shape

➳ plt.imshow(img);

➢ We can turn off the axes & grid lines and show a title using the relevant functions.

➳ plt.grid(False)

➳ plt.title('A data science meme')

➳ plt.axis('off')

➳ plt.imshow(img);

➢ We can turn off the axes & grid lines and show a title using the relevant functions.

➳ plt.grid(False)

➳ plt.title('A data science meme')

➳ plt.axis('off')

➳ plt.imshow(img);

➢ To display a part of the image, we can simply select a slice from the numpy array.

➳ plt.grid(False)

➳ plt.axis('off')

➳ plt.imshow(img_array[125:325,105:305]);

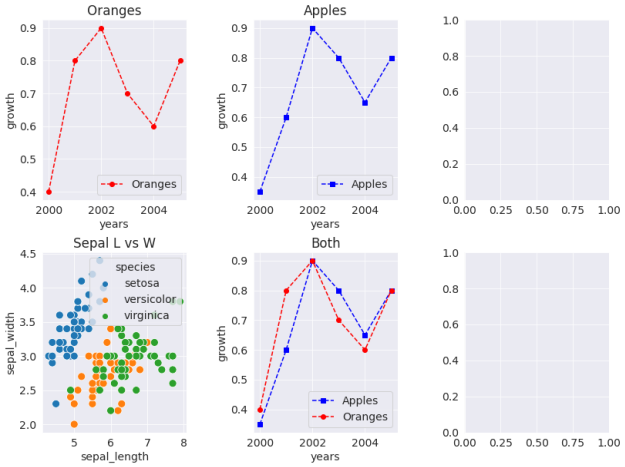
➢ **Plotting multiple charts in a grid:** Matplotlib and Seaborn also support plotting multiple charts in a grid, using **plt.subplots**, which returns a set of axes for plotting. ➳ fig, axes = plt.subplots(2, 3, figsize=(16, 8))

➢ plt.subplots(2, 3); # Create a grid of graphs 2 x 3

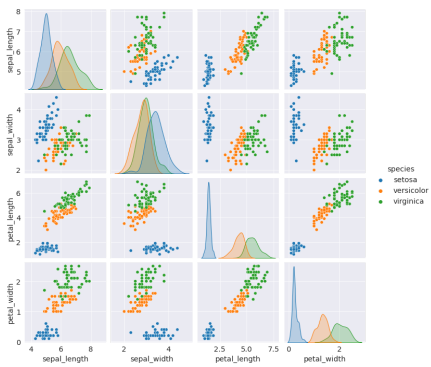
➢ plt.tight_layout(pad=2) # Makes it so the numbers don't overlap

➢ fig, axes = plt.subplots(2, 3); # Create a grid of graphs 2 x 3
➢ plt.tight_layout(pad=2) # Makes it so the numbers don't overlap
➢ axes # shows axes created in previous cell
➢ axes.shape # shows the shape of our layout is 2 grids by 3 grids
➢ axes[0,0] # points to a grid within the whole, a subgrid

➢ Designing the whole grid:
 # Create a grid of graphs 2 x 3
➢ fig, axes = plt.subplots(2, 3, figsize=(12,9));
➢ plt.tight_layout(pad=3) *# Makes it so the numbers/labels don't overlap*
➢ axes[0,0].plot(years, oranges, "o--r") *# creates the 0,0 grid*
➢ axes[0,0].set_title('Oranges') *# titles 0,0 as Oranges*
➢ axes[0,0].set_xlabel('years') *# labels the x axis*
➢ axes[0,0].set_ylabel('growth') *# labels the y axis*
➢ axes[0,0].legend(['Oranges']) *# creates a legend in the grid*
➢ axes[0,1].plot(years, apples, "s--b") *# creates the 0,1 grid*
➢ axes[0,1].set_title("Apples") *# titles 0, 1 as Apples*
➢ axes[0,1].legend(['Apples'])
➢ axes[0,1].set_xlabel('years')
➢ axes[0,1].set_ylabel('growth')
➢ axes[1,1].plot(years, apples, "s--b")
➢ axes[1,1].plot(years, oranges, "o--r")
➢ axes[1,1].set_title("Both")
➢ axes[1,1].legend(['Apples', 'Oranges'])
➢ axes[1,1].set_xlabel('years')
➢ axes[1,1].set_ylabel('growth')
➢ axes[1,0].set_title("Sepal L vs W")
➢ sns.scatterplot(x=flowers_df.sepal_length,
y=flowers_df.sepal_width,
hue=flowers_df.species,
s=100, ax=axes[1,0])



➢ <u>Pair plots</u>: Seaborn also provides a helper function <u>sns.pairplot</u> to automatically plot several different charts for pairs of features within a dataframe.



➢ sns.pairplot(flowers_df,
hue='species');
➢ sns.pairplot(tips_df, hue='sex');