MATH 1130 Companion Manual

Kelly Ramsay

2024-07-03

Table of contents

Introduction

This is a short companion manual for the course MATH 1130. It contains a brief overview of the major topics and some of the Python methods covered in each case. It is not a replacement for the cases, which are the main content of the course. It is meant to be used as a reference manual and as supplementary material to aid you in your understanding of the material. Please report any typos to kramsay2@yorku.ca .

1 Python basics

1.1 Python and Jupyter

Python is a general purpose programming language that allows for both simple and complex data analysis. Python is incredibly versatile, allowing analysts, consultants, engineers, and managers to obtain and analyze information for insightful decision-making.

The Jupyter Notebook is an open-source web application that allows for Python code development. Jupyter further allows for inline plotting and provides useful mechanisms for viewing data that make it an excellent resource for a variety of projects and disciplines.

The following section will outline how to install and begin working with Python and Juypter.

1.2 Setting up the Python Environment locally (optional)

Instruction guides for Windows and MacOS are included below. Follow the one that corresponds with your operating system.

1.2.1 Windows Install

1. Open your browser and go to https://www.anaconda.com/
2. Click on your OS and then "Download"
3. Run the downloaded file found in the downloads section from Step 2
4. Click through the install prompts
5. Go to menu (or Windows Explorer), find the Anaconda3 folder, and double-click to run OR Use a Spotlight Search and type "Navigator", select and run the Anaconda Navigator program. Note that MacOS also comes with Python pre-installed, but you should *not* use this version, which is used by the system. Anaconda will run a second installation of Python and you should ensure that you only use this second installation for all tasks.

1.2.2 Compare and contrast Jupyter, Python and Anaconda

- Jupyter Notebook is a web application that allows you to create and share documents that contain live code, equations, visualizations, and narrative text.
- Python is a programming language that is often used in scientific computing, data science, and general-purpose programming.
- Anaconda is a distribution of Python and R for scientific computing and data science. It includes the conda package manager, which makes it easy to install packages for scientific computing and data science, as well as the Jupyter Notebook and other tools.
- In simple terms Anaconda is a distribution and python is a language, Jupyter notebook is an application to create and share document that contains live code, equations, visualizations and narrative text.

1.3 File Management with Python and Jupyter

It is common practice to have a main folder where all projects will be located (e.g. "jupyter_research"). The following are guidelines you can use for Python projects to help keep your code organized and accessible:

1. Create subfolders for each Jupyter-related project
2. Group related .ipynb (the file format for Jupyter Notebook) files together in the same folder
3. Create a "Data" folder within individual project folders if using a large number of related data files

You should now be set up and ready to begin coding in Python!

1.4 Fundamentals of Python

In this case, we will introduce you to more basic Python concepts. If you prefer, you can first go through the more extensive official Python tutorial and/or the W3School Python tutorial and practice more fundamental concepts before proceeding with this case. It is highly recommended that you go through either or both of these tutorials either before or after going through this notebook to solidify and augment your understanding of Python. For a textbook introduction to Python, see this text, from which some of the following material is adapted/taken.

1.4.1 What is a program?

- A program is a sequence of instructions that specifies how to perform a computation.
- The computation might be something mathematical, such as solving a system of equations or finding the roots of a polynomial, but it can also be a symbolic computation, such as searching and replacing text in a document or something graphical, like processing an image or playing a video.
- The details look different in different languages, but a few basic instructions appear in just about every language:
  - input: Get data from the keyboard, a file, the network, or some other device.
  - output: Display data on the screen, save it in a file, send it over the network, etc.
  - math: Perform basic mathematical operations like addition and multiplication.
  - conditional execution: Check for certain conditions and run the appropriate code.
  - repetition: Perform some action repeatedly, usually with some variation.

Every program you've ever used, no matter how complicated, is made up of instructions that look pretty much like these. So you can think of programming as the process of breaking a large, complex task into smaller and smaller subtasks until the subtasks are simple enough to be performed with one of these basic instructions.

- Note that writing a program involves typing out the correct code, and then running, or executing that code:

The `print()` function allows you to print an object. Placing a python object in the round brackets and running the code makes the compute print the object. Example:

```
print("Hello World")
```

```
Hello World
```

The simplest use of Python is as a calculator. We can use the **operators** +, -, /, and * perform addition, subtraction, division and multiplication. We can use ** for exponentiation. See the following examples:

```
print(40 + 2)
print(43 - 1)
print(6 * 7)
print(2**2)
```

```
42
42
42
4
```

Note that a Python program contains one or more lines of code, which are executed in a top-down fashion.

1.4.2 Values and types

- A **value** is one of the basic things a program works with, like a letter or a number.
- Some values we have seen so far are 2, 5.0, and 'Hello, World!'.
- These **values** belong to different **types**: 2 is an integer, 5.0 is a floating-point number, and 'Hello, World!' is a string, so-called because the letters it contains are strung together.

Use `type()` to determine the type of a value. Example:

```python
print(type(2))
print(type(42.0))
print(type('Hello, World!'))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
```

If you are used to using Microsoft Excel, this is similar to how Excel distinguishes between data types such as Text, Number, Currency, Date, or Scientific. As noted above, some common data types that you will come across in Python are:

1. Integers, type `int`: `1`
2. Float type `float`: `25.5`
3. Strings, type `str`: `'Hello'`

- Here we see (1) integers and (2) floats store numeric data. The difference between the two is that floats store decimal variables (fractions), whereas the integer type can only store integer variables (whole numbers).
- Finally, (3) is the string type. Strings are used to store textual data in Python (a string of one or more characters). Later in this case we will use string variables to store country names. They are often used to store identifiers such as names of people, city names, and more.

There are other data types available in Python; however, these are the three fundamental types that you will see across almost every Python program. Always keep in mind that **every** value, or object, in Python has a type and some of these "types" can be custom-defined by the user, which is one of the benefits of Python.

1.4.3 Variables

We can assign names to objects in python so that they are easy to manipulate, a named object is called a *variable*. Use the = sign to assign a name to a variable.

- For example, if a user aims to store the integer 5 in an object named `my_int`, this can be accomplished by writing the Python statement, `my_int = 5`.
- In this case, `my_int` is a variable, much like you might find in algebra, but the `=` sign is for *assignment* not *equality* .
- So `my_int = 5` should be taken to mean something more like `Let my_int be equal to 5` rather than `my_int is equal to 5`.

```
my_int=5
print(my_int)
```

```
5
```

Here, `my_int` is an example of a *variable* because it can change value (we could later assign a different value to it) and it is of type Integer, known in Python simply as `int`. Unlike some other programming languages, Python guesses the type of the variable from the value that we assign to it, so we do not need to specify the type of the variable explicitly. For example,

1. Integers, type `int`: `my_int = 1`
2. Float type `float`: `my_float = 25.5`
3. Strings, type `str`: `my_string = 'Hello'`

Note that the names `my_int`, `my_float` and `my_string` are arbitrary here. While it is useful to name your variables so that the names contain some information about what they are used for, this code would be functionally identical to if we had used `x`, `xrtqp2` and `my_very_long_variable_name`, respectively.

```
my_int=5
print(my_int)
country="Canada"
print(country)
```

```
5
Canada
```

We mentioned before that variables can change value. Let's take a look at how this works, and also introduce a few more Python operations:

```
x = 4
print(x)
y = 2
x = y + x
print(x)
```

```
4
6
```

Again, if you're used to syntax from mathematics "x = y + x" might look very wrong at first glance. However, it simply means "throw out the existing value of x, and assign a new value which is calculated as the sum of y and x". Here, we can see that the value of x changes, demonstrating why we call them "variables".

We can also use more operators than just `+` and `-`. We use `*` for multiplication, `/` for division, and `**` for power. The standard order of operations rules from mathematics also applies and parentheses `()` can be used to override these.

1.5 Data structures:

A **data structure** is a data/value type which organizes and stores multiple values. These are more complicated data types that comprise many single pieces of data, organized in a specific way. Examples include dictionaries, arrays, lists, stacks, queues, trees, and graphs. Each type of data structure is designed to handle specific scenarios.

- As before, we use `=`, the **assignment operator**, to assign a value to the variable.

1.5.1 Dictionaries

- Python's dictionary type stores a mapping of key-value pairs that allow users to quickly access information for a particular key.
- By specifying a key, the user can return the value corresponding to the given key. Python's syntax for dictionaries uses curly braces {}:

Syntax for creation:

```
user_dictionary = {'Key1': Value1, 'Key2': Value2, 'Key3': Value3}
```

Notes:

In Python, the dictionary type has built-in methods to access the dictionary keys and values. - These methods are called by typing `.keys()` or `.values()` after the dictionary object. - We will change the return type of calling `.keys()` and `.values()` to a list by using the `list()` method. Below when we print the unconverted keys, the first thing you see is `dict_keys`, indicating the **type** of the data. Convert it to a list which is a simpler and more common data type. We can do this by passing the data into the `list(...)` function.

Example:

```python
# Creating a dictionary
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

# Accessing values
print(person["name"])  # Output: Alice
print(person["age"])   # Output: 30
print(person["city"])  # Output: New York

# Adding a new key-value pair
person["job"] = "Engineer"

# Updating an existing value
person["age"] = 31

# Deleting a key-value pair
del person["city"]

# Printing the updated dictionary
print(person)

# Getting all keys
keys = person.keys()

# Converting to list
keys_list=list(keys)
print(keys)  # Output: dict_keys(['name', 'age', 'job'])
print(keys_list) # Output: type list

# Getting all values
```

```python
values = person.values()
print(values)  # Output: dict_values(['Alice', 31, 'Engineer'])

# Converting to list
values_list=list(values)
print(values_list) # Output: type list


print(type(values_list))
print(type(list(values)))


# Creating a nested dictionary
person = {
    "name": "Alice",
    "age": 30,
    "address": {
        "city": "New York",
        "zipcode": "10001"
    }
}

# Accessing elements in a nested dictionary
print(person["address"]["city"])    # Output: New York
print(person["address"]["zipcode"]) # Output: 10001
```

```
Alice
30
New York
{'name': 'Alice', 'age': 31, 'job': 'Engineer'}
dict_keys(['name', 'age', 'job'])
['name', 'age', 'job']
dict_values(['Alice', 31, 'Engineer'])
['Alice', 31, 'Engineer']
<class 'list'>
<class 'list'>
New York
10001
```

1.5.2 Lists

A list is an incredibly useful data structure in Python that can store any number of Python objects. Lists are denoted by the use of square brackets `[]`:

Syntax:

```
user_list = [Value1, Value2, Value3]
```

Example:

```
# Creating a list
fruits = ["apple", "banana", "cherry", "date"]


# Adding a new element
fruits.append("orange")

# Updating an existing element
fruits[1] = "blueberry"

# Deleting an element
del fruits[2]

# Printing the updated list
print(fruits)

# Getting the length
print(len(fruits))


fruits = ["apple", "banana", "cherry", "date"]




# Accessing elements
print(fruits[0])  # Output: apple
print(fruits[1])  # Output: banana
print(fruits[2])  # Output: cherry

# Accessing elements by negative index
print(fruits[-1])  # Output: date
```

```python
print(fruits[-2])  # Output: cherry
print(fruits[-3])  # Output: banana
print(fruits[-4])  # Output: apple



# Accessing a range of elements
print(fruits[1:3])  # Output: ['banana', 'cherry']
print(fruits[:2])   # Output: ['apple', 'banana']
print(fruits[2:])   # Output: ['cherry', 'date']
print(fruits[:])    # Output: ['apple', 'banana', 'cherry', 'date']

# Creating a nested list
nested_list = [["apple", "banana"], ["cherry", "date"]]

# Accessing elements in a nested list
print(nested_list[0][0])  # Output: apple
print(nested_list[0][1])  # Output: banana
print(nested_list[1][0])  # Output: cherry
print(nested_list[1][1])  # Output: date
```

```
['apple', 'blueberry', 'date', 'orange']
4
apple
banana
cherry
date
cherry
banana
apple
['banana', 'cherry']
['apple', 'banana']
['cherry', 'date']
['apple', 'banana', 'cherry', 'date']
apple
banana
cherry
date
```

List computations example:

```
# Creating a list of numbers
numbers = [3.5, 1.2, 6.8, 2.4, 5.1]

# Finding the minimum value
min_value = min(numbers)
print("Minimum value:", min_value)  # Output: Minimum value: 1.2

# Finding the maximum value
max_value = max(numbers)
print("Maximum value:", max_value)  # Output: Maximum value: 6.8

# Summing all elements in the list
total_sum = sum(numbers)
print("Sum of all elements:", total_sum)  # Output: Sum of all elements: 19.0

# Rounding each element to the nearest integer
rounded_numbers = [round(num) for num in numbers]
print("Rounded numbers:", rounded_numbers)  # Output: Rounded numbers: [4, 1, 7, 2, 5]
```

```
Minimum value: 1.2
Maximum value: 6.8
Sum of all elements: 19.0
Rounded numbers: [4, 1, 7, 2, 5]
```

1.5.3 The `in` operator

The `in` operator in Python is used to check for the presence of an element within a collection, such as a list, tuple, set, or dictionary. When used with lists or other sequences, it checks if a specific value is contained in the sequence and returns `True` if it is, and `False` otherwise. When used with dictionaries, the `in` operator checks for the presence of a specified key. If the key exists in the dictionary, it returns `True`; otherwise, it returns `False`. This operator provides a simple and readable way to perform membership tests in various data structures.

Example:

```
# Creating a dictionary
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}
```

```python
# Using the 'in' operator to check for a key
print("name" in person)  # Output: True
print("job" in person)   # Output: False

# Using the 'in' operator to check for a value
print("Alice" in person.values())  # Output: True
print("Engineer" in person.values())  # Output: False


# Creating a list
fruits = ["apple", "banana", "cherry"]

# Using the 'in' operator to check for an element
print("banana" in fruits)  # Output: True
print("orange" in fruits)  # Output: False
```

```
True
False
True
False
True
False
```

1.5.4 Comments and debugging

- Comments are lines of code which begin with the # symbol. Nothing happens when you run these lines. Their purpose is to describe the code you have written, especially if it would be unclear to someone else reading it. You should be commenting your code as you go along. For example "these lines compute the average" or "These lines remove missing data" etc.
- We have seen that programs may have errors or bugs. The process of resolving bugs is known as debugging. Debugging can be a very frustrating activity. Please be prepared for this. Following, this Python textbook, there are three kinds of errors you may encouter:
  - Syntax error: "Syntax" refers to the rules of the language. If there is a syntax error anywhere in your program, Python displays an error message and quits.
  - Runtime error: The second type of error is a runtime error, so called because the error does not appear until after the program has started running. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened.

– Semantic error: If there is a semantic error in your program, it will run without generating error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing. Try runnning your code one line at a time and ensuring each line is doing what you expect, using the `print` feature.

1.5.5 For loops

One control flow element in Python is the for loop.
- The `for` loop allows one to execute the same statements over and over again (i.e. looping). - This saves a significant amount of time coding repetitive tasks and aids in code readability.

Syntax:

```
for iterator_variable in some_sequence:
    statements(s)
```

The `for` loop iterates over `some_sequence` and performs `statements(s)` at each iteration. - That is, at each iteration the `iterator_variable` is updated to the next value in `some_sequence`. - As a concrete example, consider the loop:

Example:

```
for i in [1,2,3,4]:
    print(i*i)
```

```
1
4
9
16
```

- Here, the `for` loop will print to the screen four times; that is it will print `1` on the first iteration of the loop, `4` on the second iteration, `9` on the third, and `16` on the fourth.
- Hence, the `for` loop statement will iterate over all the elements of the list `[1,2,3,4]`, and at each iteration it updates the iterator variable `i` to the next value in the list `[1,2,3,4]`.
- In `for` loops, it is an extremely good habit to choose an iterator variable that provides context rather than a random letter.
- In this case, we will use both to get you accustomed to both.
- This is because you will see both throughout the course of your data science career, but we encourage you to not use a generic name like `i` whenever possible for ease of communication.

1.5.6 List comprehensions

A list comprehension is a concise way to create a new list by applying an expression to each element of an existing list while optionally filtering elements based on a condition. It combines loops and conditional statements into a single line of code, making it efficient and readable for creating lists with specific transformations or filters. It can be used to replace a for loop with shorter code.

Example:

```python
# Using a for loop
numbers = [1, 2, 3, 4, 5]
squared_numbers = []
for num in numbers:
    squared_numbers.append(num ** 2)
print("Squared numbers (using for loop):", squared_numbers)

# Using list comprehension
squared_numbers = [num ** 2 for num in numbers]
print("Squared numbers (using list comprehension):", squared_numbers)


# Using a for loop
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = []
for num in numbers:
    if num % 2 == 0:
        even_numbers.append(num)
print("Even numbers (using for loop):", even_numbers)

# Using list comprehension
even_numbers = [num for num in numbers if num % 2 == 0]
print("Even numbers (using list comprehension):", even_numbers)
```

```
Squared numbers (using for loop): [1, 4, 9, 16, 25]
Squared numbers (using list comprehension): [1, 4, 9, 16, 25]
Even numbers (using for loop): [2, 4, 6, 8, 10]
Even numbers (using list comprehension): [2, 4, 6, 8, 10]
```

1.5.7 If statements and booleans

A boolean is a data type that represents one of two possible states: `True` or `False`. Booleans are used extensively for making decisions and comparisons. They are often the result of logical operations, such as comparisons (e.g., greater than, less than) or boolean operations (e.g., and, or, not).

Here's a brief summary:

True: Represents a condition that is considered true or valid. False: Represents a condition that is considered false or invalid. Booleans are crucial for controlling the flow of programs, making decisions, and executing code based on specific conditions being met or not.

If statements are conditional statements that allow you to execute certain blocks of code based on specified conditions. They form the foundation of decision-making in code, enabling programs to make choices and take different actions depending on whether certain conditions are True or False.

Syntax:

```
if test_expression_1:
    block1_statement(s)
elif test_expression_2:
    block2_statement2(s)
else:
    block3_statement(s)
```

Example:

```
# Example 0: A boolean
x=2
y=7
print(True)
print(x==y)
print(x > 5)



# Example 1: Simple if statement
x = 10

if x > 5:
    print("x is greater than 5")

# Example 2: if-else statement
```

```python
y = 3

if y % 2 == 0:
    print("y is even")
else:
    print("y is odd")

# Example 3: if-elif-else statement
grade = 75

if grade >= 90:
    print("Grade is A")
elif grade >= 80:
    print("Grade is B")
elif grade >= 70:
    print("Grade is C")
elif grade >= 60:
    print("Grade is D")
else:
    print("Grade is F")
```

```
True
False
False
x is greater than 5
y is odd
Grade is C
```

Explanation:

Example 0: Creates different types of boolean variables.

Example 1: Checks if `x` is greater than 5. If true, it prints "x is greater than 5".

Example 2: Checks if `y` is even (remainder of division by 2 is zero). If true, it prints "y is even"; otherwise, it prints "y is odd".

Example 3: Evaluates the value of grade and prints a corresponding grade based on the ranges specified using if, elif (else if), and else statements. This demonstrates chaining conditions to determine a grade based on numerical thresholds.

1.5.8 String formatting

String formatting refers to the various techniques used to insert dynamic values into strings in a controlled and formatted manner. We cover `.format` and `f-strings`.

f-strings (Formatted String Literals):

Syntax:

```
f"some text {expression1} more text {expression2} ..."
```

- f prefix before the string indicates it's an f-string.
- {expression} inside curly braces {} evaluates expression and inserts its value into the string.
- You can directly embed Python expressions, variables, or function calls inside {}.

.format() Method:

Syntax:

```
"some text {} more text {}".format(value1, value2)
```

- {} acts as placeholders in the string.
- .format() method is called on a string object, and values passed to it replace corresponding {} in the string.
- You can specify the order of substitution using {0}, {1}, etc., or use named placeholders {name}, {age}.

Differences: - f-strings are more concise and readable. - .format() method offers more flexibility, such as specifying formatting options or reusing values.

Examples:

```
# Example using .format() method
name = "Bob"
age = 25
formatted_string = "Hello, {}! You are {} years old.".format(name, age)
print(formatted_string)


# Example using f-strings
name = "Charlie"
age = 35
formatted_string = f"Hello, {name}! You are {age} years old."
print(formatted_string)
```

```
Hello, Bob! You are 25 years old.
Hello, Charlie! You are 35 years old.
```

2 Data extraction and transformation

This case covers the `pandas` and `numpy` libraries within Python. It also covers descriptive statistics.

2.1 Installing and importing packages

External libraries (a.k.a. packages) are code bases that contain a variety of pre-written functions and tools. This allows you to perform a variety of complex tasks in Python without having to "reinvent the wheel", i.e., build everything from the ground up. We will use two core packages: `pandas` and `numpy`.

`pandas` is an external library that provides functionality for data analysis. Pandas specifically offers a variety of data structures and data manipulation methods that allow you to perform complex tasks with simple, one-line commands.

`numpy` is a external library that offers numerous mathematical operations. We will use `numpy` later in the case. Together, pandas and numpy allow you to create a data science workflow within Python. `numpy` is in many ways foundational to `pandas`, providing vectorized operations, while `pandas` provides higher level abstractions built on top of `numpy`.

Before you use a module/package/library, it must be installed. Note that you only need to install each module/package/library once per machine. The syntax for installing a module/package/library on your machine will be either:

```
!pip install package name
```

or

```
!conda install package name
```

For example, you can run one of the following commands in a code cell to install the package `pandas`.

```
# If your machine uses pip
!pip install pandas
# If your machine uses Anaconda
!conda install pandas
```

Before using a package in each notebook or session, it must be imported. Unlike installation, importing must be done every time you use python. Let's import both packages using the `import` keyword. We will rename `pandas` to `pd` and `numpy` to `np` using the `as` keyword. This allows us to use the short name abbreviation when we want to reference any function that is inside either package. The abbreviations we chose are standard across the data science industry and should be followed unless there is a very good reason not to.

```
# Import the Pandas package
import pandas as pd

# Import the NumPy package
import numpy as np
```

2.2 Fundamentals of `pandas`

2.2.1 Series and DataFrame value types

`pandas` is a Python library that facilitates a wide range of data analysis and manipulation. Before, you saw basic data structures in Python such as lists and dictionaries. While you can build a basic data table (similar to an Excel spreadsheet) using nested lists in Python, they get quite difficult to work with. By contrast, in `pandas` the table data structure, known as the `DataFrame`, is a first-class citizen. It allows us to easily manipulate data by thinking of data in terms of rows and columns.

If you've ever used or heard of R or SQL before, `pandas` brings some functionality from each of these to Python, allowing you to structure and filter data more efficiently than pure Python. This efficiency is seen in two distinct ways:

- Scripts written using `pandas` will often run faster than scripts written in pure Python
- Scripts written using `pandas` will often contain far fewer lines of code than the equivalent script written in pure Python.

At the core of the `pandas` library are two fundamental data structures/objects: 1. Series 2. DataFrame

A `Series` object stores single-column data along with an **index**. An index is just a way of "numbering" the `Series` object. For example, in this case study, the indices will be dates, while the single-column data may be stock prices or daily trading volume.

A `DataFrame` object is a two-dimensional tabular data structure with labeled axes. It is conceptually helpful to think of a DataFrame object as a collection of Series objects. Namely, think of each column in a DataFrame as a single Series object, where each of these Series objects shares a common index - the index of the DataFrame object.

Below is the syntax for creating a Series object, followed by the syntax for creating a DataFrame object. Note that DataFrame objects can also have a single-column – think of this as a DataFrame consisting of a single Series object:

- Series: A one-dimensional labeled array capable of holding data of any type (integer, string, float, etc.). Created using `pd.Series(data, index=index)`, where data can be a list, dictionary, or scalar value.
- DataFrame: A two-dimensional labeled data structure with columns of potentially different types. Created using pd.`DataFrame(data, index=index, columns=columns)`, where data can be a dictionary of lists, list of dictionaries, or 2D array-like object.

Example:

```
# Create a simple Series object
simple_series = pd.Series(
    index=[0, 1, 2, 3], name="Volume", data=[1000, 2600, 1524, 98000]
)
simple_series


# Create a simple DataFrame object
simple_df = pd.DataFrame(
    index=[0, 1, 2, 3], columns=["Volume"], data=[1000, 2600, 1524, 98000]
)
simple_df
```

	Volume
0	1000
1	2600
2	1524
3	98000

DataFrame objects are more general than Series objects, and one DataFrame can hold many Series objects, each as a different column. Let's create a two-column DataFrame object:

```python
# Create another DataFrame object
another_df = pd.DataFrame(
    index=[0, 1, 2, 3],
    columns=["Date", "Volume"],
    data=[[20190101, 1000], [20190102, 2600], [20190103, 1524], [20190104, 98000]]
)
another_df
```

	Date	Volume
0	20190101	1000
1	20190102	2600
2	20190103	1524
3	20190104	98000

Notice how a list of lists was used to specify the data in the `another_df` DataFrame. Each element of the outer list corresponds to a row in the DataFrame, so the outer list has 4 elements because there are 4 indices. Each element of the each inner list has 2 elements because the DataFrame has two columns.

2.2.2 Reading in data

`pandas` allows easy loading of CSV files through the use of the method pd.read_csv().

Syntax:

```python
df = pd.read_csv(File name with path as a string)
```

Before loading the CSV file, you need to specify its location on your computer. The file path is the address that tells Python where to find the file. You cna use one of the following ways to specify the location

- Absolute Path: This is the complete path to the file starting from the root directory (e.g., C:/Users/username/Documents/data.csv or /Users/YourUsername/Documents/data.csv).
- Relative Path: This is the path relative to the current working directory where your Python script or Jupyter notebook is located (e.g., data/data.csv). If your CSV file is in the same directory as your Python script or Jupyter notebook, you can just provide the file name.

Examples:

```
# Load a CSV file as a DataFrame and assign to df
# Same folder: Here D.csv is in the same folder as my notebook
df = pd.read_csv("D.csv")

# Relative path: Here D.csv is in a folder called data, and the folder data is in the same fo
df = pd.read_csv("data/D.csv")

# Absolute path: Here the full path starting from my hardrive, C:, to the file D.csv is state
# On mac, it will look like: /Users/YourUsername/Documents/data.csv
df = pd.read_csv("C:\Users\OneDrive - York University\Teaching\Courses\Math1130\D.csv")
```

To find out which folder your relative path starts from, use the command `getcwd()` from the `os` module.

```
import os

# Get the current working directory
current_directory = os.getcwd()

# Print the current working directory
print("Current Directory:", current_directory)
```

2.2.3 Basic commands for DataFrames

There are several common methods and attributes that allow one to take a peek at the data and get a sense of it:

1. `DataFrame.head()` -> returns the column names and first 5 rows by default
2. `DataFrame.tail()` -> returns the column names and last 5 rows by default
3. `DataFrame.shape` -> returns (num_rows, num_columns)
4. `DataFrame.columns` -> returns index of columns
5. `DataFrame.index` -> returns index of rows

In your spare time please check the pandas documentation and explore the parameters of these methods as well as other methods. Familiarity with this library will dramatically improve your productivity as a data scientist.

Using `df.head()` and `df.tail()` we can take a look at the data contents. Unless specified otherwise, Series and DataFrame objects have indices starting at 0 and increase monotonically upward along the integers.

Example:

```
# Example DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 35, 40, 45],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Miami']
}

df = pd.DataFrame(data)

# 1. DataFrame.head()
print("DataFrame.head():")
print(df.head())
print()  # Blank line for separation

# 2. DataFrame.tail()
print("DataFrame.tail():")
print(df.tail())
print()  # Blank line for separation

# 3. DataFrame.shape
print("DataFrame.shape:")
print(df.shape)  # Output: (5, 3) - 5 rows, 3 columns
print()  # Blank line for separation

# 4. DataFrame.columns
print("DataFrame.columns:")
print(df.columns)  # Output: Index(['Name', 'Age', 'City'], dtype='object')
print()  # Blank line for separation

# 5. DataFrame.index
print("DataFrame.index:")
print(df.index)  # Output: RangeIndex(start=0, stop=5, step=1)
print()  # Blank line for separation

# Attributes
# 1. shape attribute
print("df.shape attribute:", df.shape)  # Output: (5, 3) - 5 rows, 3 columns

# 2. columns attribute
print("df.columns attribute:", df.columns)  # Output: Index(['Name', 'Age', 'City'], dtype='

# 3. index attribute
```

```
print("df.index attribute:", df.index)  # Output: RangeIndex(start=0, stop=5, step=1)
```

```
DataFrame.head():
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
3    David   40      Houston
4      Eve   45        Miami

DataFrame.tail():
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
3    David   40      Houston
4      Eve   45        Miami

DataFrame.shape:
(5, 3)

DataFrame.columns:
Index(['Name', 'Age', 'City'], dtype='object')

DataFrame.index:
RangeIndex(start=0, stop=5, step=1)

df.shape attribute: (5, 3)
df.columns attribute: Index(['Name', 'Age', 'City'], dtype='object')
df.index attribute: RangeIndex(start=0, stop=5, step=1)
```

2.2.4 Creating new columns and variables

We can create new columns by adding new columns to the DataFrame or creating a new column based on existing columns:

```
# Example DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 35, 40, 45],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Miami']
}

df = pd.DataFrame(data)

# 1. Adding new columns to the DataFrame
df['Gender'] = ['Female', 'Male', 'Male', 'Male', 'Female']
df['Salary'] = [50000, 60000, 75000, 80000, 70000]

print("DataFrame with new columns:")
print(df)
print()  # Blank line for separation

# 2. Creating a new column based on existing ones
df['Age_Squared'] = df['Age']**df['Age']

print("DataFrame with new 'Age_Squared' column:")
print(df)

# 3. We can also create columns based on multiple, other columns
df['Salary_over_Age'] = df['Salary']/df['Age']

print("DataFrame with new 'Salary_over_Age' column:")
print(df)
```

```
DataFrame with new columns:
      Name  Age         City  Gender  Salary
0    Alice   25     New York  Female   50000
1      Bob   30  Los Angeles    Male   60000
2  Charlie   35      Chicago    Male   75000
3    David   40      Houston    Male   80000
4      Eve   45        Miami  Female   70000

DataFrame with new 'Age_Squared' column:
      Name  Age         City  Gender  Salary          Age_Squared
0    Alice   25     New York  Female   50000 -6776596920136667815
1      Bob   30  Los Angeles    Male   60000  2565992168703393792
2  Charlie   35      Chicago    Male   75000  8407224849895527163
3    David   40      Houston    Male   80000                    0
4      Eve   45        Miami  Female   70000  2604998672350111773
DataFrame with new 'Salary_over_Age' column:
      Name  Age         City  Gender  Salary          Age_Squared  \
0    Alice   25     New York  Female   50000 -6776596920136667815
```

```
1      Bob   30  Los Angeles    Male   60000  2565992168703393792
2  Charlie   35      Chicago    Male   75000  8407224849895527163
3    David   40      Houston    Male   80000                    0
4      Eve   45        Miami  Female   70000  2604998672350111773

   Salary_over_Age
0      2000.000000
1      2000.000000
2      2142.857143
3      2000.000000
4      1555.555556
```

Here we see the power of `pandas`. We can simply perform mathematical operations on columns of DataFrames just as if the DataFrames were single variables themselves.

2.3 Distributions and summary statistics

A common first step in data analysis is to learn about the characteristics or **distribution** of each of the relevant columns.

2.3.1 Summary statistics

Summary statistics are numerical measures that describe important aspects of a column in a dataset. They provide a concise overview of the data's characteristics without needing to examine each individual value.

- **Examples of Summary Statistics**:
  - **Mean**: The average value of all data points.
  - **Median**: The middle value in a sorted list of numbers.
  - **Mode**: The most frequently occurring value.
  - **Max** and **Minimum**: The maximum and minimum values in a column.
  - **Range**: The difference between the maximum and minimum values.
  - **Standard Deviation**: A measure of the amount of variation or dispersion in a set of values. The standard deviation is the square root of the average of the squared distances between the data points and the the mean of the column.
  - **Percentiles**: Values below which a given percentage of observations fall.

For now, we can think of the **distribution** of a data column as a description of various aspects of that column. The distribution can be described through summary statistics, or as we will see later, through plots.

2.3.2 Standard deviation:

Standard Deviation is a measure of how spread out or dispersed the values in a dataset are from the mean (average) of the dataset. It tells you how much the individual data points typically differ from the mean value.

1. **Small Standard Deviation**:
   - **What it means**: Most of the data points are close to the mean.
   - **Example**: If the standard deviation of test scores in a class is small, it means most students scored close to the average score. There is less variability in scores.
2. **Large Standard Deviation**:
   - **What it means**: The data points are spread out over a wider range of values.
   - **Example**: If the standard deviation of test scores in a class is large, it means students' scores vary widely from the average. Some students scored much higher or lower than the average.
3. **Zero Standard Deviation**:
   - **What it means**: All data points are exactly the same.
   - **Example**: If every student in a class scored the same on a test, the standard deviation would be zero, indicating no variability.

Imagine you have two sets of data representing the ages of two different groups of people.

- **Group 1**: Ages are [25, 26, 25, 24, 25].
  - The mean age is 25.
  - The standard deviation is small because all ages are very close to the mean.
- **Group 2**: Ages are [20, 30, 25, 40, 10].
  - The mean age is also 25.
  - The standard deviation is large because the ages are spread out over a wide range (from 10 to 40).

In summary, the standard deviation helps you understand the variability of your data. A smaller standard deviation indicates data points are close to the mean, while a larger standard deviation indicates data points are more spread out. This information is useful for comparing datasets, understanding data consistency, and identifying outliers.

2.3.3 Summary statistics example:

```python
# Example DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 30, 40, 35],
    'Salary': [50000, 60000, 75000, 80000, 70000]
}

df = pd.DataFrame(data)


# Mean: The average value of all data points
mean_value = df['Age'].mean()
print(f"Mean: {mean_value}")

# Median: The middle value in a sorted list of numbers
median_value = df['Age'].median()
print(f"Median: {median_value}")

# Mode: The most frequently occurring value
mode_value = df['Age'].mode()
print(f"Mode: {mode_value.values}")

# Max and Minimum: The maximum and minimum values in a column
max_value = df['Age'].max()
min_value = df['Age'].min()
print(f"Max: {max_value}")
print(f"Min: {min_value}")

# Range: The difference between the maximum and minimum values
range_value = max_value - min_value
print(f"Range: {range_value}")

# Standard Deviation: A measure of the amount of variation or dispersion in a set of values
std_dev = df['Age'].std()
print(f"Standard Deviation: {std_dev}")

# Percentiles: Values below which a given percentage of observations fall
percentile_25 = df['Age'].quantile(0.25)
percentile_50 = df['Age'].quantile(0.50)
percentile_75 = df['Age'].quantile(0.75)
print(f"25th Percentile: {percentile_25}")
print(f"50th Percentile: {percentile_50}")
print(f"75th Percentile: {percentile_75}")


# Describe:
df['Name'].describe()
df['Age'].describe()
df.describe()
```

```
Mean: 32.0
Median: 30.0
Mode: [30]
Max: 40
Min: 25
Range: 15
Standard Deviation: 5.70087712549569
25th Percentile: 30.0
50th Percentile: 30.0
75th Percentile: 35.0
```

	Age	Salary
count	5.000000	5.000000
mean	32.000000	67000.000000
std	5.700877	12041.594579
min	25.000000	50000.000000
25%	30.000000	60000.000000
50%	30.000000	70000.000000
75%	35.000000	75000.000000
max	40.000000	80000.000000

In addition to describe, there is a value_counts() method for checking the frequency of elements in categorical data. When applied to a DataFrame class, `value_counts()` will return the frequency of each row in the DataFrame. In other words, for each unique row it returns how many instances of that row are in the DataFrame. When applied to a Series class `value_counts()` will return the frequency of each unique value in the given Series class:

```
dict_data = {
    "numbers": [1, 2, 3, 4, 5, 6, 7, 8,1],
    "color": ["red", "red", "red", "blue", "blue", "green", "blue", "green","red"],
}
category_df = pd.DataFrame(data=dict_data)
```

```
category_df

#Gives the frquency of each unique row in the DataFrame
category_df.value_counts()

#Gives the frquency of each unique value in the Series
category_df['color'].value_counts()
```

```
color
red      4
blue     3
green    2
Name: count, dtype: int64
```

2.4 More on pandas

2.4.1 Aggregating DataFrames

One way to combined multiple DataFrames is through the use of the pd.concat() method from `pandas`. We can input a list of DataFrames into `pd.concat()` that we'd like to concatenate. The `pd.concat()` function is used to concatenate (combine) two or more DataFrames or Series along a particular axis (rows or columns).

Examples:

```
# 1

# Create two example DataFrames
data1 = {
    'Name': ['Alice', 'Bob'],
    'Age': [25, 30],
    'City': ['New York', 'Los Angeles']
}
df1 = pd.DataFrame(data1)

data2 = {
    'Name': ['Charlie', 'David'],
    'Age': [35, 40],
    'City': ['Chicago', 'Houston']
}
```

```
df2 = pd.DataFrame(data2)

# Concatenate the two DataFrames
result = pd.concat([df1, df2], ignore_index=True)

print("Concatenated DataFrame:")
print(result)
```

```
Concatenated DataFrame:
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
3    David   40      Houston
```

Explanation: Two DataFrames (df1 and df2) are created with identical columns. The pd.concat([df1, df2]) function concatenates df1 and df2 along the rows (default behavior). The `ignore_index=True` argument reindexes the resulting DataFrame to have a continuous index.

```
# 2

# Create two example DataFrames
data1 = {
    'Name': ['Alice', 'Bob'],
    'Age': [25, 30]
}
df1 = pd.DataFrame(data1)

data2 = {
    'City': ['New York', 'Los Angeles'],
    'Salary': [50000, 60000]
}
df2 = pd.DataFrame(data2)

# Concatenate the two DataFrames along columns
result = pd.concat([df1, df2], axis=1)

print("Concatenated DataFrame along columns:")
print(result)
```

```
Concatenated DataFrame along columns:
    Name  Age         City  Salary
0  Alice   25     New York   50000
1    Bob   30  Los Angeles   60000
```

Explanation: `pd.concat([df1, df2], axis=1)` concatenates df1 and df2 along the columns, resulting in a DataFrame that combines the columns of both input DataFrames. Using `pd.concat()`, you can easily combine multiple DataFrames or Series into a single DataFrame, which is useful for data manipulation and analysis tasks.

2.4.2 Filtering DataFrames

Sure! Filtering a pandas DataFrame means selecting rows that meet certain criteria. This is often done using conditions on one or more columns. Here's a simple example to illustrate how filtering works:

2.4.3 Example DataFrame:

```
import pandas as pd

# Create an example DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 35, 40, 45],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Miami']
}

df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
```

2.4.4 Filtering the DataFrame:

Filtering a DataFrame means selecting rows that meet certain criteria. This is often done using conditions on one or more columns. Here's a simple example to illustrate how filtering works:

```python
import pandas as pd

# Create an example DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 35, 40, 45],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Miami']
}

df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)

# Filter rows where Age is greater than 30
filtered_df = df[df['Age'] > 30]

print("\nFiltered DataFrame (Age > 30):")
print(filtered_df)

# Filter rows where City is 'Chicago'
filtered_df_city = df[df['City'] == 'Chicago']

print("\nFiltered DataFrame (City is Chicago):")
print(filtered_df_city)


# Filter rows where Age is between 30 and 40 (inclusive)
filtered_df_age_range = df[(df['Age'] >= 30) & (df['Age'] <= 40)]

print("\nFiltered DataFrame (30 <= Age <= 40):")
print(filtered_df_age_range)
```

```
Original DataFrame:
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
3    David   40      Houston
4      Eve   45        Miami

Filtered DataFrame (Age > 30):
      Name  Age     City
```

```
2  Charlie   35  Chicago
3    David   40  Houston
4      Eve   45    Miami

Filtered DataFrame (City is Chicago):
      Name  Age     City
2  Charlie   35  Chicago

Filtered DataFrame (30 <= Age <= 40):
      Name  Age         City
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
3    David   40      Houston
```

Explanation:

- **df['Age'] > 30**: This creates a boolean Series that is `True` for rows where the 'Age' value is greater than 30 and `False` otherwise.
- **df[df['Age'] > 30]**: This filters the DataFrame, returning only the rows where the condition is `True`.
- **df['City'] == 'Chicago'**: This creates a boolean Series that is `True` for rows where the 'City' value is 'Chicago'.
- **df[(df['Age'] >= 30) & (df['Age'] <= 40)]**: This filters the DataFrame using multiple conditions. The `&` operator is used to combine conditions, ensuring both conditions must be `True` for a row to be included.

This example demonstrates how to filter a pandas DataFrame based on different conditions, helping you to extract specific subsets of data that meet your criteria.

2.4.5 Sorting

The sort_values() method in pandas is used to sort a DataFrame or Series by one or more columns or indices.

Syntax:

```
DataFrame.sort_values(by, axis=0, ascending=True,  na_position='last', ignore_index=False)
```

2.4.6 Parameters:

- **by**: (str or list of str) The name(s) of the column(s) or index level(s) to sort by.

- **axis**: (int or str, default 0) The axis to sort along. 0 or 'index' to sort rows, 1 or 'columns' to sort columns.
- **ascending**: (bool or list of bool, default True) Sort ascending vs. descending. Specify list for multiple sort orders.
- **na_position**: (str, default 'last') 'first' puts NaNs at the beginning, 'last' puts NaNs at the end.
- **ignore_index**: (bool, default False) If True, the resulting index will be labeled 0, 1, …, n - 1.

Example:

Let's create an example DataFrame and sort it using `sort_values()`.

```
# Create an example DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 35, 40, 45],
    'Salary': [50000, 60000, 75000, 80000, 70000]
}

df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)

# Sort the DataFrame by 'Age' in ascending order
sorted_df = df.sort_values(by='Age')
print("\nDataFrame sorted by Age:")
print(sorted_df)

# Sort the DataFrame by 'Salary' in descending order
sorted_df_desc = df.sort_values(by='Salary', ascending=False)
print("\nDataFrame sorted by Salary in descending order:")
print(sorted_df_desc)

# Sort the DataFrame by 'Age' and then by 'Salary'
sorted_df_multi = df.sort_values(by=['Age', 'Salary'])
print("\nDataFrame sorted by Age and then by Salary:")
print(sorted_df_multi)
```

In these examples: - The DataFrame is sorted by the 'Age' column in ascending order. - The DataFrame is sorted by the 'Salary' column in descending order. - The DataFrame is sorted first by the 'Age' column, and within each age group, by the 'Salary' column.

2.4.7 Groupby

pandas offers the ability to group related rows of DataFrames according to the values of other rows. This useful feature is accomplished using the groupby() method. The groupby method in pandas is used to group data based on one or more columns. It is often used with aggregation functions like sum(), mean(), count(), etc., to summarize data.

Syntax:

```
DataFrame.groupby(by, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=
```

Parameters descriptions:

- **by**: Specifies the column(s) or keys to group by. This can be a single column name, a list of column names, or a dictionary mapping column names to group keys.
- **axis**: Determines whether to group by rows (axis=0, default) or columns (axis=1).
- **as_index**: If True (default), the group labels are used as the index. If False, the group labels are retained as columns.
- **sort**: If True (default), the groups are sorted. If False, the groups are not sorted.
- **group_keys**: If True (default), adds group keys to the index. If False, the group keys are not added.

Syntax of common usages:

```
# Grouping by a Single Column
grouped = df.groupby('column_name')


# **Grouping by Multiple Columns**:


grouped = df.groupby(['column_name1', 'column_name2'])


# **Applying Aggregation Functions**:

grouped_mean = df.groupby('column_name')['target_column'].mean()


# **Using Multiple Aggregation Functions**:


grouped_agg = df.groupby('column_name').agg({
    'target_column1': 'mean',
    'target_column2': 'sum'
})
```

Example:

```
# Use the groupby() method, notice a DataFrameGroupBy object is returned
df[['City',"Age"]].groupby('City').mean()
```

City	Age
Chicago	35.0
Houston	40.0
Los Angeles	30.0
Miami	45.0
New York	25.0

- Here, the `DataFrameGroupBy` object can be most readily thought of as containing a DataFrame object for every group (in this case, a DataFrame object for each city).
- Specifically, each item of the object is a tuple, containing the group identifier (in this case the city), and the corresponding rows of the DataFrame that have that city.

Longer example:

```
# Create an example DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank'],
    'Age': [25, 30, 35, 40, 45, 30],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Miami', 'Chicago'],
    'Salary': [50000, 60000, 75000, 80000, 70000, 65000]
}

df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)

# Group by 'City' and calculate the mean salary for each city
grouped = df.groupby('City')['Salary'].mean()

print("\nMean Salary by City:")
print(grouped)
```

```python
# Group by 'City' and calculate the sum of salaries for each city
grouped_sum = df.groupby('City')['Salary'].sum()

print("\nSum of Salaries by City:")
print(grouped_sum)

# Group by 'City' and count the number of people in each city
grouped_count = df.groupby('City')['Name'].count()

print("\nCount of People by City:")
print(grouped_count)
```

```
Original DataFrame:
      Name  Age         City  Salary
0    Alice   25     New York   50000
1      Bob   30  Los Angeles   60000
2  Charlie   35      Chicago   75000
3    David   40      Houston   80000
4      Eve   45        Miami   70000
5    Frank   30      Chicago   65000

Mean Salary by City:
City
Chicago        70000.0
Houston        80000.0
Los Angeles    60000.0
Miami          70000.0
New York       50000.0
Name: Salary, dtype: float64

Sum of Salaries by City:
City
Chicago        140000
Houston         80000
Los Angeles     60000
Miami           70000
New York        50000
Name: Salary, dtype: int64

Count of People by City:
City
```

```
Chicago        2
Houston        1
Los Angeles    1
Miami          1
New York       1
Name: Name, dtype: int64
```

Explanation:

- **df.groupby('City')**: This groups the DataFrame by the 'City' column. Each unique value in 'City' will form a group.
- [**'Salary'].mean()**: This calculates the mean salary for each group (city).
- [**'Salary'].sum()**: This calculates the sum of salaries for each group (city).
- [**'Name'].count()**: This counts the number of entries for each group (city).
- **Grouping**: `df.groupby('City')` creates groups based on unique values in the 'City' column.
- **Aggregation**: Using aggregation functions like `mean()`, `sum()`, and `count()` allows you to summarize the data within each group.
- **Result**: The output shows the mean salary, sum of salaries, and count of people for each city, respectively.

The `groupby` method is very powerful for data analysis and manipulation, allowing you to easily aggregate and summarize data based on specific criteria.

2.4.8 Numpy's `where()`

The `np.where` function in is used to return elements chosen from two values based on whether a condition holds.

Syntax:

```
np.where(condition, x, y)
```

Parameters: - **condition**: An array-like object (e.g., a series or NumPy array, list, or etc) that evaluates to `True` or `False`. - **x**: The value to choose when the condition is `True`. - **y**: The value to choose when the condition is `False`.

`np.where(condition, x, y)` returns an array with elements from `x` where the condition is `True` and elements from `y` where the condition is `False`.

```
# Create an example DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, 30, 35, 40, 45]
}
df = pd.DataFrame(data)

# Use np.where to create a new column 'Age Group'
df['Age Group'] = np.where(df['Age'] >= 35, 'Senior', 'Junior')

print(df)
```

```
      Name  Age Age Group
0    Alice   25    Junior
1      Bob   30    Junior
2  Charlie   35    Senior
3    David   40    Senior
4      Eve   45    Senior
```

Explanation: - **Condition**: `df['Age'] >= 35` checks if the 'Age' column values are greater than or equal to 35. - **True**: For rows where the condition is `True`, it assigns 'Senior'. - **False**: For rows where the condition is `False`, it assigns 'Junior'.

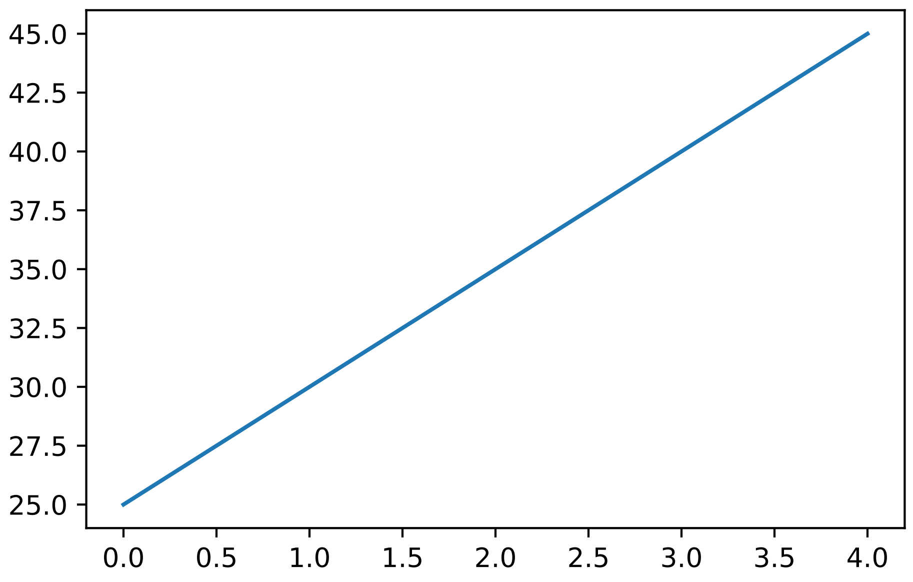
2.5 Plotting with `Matplotlib`

The standard Python plotting library is matplotlib. Let's import the library and instruct Jupyter to display the plots inline (i.e. display the plots to the notebook screen so we can see them as we run the code).

```
# import fundamental plotting library in Python
import matplotlib.pyplot as plt

# Instruct jupyter/VS Code to plot in the notebook
%matplotlib inline
```

To plot a series, use `.plot()`. We will come back to `matplotlib` later.

```
df['Age'].plot()
```



2.5.1 Datetime objects

Python's internal data representation of dates is given by DateTime objects. Datetime objects are crucial for handling time-related data in a structured way, enabling various operations like comparison, arithmetic, and formatting. `pandas` offers the to_datetime() method to convert a string that represents a given date format into a `datetime`-like object. This is useful for ensuring that date and time data are properly recognized and can be used for time series analysis, indexing, and plotting.

Syntax:

```
pd.to_datetime(arg, format=None)
```

2.5.2 Parameters:

- **arg**: The date/time string(s) or list-like object to convert.
- **format**: The strftime to parse time. For example, "%Y-%m-%d".

The `format` parameter in `pd.to_datetime()` is used to specify the exact format of the date/time strings being parsed. This is particularly useful when the input date strings do not conform to standard formats or when you want to improve parsing performance by explicitly defining the format.

Date Formatting Directives:

- **%Y**: Four-digit year (e.g., 2023).
- **%y**: Two-digit year (e.g., 23 for 2023).
- **%m**: Month as a zero-padded decimal number (e.g., 07 for July).
- **%B**: Full month name (e.g., July).
- **%b** or **%h**: Abbreviated month name (e.g., Jul for July).
- **%d**: Day of the month as a zero-padded decimal number (e.g., 03 for the 3rd).
- **%A**: Full weekday name (e.g., Monday).
- **%a**: Abbreviated weekday name (e.g., Mon).
- **%H**: Hour (24-hour clock) as a zero-padded decimal number (e.g., 14 for 2 PM).
- **%I**: Hour (12-hour clock) as a zero-padded decimal number (e.g., 02 for 2 PM).
- **%p**: AM or PM designation.
- **%M**: Minute as a zero-padded decimal number (e.g., 30 for 30 minutes past the hour).
- **%S**: Second as a zero-padded decimal number (e.g., 00 for 0 seconds).
- **%f**: Microsecond as a decimal number, zero-padded on the left (e.g., 000000).
- **%j**: Day of the year as a zero-padded decimal number (e.g., 189 for July 8th).
- **%U**: Week number of the year (Sunday as the first day of the week) as a zero-padded decimal number (e.g., 27).
- **%W**: Week number of the year (Monday as the first day of the week) as a zero-padded decimal number (e.g., 27).
- **%w**: Weekday as a decimal number (0 for Sunday, 6 for Saturday).
- **%Z**: Time zone name (e.g., UTC, EST).
- **%z**: UTC offset in the form +HHMM or -HHMM (e.g., +0530, -0800).
- **%%**: A literal '%' character.

Example formulae:

- `%Y-%m-%d` matches dates like `2023-07-03`.
- `%d/%m/%Y` matches dates like `03/07/2023`.
- `%B %d, %Y` matches dates like `July 3, 2023`.
- `%I:%M %p` matches times like `02:30 PM`.
- `%H:%M:%S.%f` matches times like `14:30:00.000000`.

Examples: **Parsing Date in Non-Standard Format**:

```python
# Example date string in non-standard format
date_str = '03-07-23'  # This represents July 3, 2023 in YY-MM-DD format

# Convert to datetime using format parameter
date = pd.to_datetime(date_str, format='%y-%m-%d')
print(date)
```

```
2003-07-23 00:00:00
```

In this example: - `%y-%m-%d` specifies the format where `%y` represents the two-digit year, `%m` represents the month, and `%d` represents the day.

Parsing Date and Time Together:

```
date_time_str = '07/03/2023 14:30:00'

# Convert to datetime using format parameter
date_time = pd.to_datetime(date_time_str, format='%m/%d/%Y %H:%M:%S')
print(date_time)
```

```
2023-07-03 14:30:00
```

In this example: - `%m/%d/%Y %H:%M:%S` specifies the format where `%m/%d/%Y` represents the date in MM/DD/YYYY format, and `%H:%M:%S` represents the time in HH:MM:SS format.

Handling Dates with Textual Month:

```
date_str_textual = 'July 3, 2023'

# Convert to datetime using format parameter
date_textual = pd.to_datetime(date_str_textual, format='%B %d, %Y')
print(date_textual)
```

```
2023-07-03 00:00:00
```

In this example: - `%B %d, %Y` specifies the format where `%B` represents the full month name (e.g., July), `%d` represents the day, and `%Y` represents the four-digit year.

You can also use ChatGPT or Google the format of your date at hand!

3 Data Transformation I

In this case, we learned how to further maniuplate DataFrames with `pandas` and `numpy`. Notably, creating your own functions and creating pivot tables.

3.1 Preliminary modules

```
import numpy as np
import pandas as pd
```

3.2 Some more `pandas` functions

3.2.1 Unique

The `unique` function in pandas is used to find the unique values in a Series or a column of a DataFrame. It returns the unique values as a NumPy array. This function is useful when you need to identify the distinct values in a dataset.

Syntax:

```
pandas.Series.unique()
```

This method returns the unique values in the Series.

Examples:

```
# Creating a Series
data = pd.Series([1, 2, 2, 3, 4, 4, 4, 5])

# Finding unique values
unique_values = data.unique()

print(unique_values)
```

```
# In this example, `data.unique()` returns a NumPy array containing the unique values `[1, 2


# Example 2: Finding Unique Values in a DataFrame Column

# In this example, `df['A'].unique()` returns a NumPy array of unique values in column 'A', a


# Creating a DataFrame
df = pd.DataFrame({
    'A': [1, 2, 2, 3, 4, 4, 4, 5],
    'B': ['a', 'b', 'b', 'c', 'd', 'd', 'd', 'e']
})

# Finding unique values in column 'A'
unique_values_A = df['A'].unique()

# Finding unique values in column 'B'
unique_values_B = df['B'].unique()

print("Unique values in column A:", unique_values_A)
print("Unique values in column B:", unique_values_B)
```

```
[1 2 3 4 5]
Unique values in column A: [1 2 3 4 5]
Unique values in column B: ['a' 'b' 'c' 'd' 'e']
```

The `.apply` function

Sure! The `.apply()` function in pandas is used to apply a function along the axis of a DataFrame or to elements of a Series. This function is highly versatile and can be used to perform complex operations on your data.

Basic Syntax

For a Series:

```
Series.apply(func, convert_dtype=True, args=())
```

For a DataFrame:

```
DataFrame.apply(func, axis=0, raw=False, result_type=None, args=(), **kwds)
```

- **func**: The function to apply to each element (Series) or to each column/row (DataFrame).
- **axis**: {0 or 'index', 1 or 'columns'}, default 0. The axis along which the function is applied:
  - 0 or 'index': apply function to each column.
  - 1 or 'columns': apply function to each row.
- **args**: Positional arguments to pass to the function.
- **raw**: Determines if the function is passed raw ndarray values or pandas objects (Series/DataFrame).
- **result_type**: {'expand', 'reduce', 'broadcast', None}, default None.

Examples:

Example 1: Applying a Function to Each Element in a Series

```
import pandas as pd

# Creating a Series
data = pd.Series([1, 2, 3, 4, 5])

# Function to square each element
def square(x):
    return x * x

# Applying the function
squared_data = data.apply(square)

print(squared_data)
```

Output:

```
0     1
1     4
2     9
3    16
4    25
dtype: int64
```

In this example, the `square` function is applied to each element of the Series `data`, resulting in a new Series `squared_data` where each value is the square of the corresponding original value.

Example 2: Applying a Function to Each Column in a DataFrame

```python
import pandas as pd

# Creating a DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
})

# Function to sum the elements of a column
def column_sum(col):
    return col.sum()

# Applying the function to each column
column_sums = df.apply(column_sum, axis=0)

print(column_sums)
```

Output:

```
A     6
B    15
C    24
dtype: int64
```

In this example, the `column_sum` function is applied to each column of the DataFrame `df`, resulting in a Series `column_sums` containing the sum of the elements in each column.

Example 3: Applying a Function to Each Row in a DataFrame

```python
import pandas as pd

# Creating a DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
})

# Function to find the maximum value in a row
def row_max(row):
    return row.max()

# Applying the function to each row
row_maxs = df.apply(row_max, axis=1)

print(row_maxs)
```

Output:

```
0    7
1    8
2    9
dtype: int64
```

In this example, the `row_max` function is applied to each row of the DataFrame `df`, resulting in a Series `row_maxs` containing the maximum value in each row.

The `.apply()` function in pandas is a powerful tool for applying custom functions to Series or DataFrame columns/rows. It is highly flexible and can handle a variety of operations, making it essential for data transformation and analysis tasks. The basic syntax involves calling the `.apply()` method with the function to apply and the axis along which to apply it, yielding a transformed Series or DataFrame.

3.2.2 Upper and lower case strings

1. `str.lower`

The `str.lower` function in pandas is used to convert all characters in a string to lowercase. It is often used when you want to standardize text data, making it easier to compare strings that might have different cases.

Syntax:

```
Series.str.lower()
```

Example:

```python
# Creating a Series
data = pd.Series(['Hello', 'World', 'PANDAS'])

# Converting to lowercase
lowercase_data = data.str.lower()

print(lowercase_data)
```

Output:

```
0    hello
1    world
2    pandas
dtype: object
```

2. `str.upper`

The `str.upper` function in pandas is used to convert all characters in a string to uppercase. This is useful for standardizing text data to a uniform case.

Syntax:

```python
Series.str.upper()
```

Example:

```python
# Creating a Series
data = pd.Series(['hello', 'world', 'pandas'])

# Converting to uppercase
uppercase_data = data.str.upper()

print(uppercase_data)
```

Output:

```
0    HELLO
1    WORLD
2    PANDAS
dtype: object
```

3. `count`

The `count` function in pandas is used to count the number of occurrences of a specified value in a Series or to count non-NA/null entries across a DataFrame.

Syntax for Series:

```
Series.str.count(pat)
```

- **pat**: The pattern or substring to count.

Example: Counting substrings in a Series:

```
# Creating a Series
data = pd.Series(['apple', 'banana', 'apple pie', 'cherry'])

# Counting occurrences of 'apple'
apple_count = data.str.count('apple')

print(apple_count)
```

Output:

```
0    1
1    0
2    1
3    0
dtype: int64
```

- **`str.lower`**: Converts all characters in a string to lowercase.
- **`str.upper`**: Converts all characters in a string to uppercase.
- **`str.count`**: Counts the occurrences of a pattern or substring in a Series.
- **`.count()`**: Counts the number of non-NA/null entries in a Series or DataFrame.

These functions are useful for text data standardization and for counting occurrences of specific values or patterns in your data.

3.3 Personalized functions

3.3.1 Custom functions

In Python, you can create your own functions to encapsulate reusable code, improve readability, and make your programs more modular. Here's a brief explanation of how to define and use your own functions in Python.

A function is defined using the `def` keyword, followed by the function name, parentheses `()`, and a colon `:`. The code block within the function is indented.

Syntax:

```python
def function_name(parameters):
    """Docstring (optional): A brief description of the function."""
    # Function body
    # Code to be executed
    return value  # Optional: return statement to return a value
```

- **function_name**: The name of the function.
- **parameters**: (Optional) A list of parameters (or arguments) that the function accepts.
- **Docstring**: (Optional) A string describing what the function does.
- **return**: (Optional) The value that the function returns.

Example 1: A Simple Function

Here's a simple example of a function that takes no parameters and prints a message:

```python
def greet():
    """Print a greeting message."""
    print("Hello, world!")

# Calling the function
greet()
```

Output:

```
Hello, world!
```

Example 2: A Function with Parameters

Here's a function that takes two parameters and returns their sum:

```python
def add(a, b):
    """Return the sum of two numbers."""
    return a + b

# Calling the function
result = add(3, 5)
print(result)
```

Output:

```
8
```

Example 3: A Function with Default Parameters

You can also define functions with default parameter values:

```python
def greet(name="world"):
    """Print a greeting message to the given name."""
    print(f"Hello, {name}!")

# Calling the function with and without an argument
greet("Alice")
greet()
```

Output:

```
Hello, Alice!
Hello, world!
```

Example 4: A Function with Variable Number of Arguments

Sometimes you might want to define a function that can accept a variable number of arguments using `*args` and `**kwargs`:

```python
def print_numbers(*args):
    """Print all the numbers passed as arguments."""
    for number in args:
        print(number)

# Calling the function with multiple arguments
print_numbers(1, 2, 3, 4, 5)
```

Output:

```
1
2
3
4
5
```

Example 5: A Function with Keyword Arguments

You can use `**kwargs` to accept a variable number of keyword arguments:

```python
def print_info(**kwargs):
    """Print key-value pairs of the passed keyword arguments."""
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Calling the function with multiple keyword arguments
print_info(name="Alice", age=30, city="New York")
```

Output:

```
name: Alice
age: 30
city: New York
```

- **Define a function** using the `def` keyword.
- **Provide parameters** within the parentheses (optional).
- **Add a docstring** to describe the function (optional but recommended).
- **Write the function body** with the code to be executed.
- **Return a value** using the `return` statement (optional).

Creating functions in Python helps organize code, reduce repetition, and improve clarity. Functions can take parameters, have default values, and handle a variable number of arguments.

3.3.2 The functions `map`, `filter` and `sorted`

The `map()`, `filter()`, and `sorted()` functions are built-in Python functions that provide powerful tools for working with iterable objects, such as lists or tuples. Here's a brief overview of each function:

`map()`

The `map()` function applies a given function to all items in an input list (or other iterable).

Syntax:

```
map(function, iterable, ...)
```

- **`function`**: A function that is applied to each item of the iterable.
- **`iterable`**: One or more iterable objects (e.g., lists, tuples).

Example:

```
# Example function
def square(x):
    return x ** 2

# Input list
numbers = [1, 2, 3, 4, 5]

# Apply `square` function to each item in the list
squared_numbers = map(square, numbers)

# Convert the map object to a list and print
print(list(squared_numbers))
```

Output:

```
[1, 4, 9, 16, 25]
```

The `filter()` function constructs an iterator from elements of an iterable for which a function returns true.

Syntax:

```
filter(function, iterable)
```

- **`function`**: A function that tests if each element of an iterable returns true or false.
- **`iterable`**: An iterable to be filtered.

Example:

```python
# Example function
def is_even(x):
    return x % 2 == 0

# Input list
numbers = [1, 2, 3, 4, 5]

# Apply `is_even` function to filter even numbers
even_numbers = filter(is_even, numbers)

# Convert the filter object to a list and print
print(list(even_numbers))
```

Output:

```
[2, 4]
```

`sorted()`

The `sorted()` function returns a new sorted list from the items in an iterable.

Syntax:

```python
sorted(iterable, key=None, reverse=False)
```

- **`iterable`**: An iterable to be sorted.
- **`key`**: A function that serves as a key for the sort comparison (optional).
- **`reverse`**: A boolean value. If `True`, the list elements are sorted as if each comparison were reversed (optional, default is `False`).

Example:

```python
# Input list
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5]

# Sort the list
sorted_numbers = sorted(numbers)

print(sorted_numbers)
```

Output:

```
[1, 1, 2, 3, 4, 5, 5, 6, 9]
```

- `map()`: Applies a function to every item in an iterable and returns a map object of the results.
- `filter()`: Constructs an iterator from elements of an iterable for which a function returns true.
- `sorted()`: Returns a new sorted list from the items in an iterable, with optional custom sorting behavior.

3.3.3 Anonymous functions

In Python, anonymous functions are defined using the `lambda` keyword. These functions are often called "lambda functions" and are used for creating small, one-time, and inline function objects. Unlike regular functions defined with `def`, lambda functions are limited to a single expression.

Syntax of Lambda Functions

```
lambda arguments: expression
```

- `lambda`: The keyword used to define an anonymous function.
- `arguments`: A comma-separated list of parameters.
- `expression`: A single expression that is evaluated and returned.

Example 1: Simple Lambda Function

Here's a simple example of a lambda function that adds two numbers:

```
# Lambda function to add two numbers
add = lambda x, y: x + y

# Using the lambda function
result = add(3, 5)
print(result)
```

Output:

```
8
```

Example 2: Lambda Function in `map()`

Lambda functions are often used with functions like `map()`, `filter()`, and `sorted()`.

```
# List of numbers
numbers = [1, 2, 3, 4, 5]

# Using lambda with map() to square each number
squared_numbers = list(map(lambda x: x ** 2, numbers))

print(squared_numbers)
```

Output:

```
[1, 4, 9, 16, 25]
```

Example 3: Lambda Function in `filter()`

```
# List of numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Using lambda with filter() to get even numbers
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even_numbers)
```

Output:

```
[2, 4, 6, 8, 10]
```

Example 4: Lambda Function in `sorted()`

```
# List of tuples
points = [(1, 2), (3, 1), (5, -1), (2, 3)]

# Using lambda with sorted() to sort by the second element of each tuple
sorted_points = sorted(points, key=lambda x: x[1])

print(sorted_points)
```

Output:

```
[(5, -1), (3, 1), (1, 2), (2, 3)]
```

Characteristics and Limitations:

- **Single Expression**: Lambda functions can only contain a single expression. They cannot include statements or annotations.
- **No `return` Statement**: The expression result is implicitly returned.
- **Limited Use**: Lambda functions are best used for short, simple operations. For more complex functions, it is better to define a regular function using `def`.

When to Use Lambda Functions:

- **Short, Simple Functions**: When you need a quick function for a short, simple operation.
- **Inline Functions**: When you need a function for a one-time use, especially within functions like `map()`, `filter()`, or `sorted()`.
- **Readability**: When using a lambda function improves the readability and conciseness of your code.

Lambda functions in Python provide a concise way to create anonymous functions for simple, one-time operations. They are defined using the `lambda` keyword and can be used wherever function objects are required. However, due to their limitations, they are best suited for short, simple tasks and should be used judiciously to maintain code readability.

3.4 Pivot Tables

The `pd.pivot_table()` function in pandas is a powerful tool for summarizing and reshaping data. It allows you to aggregate data and create a new table that is a more compact and organized representation of your original DataFrame.

Syntax:

```
pd.pivot_table(data, values=None, index=None, columns=None, aggfunc='mean', fill_value=None,
```

- `data`: The DataFrame to pivot.
- `values`: Column(s) to aggregate.
- `index`: Column(s) to set as index.
- `columns`: Column(s) to pivot.
- `aggfunc`: Function to aggregate the data (default is 'mean').
- `fill_value`: Value to replace missing values.
- `dropna`: Do not include columns whose entries are all NaN (default is True).

Example:

Let's consider a dataset containing sales data:

```
# Sample data
data = {
    'Region': ['North', 'South', 'East', 'West', 'North', 'South', 'East', 'West'],
    'Product': ['A', 'A', 'B', 'B', 'A', 'B', 'A', 'B'],
    'Sales': [100, 150, 200, 130, 120, 170, 160, 180],
    'Quantity': [10, 15, 20, 13, 12, 17, 16, 18]
}

df = pd.DataFrame(data)

print(df)
```

Output:

```
   Region Product  Sales  Quantity
0   North       A    100        10
1   South       A    150        15
2    East       B    200        20
3    West       B    130        13
4   North       A    120        12
5   South       B    170        17
6    East       A    160        16
7    West       B    180        18
```

Creating a Pivot Table:

Let's create a pivot table to summarize the total sales and quantity for each region and product.

```
pivot_table = pd.pivot_table(df, values=['Sales', 'Quantity'], index=['Region'], columns=['P

print(pivot_table)
```

Output:

```
        Sales          Quantity
Product     A    B          A   B
Region
East      160  200         16  20
North     220    0         22   0
South     150  170         15  17
West        0  310          0  31
```

Explanation:

1. **data**: The DataFrame to pivot (`df` in this case).
2. **values**: The columns to aggregate (`'Sales'` and `'Quantity'`).
3. **index**: The column(s) to set as the index of the pivot table (`'Region'`).
4. **columns**: The column(s) to pivot (`'Product'`).
5. **aggfunc**: The aggregation function (`'sum'`), to get the total sales and quantity.
6. **fill_value**: The value to replace missing values (`0`).

The resulting pivot table shows the total sales and quantities for each combination of region and product. The rows represent the regions, and the columns represent the products. The values in the table are the sums of sales and quantities.

The `pd.pivot_table()` function in pandas is a versatile tool for data summarization and reshaping. It allows you to: - Aggregate data using various functions (e.g., sum, mean). - Pivot data by specifying index and columns. - Handle missing values using `fill_value`.

3.5 Line plots with `seaborn`

Line plots are useful for visualizing data over continuous intervals or time series. The module `seaborn` is a powerful data visualization library built on top of Matplotlib, you can create line plots easily using the `lineplot` function.

Syntax:

```
seaborn.lineplot(data=None, *, x=None, y=None, hue=None, size=None, style=None, palette=None
```

- **data**: DataFrame, array, or list of arrays, optional. Dataset for plotting.
- **x**: Name of the x-axis variable.
- **y**: Name of the y-axis variable.
- **hue**: Grouping variable that will produce lines with different colors.
- **size**: Grouping variable that will produce lines with different widths.
- **style**: Grouping variable that will produce lines with different dash patterns.
- **palette**: Colors to use for different levels of the hue variable.

Example 1: Simple Line Plot

Here's a basic example of creating a line plot with Seaborn.

```
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt
```

```
# Sample data
data = {
    'Year': [2015, 2016, 2017, 2018, 2019, 2020],
    'Sales': [200, 300, 400, 500, 600, 700]
}

df = pd.DataFrame(data)

# Create a line plot
sns.lineplot(data=df, x='Year', y='Sales')

# Show the plot
plt.show()
```

Example 2: Line Plot with Multiple Lines

You can plot multiple lines by specifying a `hue` parameter.

```
# Sample data
data = {
    'Year': [2015, 2016, 2017, 2018, 2019, 2020, 2015, 2016, 2017, 2018, 2019, 2020],
    'Sales': [200, 300, 400, 500, 600, 700, 100, 150, 200, 250, 300, 350],
    'Product': ['A', 'A', 'A', 'A', 'A', 'A', 'B', 'B', 'B', 'B', 'B', 'B']
}

df = pd.DataFrame(data)

# Create a line plot with multiple lines
sns.lineplot(data=df, x='Year', y='Sales', hue='Product')

# Show the plot
plt.show()
```

Example 3: Customizing Line Plot

You can customize the appearance of your line plot by using various parameters and options.

```
# Sample data
data = {
    'Year': [2015, 2016, 2017, 2018, 2019, 2020],
    'Sales': [200, 300, 400, 500, 600, 700]
}
```

```
df = pd.DataFrame(data)

# Create a customized line plot
sns.lineplot(data=df, x='Year', y='Sales', marker='o', linestyle='--', color='red')

# Add titles and labels
plt.title('Yearly Sales')
plt.xlabel('Year')
plt.ylabel('Sales')

# Show the plot
plt.show()
```

- **Creating Line Plots**: Use `sns.lineplot()` to create line plots easily.
- **Multiple Lines**: Differentiate groups with the `hue` parameter.
- **Customization**: Customize appearance using parameters like `marker`, `linestyle`, and `color`.
- **Data Integration**: Seamlessly integrates with pandas DataFrames, allowing for intuitive plotting using column names.

3.6 Writing to a csv file

Certainly! The `to_csv()` function in pandas is used to export a DataFrame to a CSV (Comma-Separated Values) file. CSV files are a common data storage format that is both human-readable and easy to process with various software tools. You can customize the output by specifying various parameters.

Syntax:

```
DataFrame.to_csv(path_or_buf=None, sep=',', na_rep='', float_format=None, columns=None, heade
```

Parameters:

- **`path_or_buf`**: The file path or object where the CSV will be saved. If not specified, the result is returned as a string.
- **`sep`**: The string to use as the separating character (default is a comma `,`).
- **`na_rep`**: String representation of missing data.
- **`float_format`**: Format string for floating-point numbers.
- **`columns`**: Subset of columns to write to the CSV file.
- **`header`**: Write out the column names (default is `True`).
- **`index`**: Write row names (index) (default is `True`).

- **index_label**: Column label for the index column(s) if desired.
- **encoding**: Encoding to use for writing (default is `None`).
- **date_format**: Format string for datetime objects.

Example 1: Simple Export

Here's a basic example of exporting a DataFrame to a CSV file.

```python
import pandas as pd

# Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}

df = pd.DataFrame(data)

# Export DataFrame to CSV
df.to_csv('people.csv')
```

This will create a file named `people.csv` with the following content:

```
,Name,Age,City
0,Alice,25,New York
1,Bob,30,Los Angeles
2,Charlie,35,Chicago
```

Example 2: Customizing Output

You can customize the output by specifying different parameters.

```python
# Export DataFrame to CSV without the index and with a custom separator
df.to_csv('people_no_index.csv', index=False, sep=';')
```

This will create a file named `people_no_index.csv` with the following content:

```
Name;Age;City
Alice;25;New York
Bob;30;Los Angeles
Charlie;35;Chicago
```

Example 3: Handling Missing Values and Specifying Columns

```python
# Sample DataFrame with missing values
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [25, None, 35, 40],
    'City': ['New York', 'Los Angeles', 'Chicago', None]
}

df = pd.DataFrame(data)

# Export DataFrame to CSV with custom NA representation and specific columns
df.to_csv('people_missing_values.csv', na_rep='N/A', columns=['Name', 'Age'])
```

This will create a file named `people_missing_values.csv` with the following content:

```
,Name,Age
0,Alice,25.0
1,Bob,N/A
2,Charlie,35.0
3,David,40.0
```

Summary

- **Basic Usage**: Use `to_csv('filename.csv')` to export a DataFrame to a CSV file.
- **Custom Separator**: Change the delimiter using the `sep` parameter.
- **No Index**: Exclude the DataFrame index using `index=False`.
- **Handle Missing Values**: Specify a representation for missing values with `na_rep`.
- **Select Columns**: Export only specific columns using the `columns` parameter.
- **Advanced Options**: Control formatting, quoting, encoding, and more with additional parameters.

4 Data Transformation II

In this case, we covered more data manipulation with `pandas`, as well as introducing a variety of new plots, and the libraries `matplotlib` and `seaborn`. You should familiarize yourself with these libraries.

4.1 Preliminary modules

```
# Load packages
import os
import pandas as pd
import numpy as np

# This line is needed to display plots inline in Jupyter Notebook
%matplotlib inline

# Required for basic python plotting functionality
import matplotlib.pyplot as plt

# Required for formatting dates later in the case
import datetime
import matplotlib.dates as mdates

# Advanced plotting functionality with seaborn
import seaborn as sns

sns.set(style="whitegrid")  # can set style depending on how you'd like it to look
```

4.2 Overview of `matplotlib` plotting

For an introduction to `matplotlib` please use their quickstart guide.

4.3 Overview of `seaborn` plotting

Seaborn can be used to make more advanced plots, it also has a fairly simple syntax. We give a very brief introduction here. For more information, please follow the tutorials here.

Syntax:

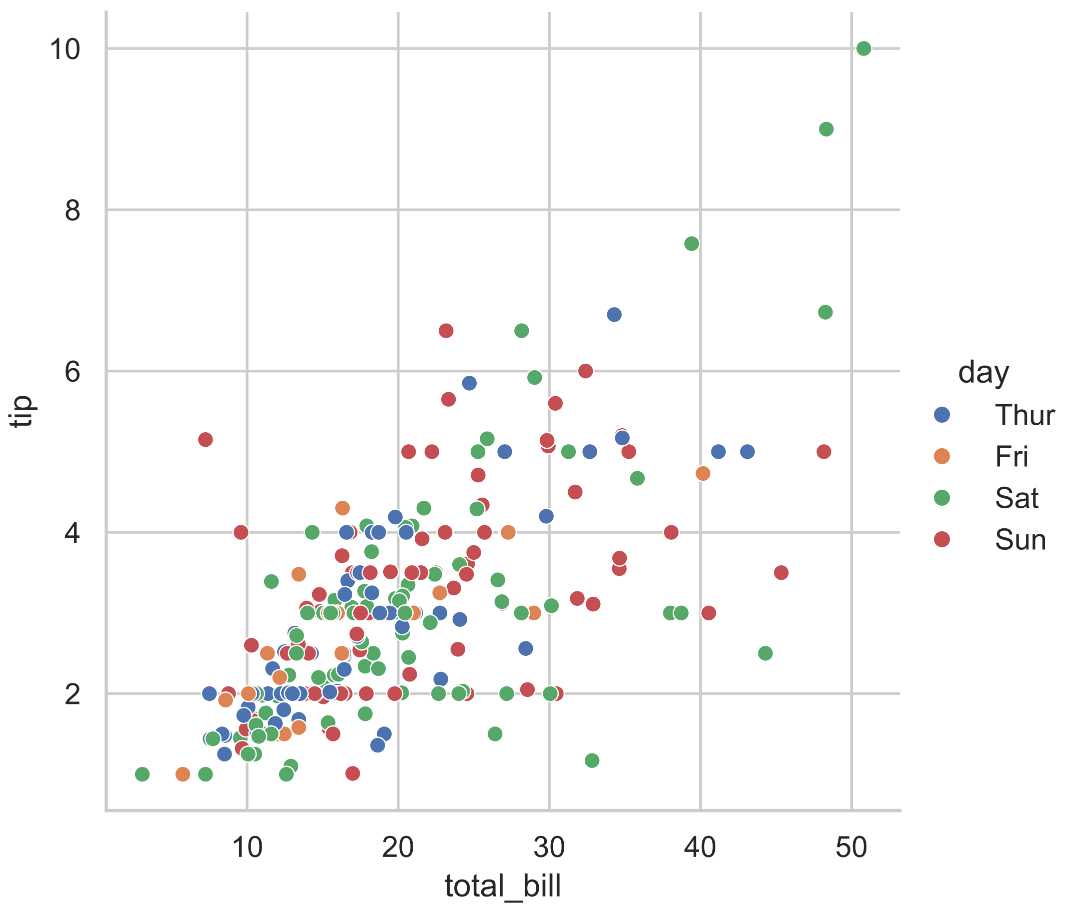
```
sns.plot_function(data=pandas DF, x=column for the x-axis, y=column for the y-axis, hue=colu
```

Essentially, you specify the DataFrame, which variables go on the y-axis and the x-axis. Optionally, you can specidy a variable for the **`hue`** parameter. When it is specified, Seaborn assigns different colors to different levels of the **`hue`** variable, making it easy to see how different categories compare. **`kind`** tells seaborn whether to use lines or points etc.

Example:

Our example will use **`relplot`**. The **`relplot`** function in Seaborn is a high-level interface for creating relational plots that combine several plots like **`scatterplot`** and **`lineplot`**. It allows you to easily visualize relationships between multiple variables in a dataset. We'll use the built-in **`tips`** dataset from Seaborn, which contains information about restaurant tips.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load the tips dataset
tips = sns.load_dataset('tips')

# Create a relational plot using relplot
sns.relplot(data=tips, x='total_bill', y='tip', hue='day', kind='scatter')

# Show the plot
plt.show()
```



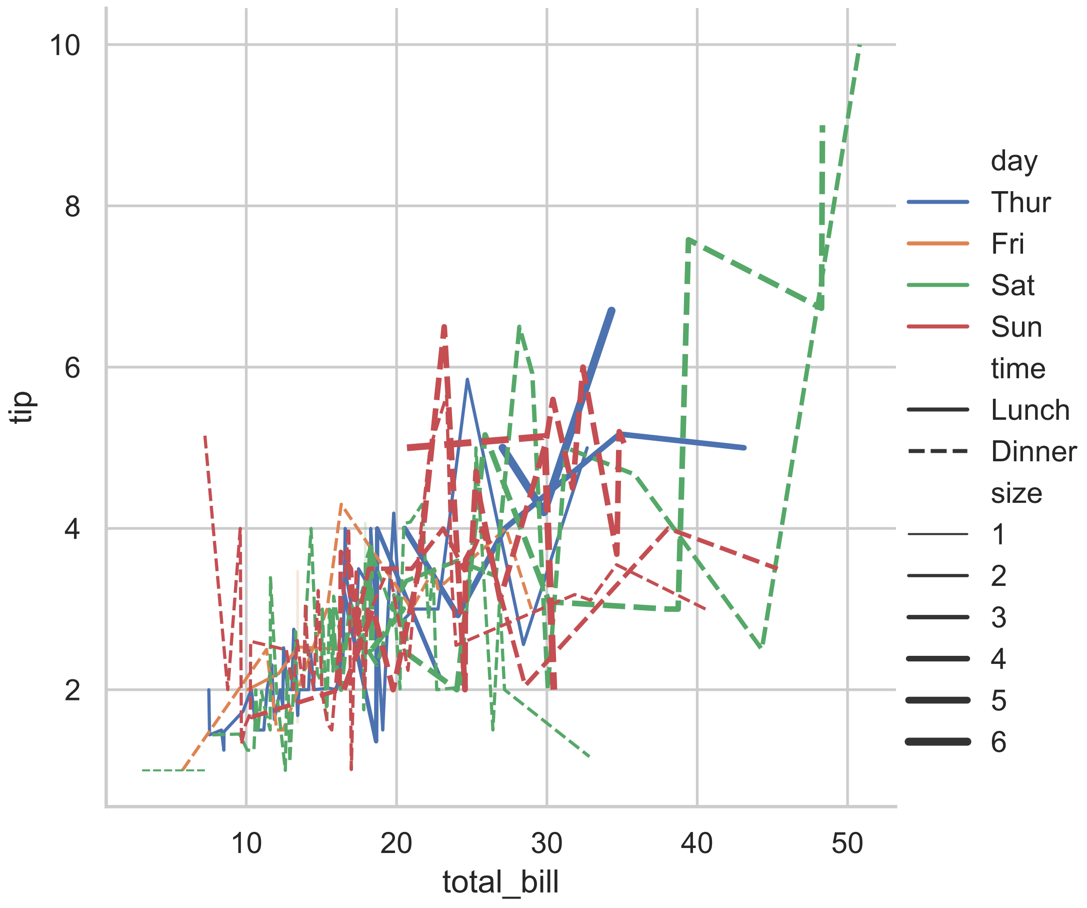
Explanation:

- **Data**: The `tips` dataset contains columns such as `total_bill`, `tip`, `sex`, `smoker`, `day`, `time`, and `size`.
- **x**: The `total_bill` column is used for the x-axis.
- **y**: The `tip` column is used for the y-axis.
- **hue**: The `day` column is used to color the points differently for each day of the week.

The resulting plot shows a scatter plot of `total_bill` vs. `tip`, with different colors representing different days of the week. This makes it easy to see if there are any trends or differences in tipping behavior on different days.

You can further customize the `relplot` with additional parameters, such as changing the kind of plot to a line plot and adding more dimensions with `style` and `size`.

```
# Create a line plot with relplot
sns.relplot(data=tips, x='total_bill', y='tip', hue='day', style='time', size='size', kind='l
# Show the plot
plt.show()
```



4.4 Overview of new plots

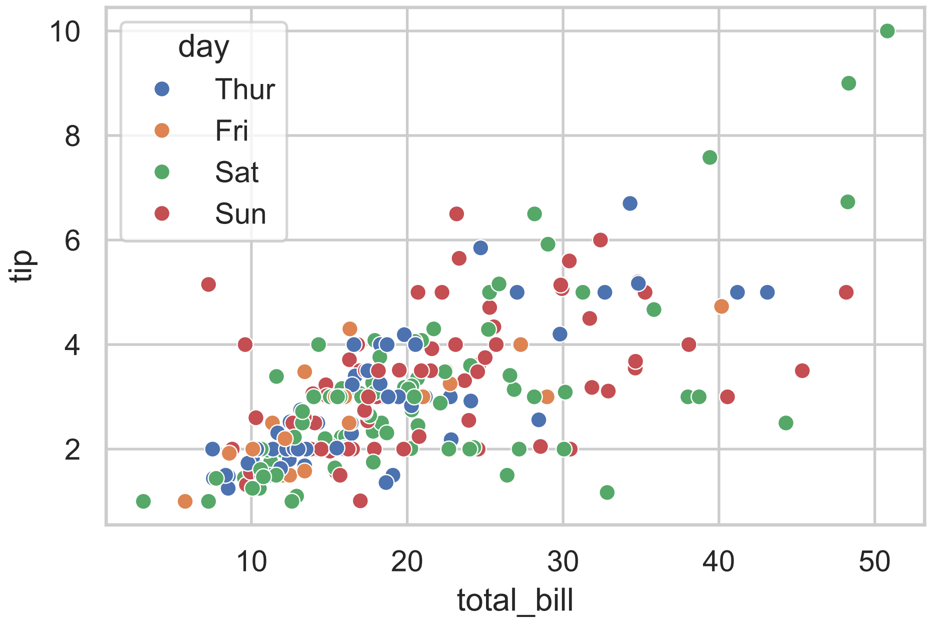
Below is a brief overview of each of the new plot-types introduced in Case 4, along with the Seaborn syntax for creating them:

4.4.1 Scatterplot

A scatter plot is used to display the relationship between two continuous variables. Each point represents an observation in the dataset.

Example:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load example data
tips = sns.load_dataset('tips')

# Create a scatter plot
sns.scatterplot(data=tips, x='total_bill', y='tip', hue='day')

# Show the plot
plt.show()
```



4.4.2 Histogram

A histogram displays the distribution of a single continuous variable by dividing the data into bins and counting the number of observations in each bin. When analyzing a histogram, people typically look for several key characteristics:

1. **Shape of the Distribution**

    - **Symmetry**: Whether the distribution is symmetric or asymmetric. A symmetric histogram has a bell-shaped curve.

- **Skewness**: The direction of the tail. A histogram with a long tail on the right side is right-skewed (positive skewness), and a histogram with a long tail on the left side is left-skewed (negative skewness).
- **Modality**: The number of peaks in the histogram. A unimodal histogram has one peak, a bimodal histogram has two peaks, and a multimodal histogram has multiple peaks.

2. **Central Tendency**

   - **Mean**: The average value of the data, which can be roughly identified by the center of the distribution.
   - **Median**: The middle value of the data, which is helpful to compare with the mean.

3. **Spread or Variability**

   - **Range**: The difference between the maximum and minimum values, indicating the spread of the data.
   - **Standard Deviation**: Although not directly visible on the histogram, the spread of the bars gives an idea of how dispersed the data is.
   - **Interquartile Range (IQR)**: The range within which the central 50% of the data lies, often inferred by looking at the central bulk of the histogram.

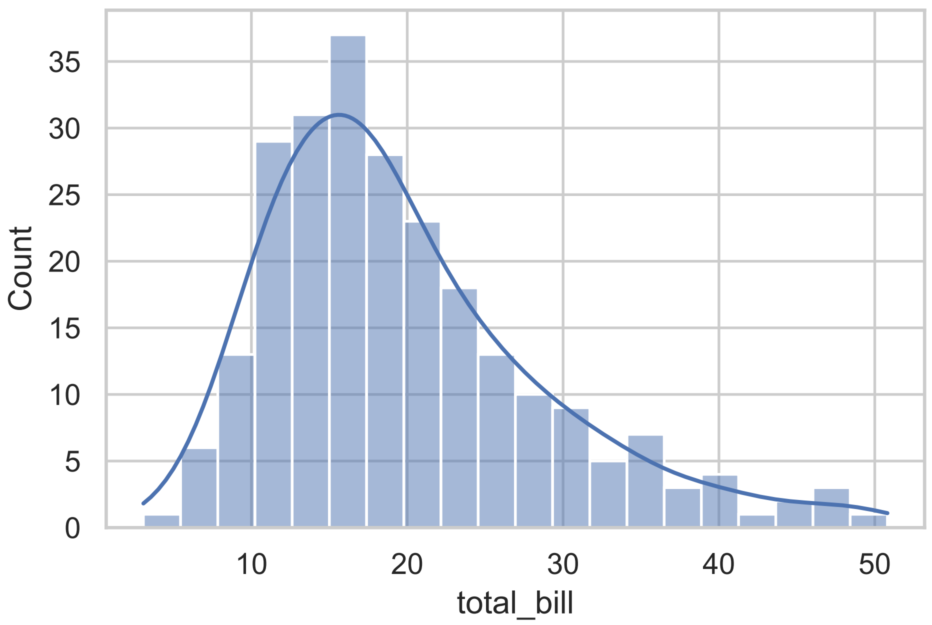
4. **Outliers**

   - **Extreme Values**: Data points that fall far outside the general distribution, which can be seen as isolated bars away from the main body of the histogram.

Example:

Consider a histogram of the variable `total_bill` from a restaurant tips dataset:

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Load the tips dataset
tips = sns.load_dataset('tips')

# Create a histogram
sns.histplot(data=tips, x='total_bill', bins=20, kde=True)

# Show the plot
plt.show()
```



When looking at this histogram, you might analyze it as follows:

1. **Shape**: Determine if the distribution is symmetric, skewed left, or skewed right. For example, if the histogram is right-skewed, it suggests that higher total bills are less common.
2. **Central Tendency**: Look for the central peak to get a sense of where most of the `total_bill` values lie. This provides insight into the average bill amount.
3. **Spread**: Assess how spread out the `total_bill` values are. A wide spread indicates more variability in the bill amounts.
4. **Outliers**: Identify any bars that are far away from the main distribution, indicating unusually high or low bill amounts.
5. **Frequency**: Observe the height of the bars to understand how many observations fall into each bin.

4.4.3 Boxplot

Like a histogram, a boxplot (or box-and-whisker plot) displays the distribution of a continuous variable. It displays the five-number summary:

- **Lower whisker**: The maximum of 1.5 times the interquartile range and the minimum.
- **First quartile (Q1)**: The 25th percentile of the data. It is the left (or bottom) edge of the box.
- **Median (Q2)**: The middle value of the data set (50th percentile). It is represented by the line inside the box.

- **Third quartile (Q3)**: The 75th percentile of the data. It is the right (or top) edge of the box.
- **Upper whisker**: The minimum of 1.5 times the interquartile range and the maximum.

We can view the following characteristics: 1. **Spread of the data** - The Interquartile range (IQR) is the difference between the first quartile (Q1) and the third quartile (Q3). It represents the middle 50% of the data. This is the size of the box. THe larger the box, the larger the spread of the data.

2. **Location of the data**

   - The location of the data can be read by looking at the line in the middle of the box, the median.

3. **Symmetry and Skewness of the data**

   - **Symmetry**: If the median is in the center of the box and the whiskers are of equal length, the data is symmetric.
   - **Skewness**: If the median is not centered or the whiskers are of unequal length, the data is skewed. A longer whisker on the right indicates right skewness (positive skew), and a longer whisker on the left indicates left skewness (negative skew).

4. **Tails or outliers**

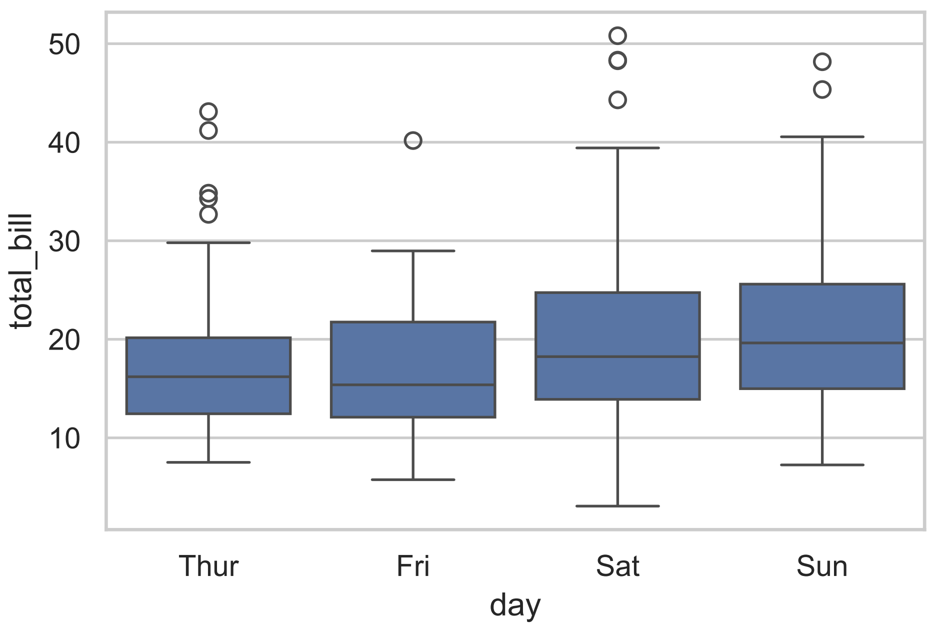
   - Many points beyond the whiskers represents a heavy tail (a high tendency for observations to be far from the median.) A few points beyond the whiskers may indicate outliers.

Multiple boxplots are often plotted together, to compare distributions for different populations. In this case, the boxplot can be used to determine the relationship between a categorical variable and a continuous variable.

Example:

Let's create a boxplot using the `tips` dataset from Seaborn:

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Load the tips dataset
tips = sns.load_dataset('tips')

# Create a box plot
sns.boxplot(data=tips, x='day', y='total_bill')

# Show the plot
plt.show()
```



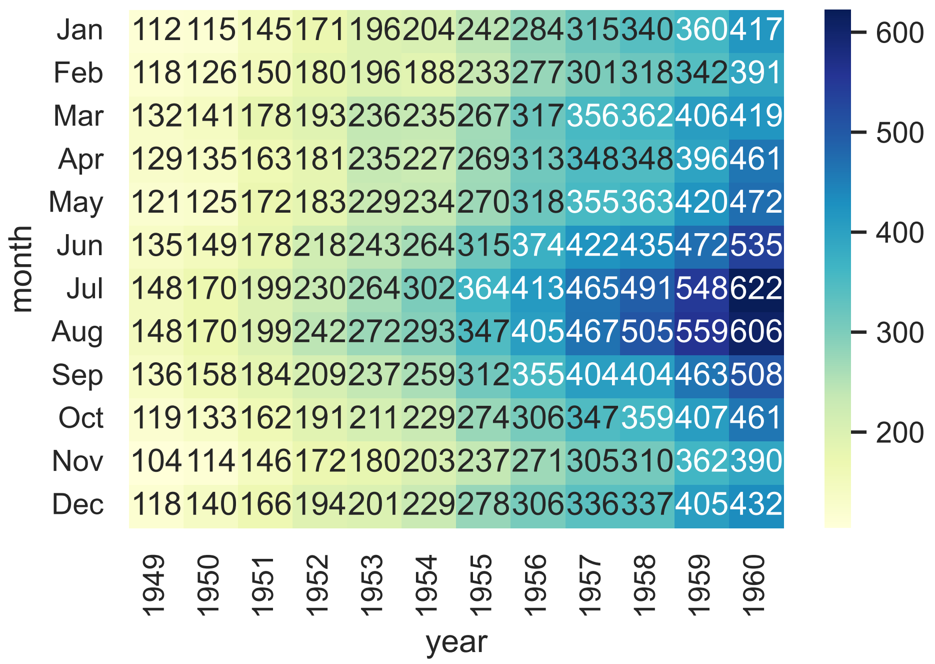
Here, the spread of Saturday is larger than that of the Thursday. In addition, we have that average the bills are larger on the weekend. The distributions appear symmetric for all days.

4.4.4 Heatmap

A heatmap compares two categorical variables to a continuous variable. The two categorical variables are represented on the horizontal and vertical axes, and the intensity of the cells represent the continuous variable.

Example:

```
# Load example data
flights = sns.load_dataset('flights')

# Pivot the data to create a matrix
flights_pivot = flights.pivot(index='month', columns='year', values='passengers')

# Create a heatmap
sns.heatmap(flights_pivot, annot=True, fmt='d', cmap='YlGnBu')

# Show the plot
plt.show()
```

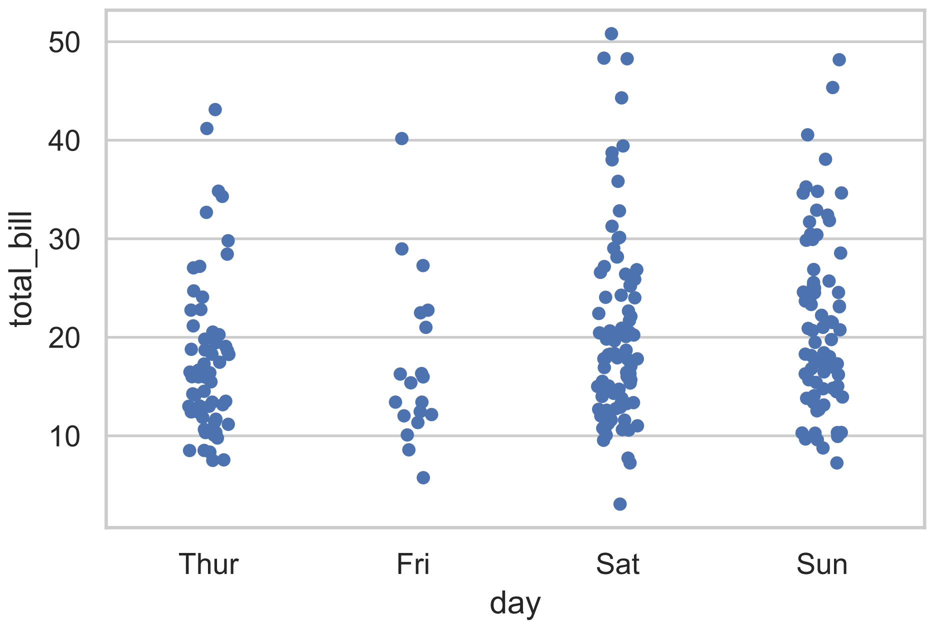


4.4.5 Stripplot

A strip plot is used to display the distribution of a single continuous variable or the relationship between a continuous variable and a categorical variable. Each observation is represented as a point.

Example:

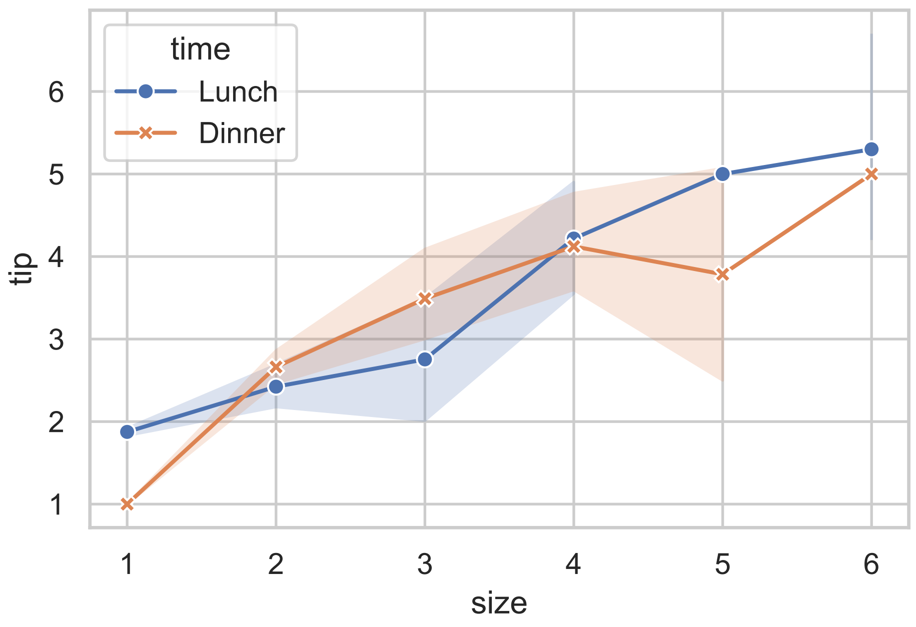
```
# Create a strip plot
sns.stripplot(data=tips, x='day', y='total_bill', jitter=True)

# Show the plot
plt.show()
```



4.4.6 Lineplot

A line plot is used to display the relationship between two continuous variables, often to visualize trends over time.

```
# Create a line plot
sns.lineplot(data=tips, x='size', y='tip', hue='time', style='time', markers=True, dashes=Fal

# Show the plot
plt.show()
```



4.4.7 Summary

- **Scatterplot**: Shows the relationship between two continuous variables.
- **Histogram**: Displays the distribution of a single continuous variable.
- **Boxplot**: Summarizes the distribution of a continuous variable or the relationship between a continuous and a categorical variable.
- **Heatmap**: Represents data in a matrix format with colors indicating values.
- **Stripplot**: Displays the distribution of a continuous variable or the relationship between a continuous and a categorical variable.
- **Lineplot**: Visualizes the relationship between two continuous variables, often to show trends.

4.5 Misc. Python functions used:

Here are brief explanations for some of the new functions:

1. `.dropna()`:

   - This method is used to remove missing values (NaNs) from a DataFrame or Series. By default, it drops any row containing at least one missing value. However, it can be configured to drop columns instead, or only rows/columns with all missing values, depending on the parameters passed. This is useful for cleaning data before analysis or visualization.

2. **`pd.merge`**:
   - This function from the pandas library is used to merge two DataFrames based on one or more common columns or indices. This is useful for combining datasets that have related information spread across different tables.
3. **`plt.subplots()`**:
   - This function from the matplotlib library creates a figure and a set of subplots. It returns a tuple containing a figure object and an array of axes objects. This function is handy for creating complex plots with multiple subplots in a single figure, allowing for more detailed and organized visualizations. The function can be customized to specify the number of rows and columns of subplots.

5 Interpretation of charts and graphs

The goal of this case was to introduce the power of charts, and demonstrate how they can be used to uncover key patterns in a dataset. In this case, we learnt that, after cleaning, it is useful to examine the summary statistics of important variables. Later, then we can look at the variables individually, via one-dimensional charts. After which, we can evaluate relationships between variables via 2-dimensional charts. To look at 3-way relationships, one can use color or size in charts.

5.1 Steps you can take to avoid bad charts

Here are some steps to help avoid people misinterpretating your charts and ensure your charts communicate data effectively and accurately:

1. **Label Axes Clearly**:
   - **Action**: Always provide clear and descriptive labels for the x-axis and y-axis.
   - **Benefit**: This helps viewers understand what the chart is measuring and comparing.
2. **Use Appropriate Scales**:
   - **Action**: Ensure the scales on your axes are appropriate for the data being presented, starting from zero where necessary.
   - **Benefit**: This avoids exaggerating differences or trends in the data.
3. **Provide Context**:
   - **Action**: Include necessary context or background information, such as data source, time period, and any relevant notes.
   - **Benefit**: Context helps viewers interpret the data correctly and understand its relevance.
4. **Choose the Right Chart Type**:
   - **Action**: Select the chart type that best represents your data (e.g., bar chart for categorical data, line chart for trends over time).
   - **Benefit**: This ensures that the data is presented in the most understandable format.
5. **Avoid Overloading with Data**:

- **Action**: Limit the amount of data displayed in a single chart to avoid clutter.
- **Benefit**: A cleaner chart helps viewers focus on the key insights without being overwhelmed.

6. **Use Consistent Color Schemes**:
   - **Action**: Use consistent and color-blind friendly color schemes that differentiate data points without causing confusion.
   - **Benefit**: This helps in distinguishing different categories or series and avoids misinterpretation due to color issues.
7. **Highlight Key Insights**:
   - **Action**: Use annotations, different colors, or other visual cues to highlight key data points or trends.
   - **Benefit**: This draws attention to the most important parts of the data and aids in interpretation.
8. **Avoid Distorting Data**:
   - **Action**: Avoid using 3D effects or other distortions that can misrepresent data.
   - **Benefit**: This ensures that the data is presented accurately and prevents misleading viewers.
9. **Show Data Distribution**:
   - **Action**: Use boxplots, histograms, or other charts to show data distribution, not just summary statistics.
   - **Benefit**: This provides a fuller picture of the data, revealing any underlying patterns or outliers.
10. **Add Descriptive Titles and Legends**:
    - **Action**: Include descriptive titles and legends that clearly explain what the chart is showing.
    - **Benefit**: This helps viewers quickly grasp the purpose and content of the chart.
11. **Use Gridlines Sparingly**:
    - **Action**: Use gridlines to help viewers accurately read values, but don't overuse them to the point of clutter.
    - **Benefit**: This aids in precision without overwhelming the chart.
12. **Avoid Cherry-Picking Data**:
    - **Action**: Present all relevant data, not just data that supports a particular narrative or viewpoint.
    - **Benefit**: This ensures a fair and unbiased representation of the data.
13. **Review and Test Charts**:

    - **Action**: Review charts with colleagues or test with a sample audience to identify potential misinterpretations.
    - **Benefit**: Feedback can help identify and correct any issues before presenting the chart to a wider audience.

14. **Use Interactive Elements**:
    - **Action**: When possible, use interactive charts that allow viewers to explore the data more deeply.
    - **Benefit**: This enables viewers to drill down into the data and gain a better understanding.

15. **Regularly Update Data**:
    - **Action**: Ensure that the data in your charts is up-to-date and accurate.
    - **Benefit**: This maintains the relevance and reliability of the insights provided by the charts.

5.2 Parallel coordinates plots

Certainly! A parallel coordinates plot displays the values of multiple variables for a set of data points on parallel axes, allowing you to see patterns and relationships between the variables.

- **Axes**: Each variable in the dataset is represented by a vertical axis.
- **Lines**: Each data point is represented by a line that connects the values of each variable across the parallel axes.
- **Patterns**: By observing how the lines intersect and align, you can identify patterns, correlations, and clusters in the data.
- **Comparisons**: It is useful for comparing multiple observations and understanding the overall structure of the dataset.

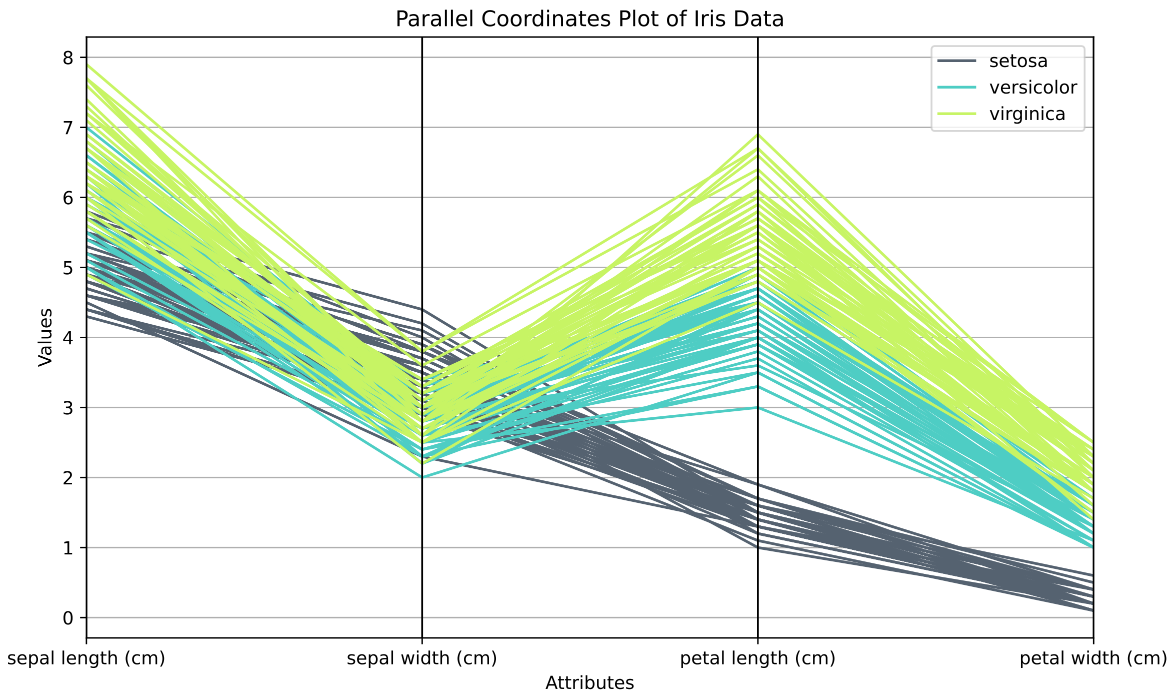
Example:

Let's create an example using the `pandas` and `matplotlib` libraries in Python. We'll use the Iris dataset, which contains measurements of different species of iris flowers. Install `scikit-learn` if neccessary:

```
!pip install scikit-learn
```

Let's create the plot

```python
import pandas as pd
import matplotlib.pyplot as plt
from pandas.plotting import parallel_coordinates

# Load the Iris dataset
from sklearn.datasets import load_iris
iris_data = load_iris()

# Convert to a DataFrame
iris = pd.DataFrame(iris_data.data, columns=iris_data.feature_names)
iris['species'] = pd.Categorical.from_codes(iris_data.target, iris_data.target_names)

# Create a parallel coordinates plot
plt.figure(figsize=(10, 6))
parallel_coordinates(iris, 'species', color=('#556270', '#4ECDC4', '#C7F464'))

# Show the plot
plt.title('Parallel Coordinates Plot of Iris Data')
plt.xlabel('Attributes')
plt.ylabel('Values')
plt.show()
```



- **Lines**: Each line represents an individual iris flower sample.
- **Axes**: The vertical axes represent the four measurements (sepal length, sepal width, petal length, and petal width).
- **Colors**: Different colors represent different species of iris flowers (Setosa, Versicolor, Virginica).
- **Patterns**: By examining how the lines group and intersect, we can observe that:
  - Setosa flowers have distinctly different measurements compared to Versicolor and Virginica flowers.
  - Versicolor and Virginica have overlapping measurements, but with careful observation, patterns and differences can still be discerned.

5.3 Colored backgrounds on scatterplots

In this case, we colored the background based on the color of the closest point in the dataset. This is useful for classification tasks - where we want to determine if continuous variates are able to classify data points. Here is a simple example of how to do this in Python. This process includes creating a mesh grid of points, determining the closest point in the dataset for each point in the grid, and coloring the grid accordingly.

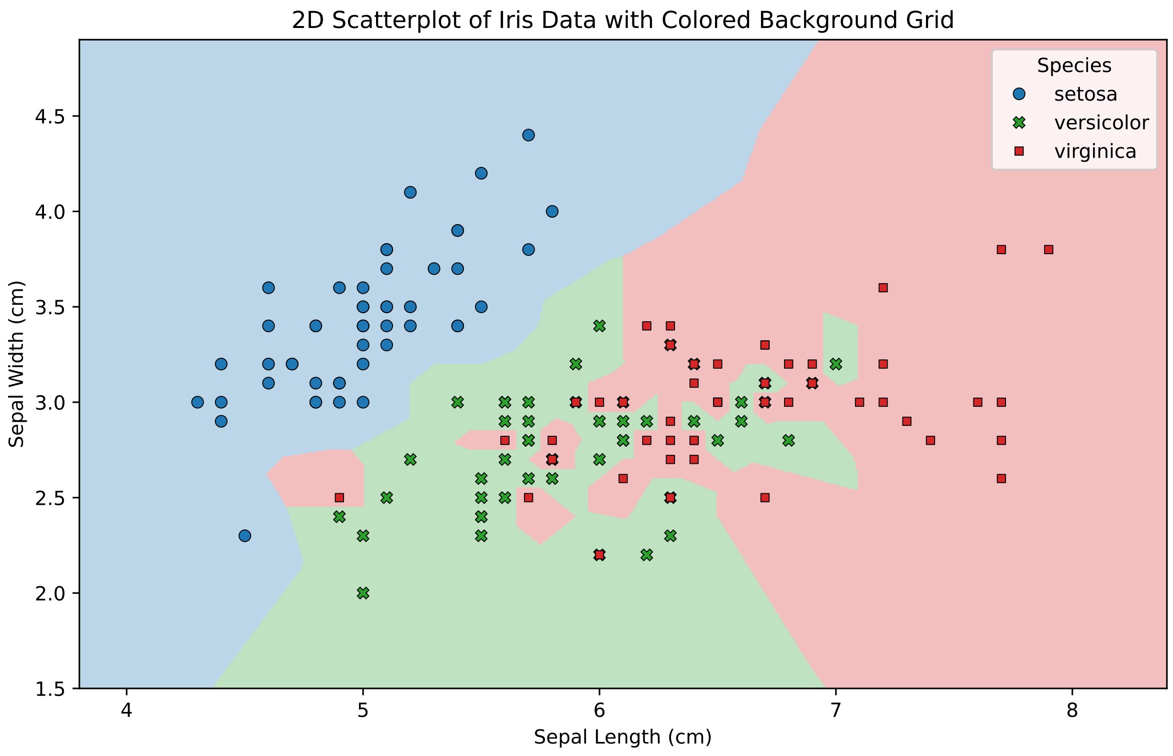
Example:

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.spatial import cKDTree
from matplotlib.colors import ListedColormap

# Load the Iris dataset
iris = sns.load_dataset('iris')

# Extract the relevant data
x = iris['sepal_length']
y = iris['sepal_width']
species = iris['species']

# Create a color map for species - dictionary of colors to species
species_colors = {'setosa': (0.1215, 0.466666, 0.705882),
                  'versicolor': (0.17254, 0.6274, 0.17259),
                  'virginica': (0.83921, 0.15294, 0.1568)}

# This is a series object where each row is the color associated with teh species
```

```
colors = iris['species'].map(species_colors)
print(colors)

# Create a mesh grid
x_min, x_max = x.min() - 0.5, x.max() + 0.5
y_min, y_max = y.min() - 0.5, y.max() + 0.5
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500), np.linspace(y_min, y_max, 500))

# Combine the grid points into a single array
grid_points = np.c_[xx.ravel(), yy.ravel()]

# This code finds the closes points

# Create a KDTree for fast nearest-neighbor lookup
tree = cKDTree(np.c_[x, y])

# Find the index of the closest point in the dataset for each grid point
_, idx = tree.query(grid_points)

# Map the indices to colors
grid_colors = np.array(colors.tolist())[idx].reshape(xx.shape + (3,))

# Plot the background grid
plt.figure(figsize=(10, 6))
plt.imshow(grid_colors, extent=(x_min, x_max, y_min, y_max), origin='lower', aspect='auto', a

# Overlay the scatter plot
sns.scatterplot(x=x, y=y, hue=species, palette=species_colors, style=species, edgecolor='k')

# Customize and show the plot
plt.title('2D Scatterplot of Iris Data with Colored Background Grid')
plt.xlabel('Sepal Length (cm)')
plt.ylabel('Sepal Width (cm)')
plt.legend(title='Species')
plt.show()
```

```
0      (0.1215, 0.466666, 0.705882)
1      (0.1215, 0.466666, 0.705882)
2      (0.1215, 0.466666, 0.705882)
3      (0.1215, 0.466666, 0.705882)
4      (0.1215, 0.466666, 0.705882)
                    ...
```

```
145      (0.83921, 0.15294, 0.1568)
146      (0.83921, 0.15294, 0.1568)
147      (0.83921, 0.15294, 0.1568)
148      (0.83921, 0.15294, 0.1568)
149      (0.83921, 0.15294, 0.1568)
Name: species, Length: 150, dtype: object
```



6 Exploratory Data Analysis

This case introduces **exploratory data analysis (EDA)** on an unfamiliar dataset. Here, we learned how to systematically approach investigating an unknown dataset while maintaining creativity to look for insights. Exploratory data analysis is an essential data analysis step and should never be skipped.

6.1 Preliminary modules

```
import numpy                    as np
import pandas                   as pd
import matplotlib.pyplot        as plt
import seaborn                  as sns
import sklearn.metrics          as Metrics


import folium  #needed for interactive map
from folium.plugins import HeatMap

from   collections          import Counter
from   sklearn              import preprocessing
from   datetime             import datetime
from   collections          import Counter
from   math                 import exp
from   sklearn.linear_model import LinearRegression as LinReg
from   sklearn.metrics      import mean_absolute_error
from   sklearn.metrics      import median_absolute_error
from   sklearn.metrics      import r2_score

%matplotlib inline
sns.set()
```

6.2 Attributes of the dataset

When loading in a new dataset it is important to answer the following questions:

1. What is the size of the dataset?
   - Potentially useful Python attributes/methods: `shape`
2. What are the important columns to my research question?
   - Potentially useful Python attributes/methods: `.columns, .head(), .index`
3. What does the data look like in these columns? Is it clean? What are the unique values?
   - Potentially useful Python attributes/methods: `.head(), .unique(), .isnull()`
4. How many missing values are there? How are they coded?
   - Potentially useful Python attributes/methods: `.isnull(), .dropna(), .isnull().sum()`
5. Do I need to create new columns?
   - Potentially useful Python attributes/methods: `.apply()`

6.3 Assessing variables individually

Once our dataset is clean and we have identified the important variables, then we can investigate the distributions of the important variables in isolation. This helps identify anything "odd" going on in our data, helps us understand what the dataset looks like, and helps us understand the properties of each of the variables (location, spread, skewness, tails).

One way to do this is through printing summary statistics. Potentially useful Python attributes/methods: `.describe(), .value_counts(), .mean(), .quantile(), .std(), .median()` - See Data extraction and transformation for more.

We can also make one-dimensional charts - bar charts, histograms and boxplots - to assess each variate. Potentially useful Python attributes/methods: `.hist(), .barplot(), .boxplot(), .bar()` - See the plots covered in the previous cases for more.

6.3.1 Barplots

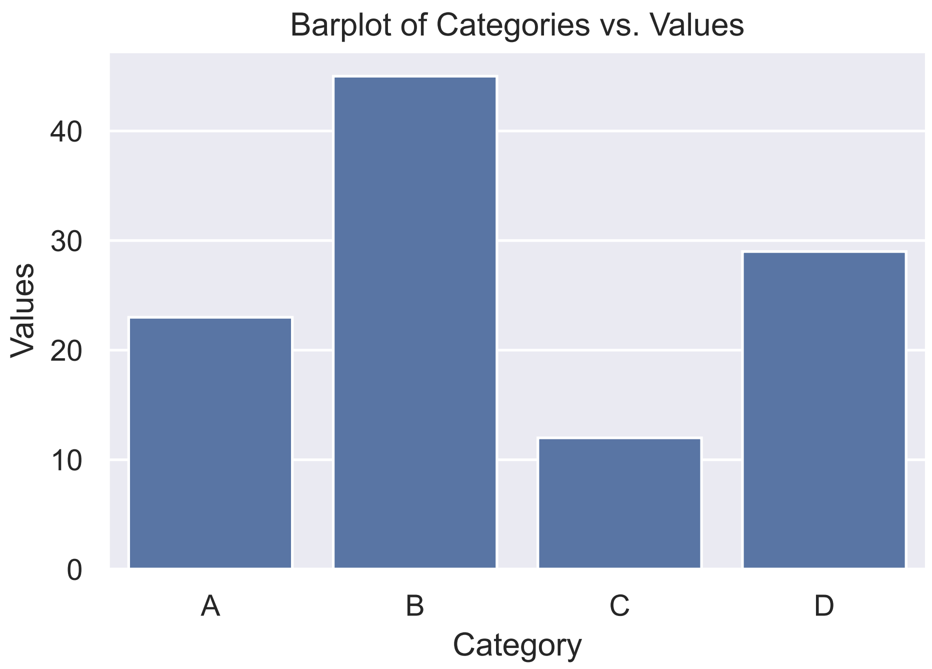
Barplots provide a way to summarize a categorical column or variable. Each bar displays the number of occurences of each category. Sometimes, when one category has most of the responses, it is useful to instead set the height of the bar to be the proportion of the responses attributed to each category.

Example:

- **Seaborn**: Use `sns.barplot()` for creating bar plots with a high-level, statistical focus.
- **Matplotlib**: Use `plt.bar()` for more basic bar plots and lower-level control over plot elements.

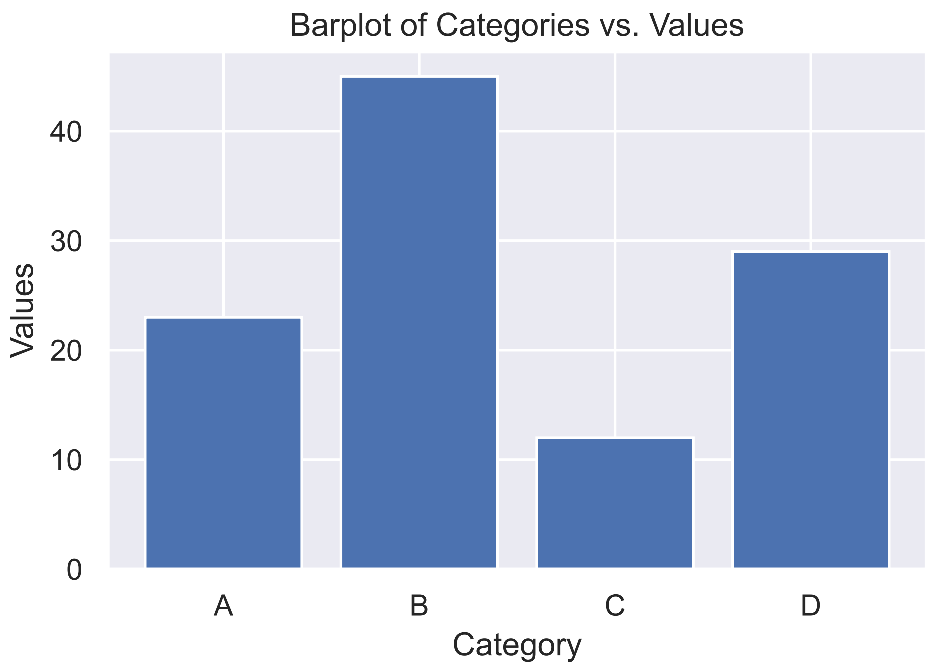
Here is an example of how to use Seaborn's `.barplot()` function:

```python
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Sample DataFrame
data = {
    'Category': ['A', 'B', 'C', 'D'],
    'Values': [23, 45, 12, 29]
}
df = pd.DataFrame(data)

# Create a barplot
sns.barplot(x='Category', y='Values', data=df)
plt.title('Barplot of Categories vs. Values')
plt.show()
```



While `.barplot()` is specific to Seaborn, Matplotlib provides similar functionality through the `bar()` function:

```python
import matplotlib.pyplot as plt

# Sample data
categories = ['A', 'B', 'C', 'D']
values = [23, 45, 12, 29]

# Create a barplot
plt.bar(categories, values)
plt.title('Barplot of Categories vs. Values')
plt.xlabel('Category')
plt.ylabel('Values')
plt.show()
```



6.4 Relationships between two variables

6.4.1 Investigating relationships between two variables

Once we have sufficient knowledge of the single variable distributions, known as the univariate distributions, we can inspect relationships between variables. Sometimes, it is too much to look at each pair of variables, so our overall research question(s) should guide which relationships we investigate. Here, to compare continuous to categorical variables, we can use boxplots, and to compare two continuous variables, we can use scatterplots or line plots. Potentially useful Python attributes/methods: `.relplot(), .boxplot()` - See the plots covered in the previous cases for more.

We can also use the correlation (see below) between two continuous variates. Note that when a correlation does not match with out intution, there may be an interaction effect, and we should investigate three-way relationships with these variables.

We should be looking for how the mean/median, scale, skewness and outliers of one variate changes with respect to the other. Take note of any interesting patterns. If we see something unusual or unexpected, then we should investigate further to explain it. This will also help in the future, when we use staitsitcal/machine learning models on the data. Having a good knowledge of what the dataset looks like will help you build better models, and identify problems with your models.

6.4.2 Correlation

Correlation is a statistical measure that describes the strength and direction of a relationship between two variables. Correlation tells us how closely two variables move together. When two variables are correlated, knowing the value of one variable helps you predict the value of the other. Correlation falls on a scale of -1 to 1. There are two main types of correlation:

1. **Positive Correlation**: This occurs when two variables tend to increase or decrease together. If one variable increases, the other variable also tends to increase. Conversely, if one decreases, the other tends to decrease.
2. **Negative Correlation**: This occurs when one variable tends to increase as the other decreases, and vice versa. When one variable increases, the other tends to decrease.

6.4.3 Examples:

- **Positive Correlation**:
  - Example: As the number of hours studied increases, test scores tend to increase.
  - Example: Higher levels of exercise are correlated with lower levels of obesity.
- **Negative Correlation**:
  - Example: As the temperature decreases, heating costs tend to increase.
  - Example: Increased smoking is correlated with a higher incidence of lung cancer.

6.4.4 Pitfalls of Correlation:

1. **Correlation Does Not Imply Causation**: Just because two variables are correlated does not mean that one causes the other to change. There may be other factors (confounding variables) influencing both variables.
2. **Non-linear Relationships**: Correlation measures only linear relationships. If the relationship between variables is non-linear (e.g., quadratic), correlation may not accurately reflect the strength of the relationship.
3. **Outliers**: Extreme values (outliers) can disproportionately influence correlation calculations, leading to misleading results.
4. **Spurious Correlations**: Sometimes variables may appear to be correlated by chance, without any meaningful relationship. Care should be taken to analyze whether we have enough sample points to ensure that the computed correlations are useful.

Correlation is a useful measure for understanding relationships between variables. However, it's important to interpret correlation carefully, considering other factors and potential limitations, to avoid drawing incorrect conclusions.

The relevent python function is `DataFrame.corr()`. This returns the correlation matrix of the variates in the relevant DataFrame. The correlation matrix is a matrix, whose ij th entry is the correlation between variable in column i and column j.

6.5 Relationships between three variables

6.5.1 Location data and `folium`

Whenever one variable we are interested in is a set of locations, it is useful to use the `folium` package to create visuals. Please visit this Quickstart guide to learn more about using `folium`.

6.5.2 Other relationships between three variables

Again, we can use color and/or size to add a third variate to the scatter and boxplots. You can use the `hue` parameter in the `seaborn` package to achieve this. This is particularly useful for identifying or investigating interaction effects.

6.5.3 Interaction effects

An interaction effect refers to a situation where the effect of one variable on an outcome depends on the level or value of another variable.

In simpler terms:

Imagine you have two variables: - **Variable A**: Age (Young or Old) - **Variable B**: Treatment (New Drug or Placebo)

An interaction effect occurs when the effect of Variable A (Age) on the outcome (e.g., improvement in health) is different depending on whether Variable B (Treatment) is a New Drug or Placebo.

- **Dependence**: An interaction effect means that the relationship between one variable and an outcome is not consistent across different levels of another variable.
- **Significance**: Finding an interaction effect can change how we interpret the impact of individual variables on outcomes. It suggests that the combined effect of variables is different from what would be predicted by their individual effects alone.

Example:

Let's say researchers are studying the effect of a new drug on health improvement in both young and old patients. They find that the drug is more effective in younger patients than older patients. This shows a main effect of age on health improvement.

Interaction Effect: However, further analysis reveals that the difference in effectiveness between young and old patients depends on whether they received the drug or a placebo. If the drug has a significantly larger effect on young patients compared to old patients (more than what would be expected from just the main effects of age and treatment), then we say there is an interaction effect between age and treatment.

Example:

- Imagine we have a study on the effectiveness of a new teaching method (Variable A) on student performance (Outcome), where we also consider the student's prior knowledge (Variable B). If the teaching method is more effective for students with high prior knowledge but less effective for students with low prior knowledge, then there is an interaction effect between teaching method and prior knowledge.

6.5.4 Kernel density plots

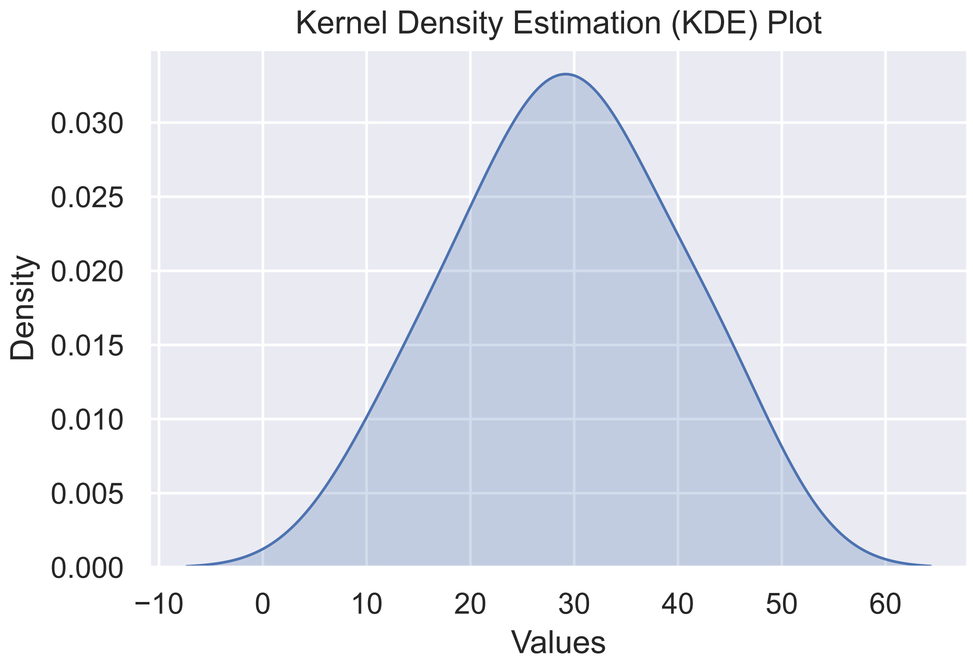
We learned a new plot, called the kernel density plot. This is a smoothed version of the histogram. The bandwidth parameter controls the smoothness of the resulting KDE plot. Kernel density plots are useful to compare the distributions of two or more different samples, as they can be nicely overlaid. They also look nicer than a histogram.

Example:

To create a kernel density plot, use the `sns.kdeplot()` function.

```python
import seaborn as sns
import matplotlib.pyplot as plt
import pandas as pd

# Sample data
data = pd.Series([23, 45, 12, 29, 30, 35, 42, 18, 25, 33])

# Create a KDE plot with Seaborn
sns.kdeplot(data, shade=True)
plt.title('Kernel Density Estimation (KDE) Plot')
plt.xlabel('Values')
plt.ylabel('Density')
plt.show()
```

```
C:\Users\12RAM\AppData\Local\Temp\ipykernel_19996\2075725077.py:9: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(data, shade=True)
```



Explanation of the Code:

- **`sns.kdeplot()`**: This function plots the KDE of the given data.
  - **`data`**: The input data for which the KDE plot will be generated. It can be a Pandas Series, NumPy array, or a list.
  - **`shade=True`**: Adds shading beneath the KDE curve for better visualization.
- **`plt.title()`, `plt.xlabel()`, `plt.ylabel()`**: These functions from Matplotlib are used to add a title, x-axis label, and y-axis label to the plot, respectively.

Additional Options:

- **Adjusting Bandwidth**: You can adjust the bandwidth of the KDE plot using the `bw_adjust` parameter in `sns.kdeplot()`. Lower values result in a smoother plot, while higher values result in a more jagged plot.

- **Multiple Plots**: `sns.kdeplot()` can also be used to plot KDEs for multiple variables or groups by passing multiple datasets or using the `hue` parameter.

7 Data cleaning/wrangling

In this case we learnt the steps we must take to clean a dataset. We also learnt several ways to handle missing data.

7.1 Preliminary modules

```python
import seaborn as sns
import matplotlib.pyplot as plt
import base64
import datetime
import json
import os
import numpy as np
import pandas as pd
from geopy.geocoders import Nominatim
```

7.2 Handling missing data

One of the first steps in data cleaning is to deal with null or missing values. First, one should determine how missing values are labelled in the dataset, as this can vary widely. Some common encodings for missing data are as follows:

1. **NaN or nan**: This is the most common representation of missing values, especially in Python.
2. **None**: In some programming languages, `None` is used to indicate the absence of a value.
3. **Null or NULL**: Common in SQL databases.
4. **Empty strings (" ")**: Often used in text data.
5. **Special values**: Sometimes a special value, like `-9999` or `9999`, is used to represent missing data.
6. **NA or N/A**: These are often used in spreadsheets and text files to denote missing values.

7. **0 or -1**: Occasionally used in cases where 0 or -1 are not valid values in the context of the data.
8. **Blanks or Whitespaces**: Spaces or tabs might be used to signify missing values in some text data.
9. **Placeholders like "missing" or "unknown"**: Textual placeholders indicating the data is not available.
10. **INF or -INF**: Infinity values, sometimes used to denote missing values.

Before moving forward, it is important to convert the missing values to either 'NaN' or 'nan', as this is what python functions built for missing values will expect to see.

Previous cases only gave a passing treatment of missing data and resulted in dropping the rows containing null values entirely. Here, we will be more nuanced and look at ways that missing values can be replaced appropriately. As always, domain knowledge is crucial in selecting what method to go forward with.

The goal of filling in missing data is to not neccessarily to recover the exact values of the missing values, but rather to ensure that the conclusions of the analysis are not overtly impacted by the missing values.

When dealing with missing data, it is important not to just use generic pandas/Python functions without thinking to fill them in. This can drastically corrupt your analysis.

Instead, a first step is to find the source of the missing data. Why is it missing? and/or how did it go missing? If we can answer these questions, then this will help us in determining how to fill in the missing values.

One way to handle missing values is to simply remove any row which contains a missing value from the dataset. This is known as **complete case analysis**. Complete case analysis is desirable when there are not that many rows with missing values, and we do not lose any critical sub-populations as a results of dropping those rows. For instance, a rule of thumb is that if <1% of the rows are missing, then it might be desirable to do a complete case analysis. It is fast and easy, and does not require extensive statistical knowledge. This also depends on the size of the dataset. If you have 10 million observations, and 1 million within each subpopulation, then removing even 5% of the rows should not impact the analysis too much. On the other hand, if we have only 20 observations, we should try not to remove any rows.

Another approach is to fill in the missing values - this is known as **imputation**. Usually, we use the other observations or rows to fill in the missing values.

Note that a once population method of imputation which you **should not** use is replacing missing values with the mean/median of the available values in the column. The reason for this is that it lowers the sample standard deviation of the resulting column and can bias the resulting analysis.

Some other methods of imputation are given below:

- **Forward/Backward Fill:** Use the next or previous value to fill in the missing value. Forward/backward filling may make sense when there is a relationship between the rows. Example: In a time series dataset where stock prices are recorded daily, you might use forward fill to propagate the last known price to the days when the market data is missing.
- **Interpolation:** Estimate missing values using interpolation techniques. Example: In a climate dataset with daily temperature readings, you could use linear interpolation to estimate the temperature on days when data is missing by averaging the temperatures of the surrounding days.
- **K-Nearest Neighbors (KNN):** The KNN algorithm uses points that are similar to the points with missing data (called nearest neighbors) to impute missing values. Example: In a survey dataset where some participants skipped a few questions, KNN can impute their missing responses by considering the responses of the most similar participants (neighbors).
- **Domain-Specific Imputation:** Depending on the domain, you might use specific rules or external datasets to impute missing values. Example: In an educational dataset where some students' test scores are missing, you might impute the missing scores using the average scores from other tests taken by the same student or from other students in the same class or grade level. This method leverages the relationship between students' performance across different tests or the performance of their peers.
- **Regression/Machine Learning Models:** Use models, such as regression models or Neural Networks to predict and fill in missing values. Example: In a housing dataset where some houses are missing the number of bedrooms, you could use a regression model that predicts the number of bedrooms based on the house's size, location, and price. In a complex dataset with multiple features about customer behavior, a Random Forest model can be used to predict and fill in missing values by leveraging the patterns and interactions between the different features.

7.2.1 Interpolate

The `.interpolate()` function allows us to interpolate numerical values, based on the surrounding values. Interpolation is a method of estimating unknown values based on values that are close to in (in terms of rows). In this case, we know that the rate of change in riders is probably similar to the rate of change on the surrounding days. If we think that it falls between the values of the preceding and following days, and the rows are evenly spaced in time, then we can use the default parameters, which uses linear interpolation. There are a number of other methods of intperolation that can be specified via the `method` parameter. Be sure to read and understand each method carefully before using it. Otherwise, you could just be filling in the values with nonsense.

Parameters of `.interpolate()`

1. **`method`**: Specifies the interpolation method to use.
   - `'linear'` (default): Linear interpolation.
   - `'time'`: Interpolation for time-series data.
   - `'index'`: Uses the index for interpolation.
   - Other methods like `'polynomial'`, `'spline'`, `'barycentric'`, etc. see the documentation.
2. **`axis`**: Specifies the axis along which to interpolate.
   - `0` or `'index'`: Interpolates along the index (default for DataFrame).
   - `1` or `'columns'`: Interpolates along the columns.

Example:

Here's a basic example demonstrating the use of `.interpolate()`:

```python
import pandas as pd
import numpy as np

# Creating a sample DataFrame with NaN values
df = pd.DataFrame({
    'A': [1, np.nan, 3, np.nan, 5],
    'B': [np.nan, 2, np.nan, 4, np.nan]
})

print("Original DataFrame:")
print(df)

# Interpolating missing values
df_interpolated = df.interpolate()

print("\nInterpolated DataFrame:")
print(df_interpolated)
```

```
Original DataFrame:
     A    B
0  1.0  NaN
1  NaN  2.0
2  3.0  NaN
3  NaN  4.0
4  5.0  NaN
```

```
Interpolated DataFrame:
     A    B
0  1.0  NaN
1  2.0  2.0
2  3.0  3.0
3  4.0  4.0
4  5.0  4.0
```

Output:

```
Original DataFrame:
     A    B
0  1.0  NaN
1  NaN  2.0
2  3.0  NaN
3  NaN  4.0
4  5.0  NaN

Interpolated DataFrame:
     A    B
0  1.0  NaN
1  2.0  2.0
2  3.0  3.0
3  4.0  4.0
4  5.0  NaN
```

Interpolation is particularly useful for time-series data and datasets where you want to estimate missing values based on the surrounding data.

7.3 Handling string formats

It is important to print the unique values of a string variable, in order to check for spaces, mixed case and spelling mistakes. If the variable has many entries, we may not be able to check all unique values. In that case, it is a good idea to use `.strip()` and `str.lower()`/`str.upper()` to remove extra spaces and convert all characters to lower case. In the natural language processing lesson, we will learn more about manipulating strings.

7.4 Unique IDs

When generating IDs, it is important to make sure the generation process is idempotent (i.e. the same ID should be generated for each trip no matter how many times you run the script). The idempotency is required because there may be chances that the same trip is input into this tool multiple times. For example, the customer first uploads the data set for the first week of the month (may be for testing purposes, or based on data availability, etc.) and then uploads the data for the entire month. Now if the same trip is assigned different IDs on each run, then it might result in the analytics platform interpreting this as two different trips and this will corrupt the analysis.

7.5 More on `datetime` objects

We have learnt about `datetime` objects in the past. In this case, we covered some new functionalities of these objects. We should almost always convert dates and times to `datetime` objects.

Here's an overview of some new attributes:

1. **`.year`**: Returns the year.
2. **`.month`**: Returns the month (1-12).
3. **`.day`**: Returns the day of the month (1-31).
4. **`.hour`**: Returns the hour (0-23).
5. **`.minute`**: Returns the minute (0-59).
6. **`.second`**: Returns the second (0-59).
7. **`.microsecond`**: Returns the microsecond (0-999999).
8. **`.tzinfo`**: Returns the timezone information.

In addition, you can use the following methods to get week-related information:

1. **`.weekday()`**: Returns the day of the week as an integer, where Monday is 0 and Sunday is 6.
2. **`.isoweekday()`**: Returns the day of the week as an integer, where Monday is 1 and Sunday is 7.
3. **`.isocalendar()`**: Returns a tuple containing the ISO year, ISO week number, and ISO weekday.

Example:

```python
from datetime import datetime

# Create a datetime object
dt = datetime(2023, 7, 5, 14, 30, 0)

# Year, month, day, etc.
print("Year:", dt.year)          # Output: 2023
print("Month:", dt.month)        # Output: 7
print("Day:", dt.day)            # Output: 5
print("Hour:", dt.hour)          # Output: 14
print("Minute:", dt.minute)      # Output: 30
print("Second:", dt.second)      # Output: 0

# Weekday (Monday = 0, Sunday = 6)
print("Weekday:", dt.weekday())  # Output: 2 (Wednesday)

# ISO Weekday (Monday = 1, Sunday = 7)
print("ISO Weekday:", dt.isoweekday())  # Output: 3 (Wednesday)

# ISO Calendar (Year, Week number, Weekday)
iso_calendar = dt.isocalendar()
print("ISO Calendar:", iso_calendar)    # Output: (2023, 27, 3)
```

```
Year: 2023
Month: 7
Day: 5
Hour: 14
Minute: 30
Second: 0
Weekday: 2
ISO Weekday: 3
ISO Calendar: datetime.IsoCalendarDate(year=2023, week=27, weekday=3)
```

Additional operations:

1. **Current Date and Time**:

   ```python
   now = datetime.now()
   print(now)  # Output: current date and time
   ```

   ```
   2024-07-08 17:21:11.925084
   ```

2. **Parsing Dates**:

```python
date_str = "2023-07-05"
parsed_date = datetime.strptime(date_str, "%Y-%m-%d")
print(parsed_date)  # Output: 2023-07-05 00:00:00
```

```
2023-07-05 00:00:00
```

3. **Formatting Dates**:

```python
dt = datetime(2023, 7, 5, 14, 30, 0)
formatted_date = dt.strftime("%Y-%m-%d %H:%M:%S")
print(formatted_date)  # Output: 2023-07-05 14:30:00
```

```
2023-07-05 14:30:00
```

4. **Date Arithmetic**:

```python
dt1 = datetime(2023, 7, 5)
dt2 = datetime(2023, 7, 10)
delta = dt2 - dt1
print(delta)  # Output: 5 days, 0:00:00
```

```
5 days, 0:00:00
```

5. **Handling Time Zones**:

```python
from datetime import timezone, timedelta

utc_dt = datetime(2023, 7, 5, 14, 30, 0, tzinfo=timezone.utc)
local_tz = timezone(timedelta(hours=-5))
local_dt = utc_dt.astimezone(local_tz)
print(local_dt)  # Output: 2023-07-05 09:30:00-05:00
```

```
2023-07-05 09:30:00-05:00
```

7.6 Adding new data

Often, you will have to add data from other datasets, websites or sources to your dataset. For instance, in Case 7, we showed how to get address data using `geopy`.

The function `geolocator.geocode()` can be used to obtain longitude and latitude from a string which contains the address. To learn more about `geopy`, you can follow the functions and tutorials here. In general, you can use `geopy` to obtain coordinates and other location data, given data like address, postal code etc. You can use this in conjunction with the `folium` package to create some nice visualizations.

8 Hypothesis testing I

One of the key problems in data science is assessing whether a pattern you notice is :

1. a pattern inherent in the overall population from which the observations are drawn

OR

2. a spurious pattern specific to the sample of observations you have.

In this case, you learnt a fundamental tool to approach this problem called **statistical hypothesis testing**. You should know how to conduct a hypothesis test, analyze its outcome, and identify its shortcomings.

8.1 Preliminary modules

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
from statsmodels.formula.api import ols
import statsmodels
from scipy import stats
from pingouin import pairwise_tests #this is for performing the pairwise tests
from pingouin import pairwise_ttests #this is for performing the pairwise tests
```

8.2 Hypothesis testing overview

Hypothesis help you rule out *sampling variation* as an explanation for an observed pattern in a data set. It helps us lift a pattern from a sample to a population. The idea is this - every data set we have seen in this course is comprised of a small subset of a population we are interested in learning about. That small subset is called a sample. For instance, in the previous case, we were interested in the population of trips taken on the rentable bikes. We had access to a small subset of the trips taken - this is the sample. Now, there is a small problem with the results of any data analysis. The subset of observations or sample that we have access to is random. We could have easily drawn a different subset and a different subset will give different output in data analysis - that is - different plots and summary statistics. For instance, the average length of a bike trip could be 45 minutes in one subset and 30 minutes in another. The true average bike trip length for the population could be anything - say 39. The purpose of hypothesis testing is to rule out the situation where the sample may not resemble the population. In other words, it is useful to say that the conclusions made in our data analysis are likely to reflect that of the population, and are not due to drawing a bad sample.

In this course, we will mainly focus on saying something about the mean of potentially several populations. In general, the hypothesis testing procedure concerns two hypotheses - the null and alternative hypothesis. These contain mutually exclusive statements about a population.

For example:

$$H_0 : \text{The population trip length is 40 } vs. \ H_a : \text{The population trip length is not 40.}$$

H_0 is called the null hypothesis. The null hypothesis often corresponds to the situation of "no effect" - but not always.

In opposition to the null hypothesis, we define an alternative hypothesis (often indicated with H_1 or H_a) to challenge the null hypothesis. In general, this is the hypothesis of there existing some pattern or effect in the data.

For instance, we may suspect that the average trip length is 40, and we may be concerned about whether our data shows that the trip length is longer than this. This may be of concern for computing bike maintenance costs. In that case, we would define H_a : The population trip length is greater than 40.

The next step in a hypothesis test is to compute a **p-value** using our sample. The p-value measures the evidence against the null hypothesis. It is a number between 0 and 1. The closer to 0 the p-value is, the more evidence there is against the null hypothesis. The p-value can be interpreted as follows: The p-value is the probability that, assuming the null hypothesis is true, we drew a sample that differs from the null hypothesis at least as much as the one at hand.

Let's unpack this statement using our example. First, before computing our p-value we assume that the null hypothesis is true. In our example, we would assume that the mean population trip length is 40 minutes. Then, if the mean population trip length is 40 minutes we can measure how far the sample mean is from 40. Say that our sample mean is 45, and so then we would have observed a sample whose mean is 5 minutes higher than that of the assumed population mean. The p-value is then the probability that we saw a sample whose sample mean is at least 5 minutes away from 40. You will learn in a later course how to compute such a probability. Intuitively, the higher the distance of the sample mean from 40, the lower the p-value. Therefore, if we observe a sample mean far from 40, the p-value will be very close to 0.

We can interpret the p-value as the probability, assuming the mean population trip length is 40, that we drew a sample whose mean differs from 40 at least as much as the one at hand.

Caution

The p-value DOESN'T mean that the probability of H_a is 1-(p - value).

In general, p-values close to 0, as in <0.1 or <0.05 present evidence against the null hypothesis. This threshold can depend on the application.

When doing a formal hypothesis tests, there are two possible outcomes for a test: - (1) We conclude H_0 is false, and say we **reject** H_0. In this case we will conclude that there is statistical evidence for the alternative H_a. - Or (2) we **fail to reject** H_0. In this case, we conclude that there is not enough statistical evidence to say that H_0 is false.

Caution

Notice that in the second case we cannot say that the original hypothesis is true - it might be that we just don't have enough data to rule it out.

In a formal hypothesis test, we define a threshold α, where if the p-value falls below this threshold, then we reject the null hypothesis. In general, less formal cases, one may just compute the p-value and use it to inform decision making, along with other factors.

To summarize:

1. A hypothesis test is used to confirm that a pattern is a feature of the population, and is not due to sampling variation.
2. To conduct a hypothesis test, we define a null and alternative hypothesis.
3. After defining the hypohtheses, we then compute the evidence against the null hypothesis, given by the p-value.
4. If the p-value is small, we reject the null hypothesis and conclude the alternative. Otherwise, we fail to reject the null hypothesis.
5. **You should always interpret the p-value and state your conclusion as the final step in the hypothesis test.**

To elaborate on point 5. - the point of any statistical analysis is not to run the correct code, but to properly choose the analysis procedure and interpret the results appropriately.

Warning

A hypothesis test cannot tell you which scenario is certainly true - we would have to have access to the whole population to know that. It can tell us that things hold with very high certainty, which is generally enough for most situations.

The hypothesis tests introduced in Case 8 concern only testing for the mean of one or more populations. We cover each of those in turn.

8.3 Testing for the mean in a single population

We first cover how to perform a hypothesis test concerning the mean value of a single population. Define the population mean of a single population to be μ. The null hypothesis in this case is in the form of $H_0 \colon \mu = \mu_0$. For instance, above, $\mu_0 = 40$.

We have three different ways to define an alternative hypothesis:

1. $H_a : \mu \neq \mu_0$ (two-sided test)
2. $H_a : \mu > \mu_0$ (one-sided test)
3. $H_a : \mu < \mu_0$ (one-sided test)

The syntax for perfomring the test is given as follows:

```
stats.ttest_1samp(Series you want to test, popmean= mu_0, alternative= specify which of (1-3
```

The p-value is listed in the output, and can be interpreted as instructed above.

8.4 Testing for a difference in mean for two populations

We now cover how to perform a hypothesis test concerning the difference in the mean values between two populations. We would like to test whether two populations have different population means. That is, whether the difference between the population mean of group 1 (μ_1) is different from the population mean of group 2 (μ_2). The hypotheses look like:

$$H_0 : \mu_1 = \mu_2$$

$$H_a : \mu_1 \neq \mu_2$$

The syntax for performing this test is given as follows:

```
stats.ttest_ind(data for group 1 , data for group 2, equal_var=False)
```

The p-value is listed in the output, and can be interpreted as instructed above.

8.5 Testing for a difference in mean for several populations

Lastly, if we would like to perform a hypothesis test for whether or not all the population means are the same when considering $k > 2$ populations, we can do the following.

First, the hypotheses are given by

$$H_0 : \mu_1 = \mu_2 \dots \mu_3 = \mu_k,$$

vs.

$$H_a : \text{At least one of the means } \mu_j \text{ is different from the others.}$$

To test this hypothesis we need an extension of the capabilities of the t - test (which can test only two groups at the same time). This test is called **Analysis of Variance (ANOVA)**.

The syntax is given as follows:

```
# This is code you can fill in to perform this test
mod = ols('quantity of interest ~ grouping variable', data= YOUR_DATAFRAME).fit()
sm.stats.anova_lm(mod, typ=2)
```

The p-value is listed in the output, and can be interpreted as instructed above.

8.6 Errors in hypothesis testing

There are two ways that a test can lead us to an incorrect decision:

1. When H_0 is true and we reject it. This is called **Type 1 Error**. It corresponds to obtaining a **false positive**.
2. When H_0 is false and we do not reject it. This is called **Type 2 Error**. It corresponds to having a **false negative**.

H_0 is true

H_0 is False

Reject H_0

Type I error

Correct Decision (True Positive)

Fail to Reject H_0

Correct Decision (True negative)

Type II error

In general, a Type I error is thought to be more serious and so it is standard practice to control the probability of making a Type I error. In a formal hypothesis test, when the null hypothesis is true, the probability of making a Type I error is the threshold α, also known as the significance level, that we introduced above. Often, we choose our significance level α to be small, e.g., $1\%, 5\%, 10\%$. Lowering the α value (say to 1%) will decrease the probability of making a false positive conclusion, when the null hypothesis is true. Of course, because we control α, we cannot control the Type II error we make. Note that lowering α is not without consequence, as, if the alternative hypothesis is true, then a lower threshold increases the probability of a Type II error.

In summary, it is important to be aware of the two types of error and evaluate the gravity of what making each error would mean for your context. In particular, you should evaluate the consequences of making a Type I error and choose your the threshold α accordingly. Lastly, you should know that there is a trade-of between Type I error and Type II error in a given hypothesis test.

8.7 Misc. Python functions

1. `plt.subplot(rows, cols, curr_plot)` - use this function to position multiple plots in a single figure. For example, `plt.subplot(2, 3, 4)` creates a grid with 2 rows and 3 columns and activates the 4th subplot.
2. `sns.countplot` - creates a barplot.
3. `enumerate` Use this function to get both the index and the value from an iterable in a loop. Example:

```python
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits, start=1):
    print(f"{index}: {fruit}")
```

```
1: apple
2: banana
3: cherry
```

4. `plt.xticks(rotation = 90)` used to rotate the x-axis labels.

9 Hypothesis testing II

10 Natural language processing

11 Linear Regression I

12 Linear Regression II

13 SQL