

Computer Programming with Python

Instructional Annotations, Web Links, & Study References Compiled by: Jordan Cunnagin

Collaboration & Support



We are here to support you — please reach out for help!



Join teacher office hours for additional support.



Submit questions in the Moodle 'Student Help Forum'.

1 Syllabus, Introduction & Review

Variables: Labels to values that can be of any data type. Variable names should be descriptive and follow naming conventions. Good variable names define the purpose of the value they possess. Variable names are case sensitive. Variable names cannot start with numbers. Yes, Python does understand unicode characters. No, do not use them for variable/object names.

Inputs and Outputs: Use `print()` to display text or values, use `input()` to get user input as a string, use formatting or f-strings to format output with precision or expressions.

Expressions: Combinations of values, variables, and operators that can be evaluated. Follow the order of operations (PEMDAS) and use parentheses to group expressions. Use assignment, arithmetic, and shorthand operators to manipulate values.

Functions: Blocks of code that run when called and can accept parameters and return values. Use `def` to define a function, use `return` to exit a function and send a value back to the caller, use `'''docstrings'''` to document a function's purpose and behavior.

Conditional Control: Execute code based on the truth value of a condition. Use `if`, `elif`, and `else` statements to branch the code execution, use boolean logic (*and*, *or*, *not*) to

combine or negate conditions. If one condition in a *if/else* structure is true, other conditions after that are not checked by code.

Looping: Execute the same code block multiple times. Use *while* loops to iterate until a condition is false, use *for* loops to iterate over a sequence or a range of values, use *break* and *continue* to alter the loop flow, use nested loops to combine loops.

Data Structures: Collections of data that can be manipulated as a single entity. Use lists to store ordered, mutable, and heterogeneous data, use tuples to store ordered, immutable, and heterogeneous data, use dictionaries to store unordered, mutable, and key-value paired data, use sets to store unordered, unique, and mutable data.

Debugging: The process of finding and fixing errors in code. Use syntax errors, semantic errors, and runtime errors to identify the type and source of the error, use `print` statements, comments, and breakpoints to test and inspect the code. Semantic errors don't have a display message, but the code does not run as expected.

Reference Links:

- The Ultimate Python Beginner's Handbook:
 - <https://www.freecodecamp.org/news/the-python-guide-for-beginners/>
- Python for Beginners - Full Course (4 Hours Programming Tutorial)
 - <https://www.youtube.com/watch?v=eWRfhZUzrAc>
- Python & Its Basics: A Beginner's Guide:
 - <https://bootcamp.cvn.columbia.edu/blog/python-basics-guide/>
- Learning Python — The Hitchhiker's Guide to Python:
 - <https://docs.python-guide.org/intro/learning/>
- Python Basics:
 - <https://realpython.com/tutorials/basics/>
- Python for Beginners — Learn about variables, collections...
 - <https://towardsdatascience.com/python-for-beginners-basics-7ac6247bb4f4>
- Variables | Introduction to Python:
 - <https://textbooks.cs.ksu.edu/intro-python/01-basic-python/06-variables/>
- Python Variables:

-
- <https://www.geeksforgeeks.org/python-variables/>
 - Basic Input, Output, and String Formatting in Python:
 - <https://realpython.com/python-input-output/>
 - Input and Output in Python:
 - <https://www.ccbp.in/blog/python-tutorial/python-input-and-output>
 - Input and Output Tutorials & Notes:
 - <https://www.geeksforgeeks.org/input-and-output-in-python/>
 - 7. Input and Output — Python Documentation:
 - <https://docs.python.org/3/tutorial/inputoutput.html>
 - Expressions in Python:
 - <https://www.geeksforgeeks.org/expressions-in-python/>
 - Expressions and Operators in Python (3 Min Video)
 - <https://www.youtube.com/watch?v=k5l3GBcGscw>
 - Python Operators:
 - https://www.w3schools.com/python/python_operators.asp
 - Python Functions:
 - https://www.w3schools.com/python/python_functions.asp
 - Python Basics – Conditional Logic and Control Flow:
 - <https://realpython.com/courses/basics-conditional-logic-control-flow/>
 - Python Conditionals and Booleans (16 Min Video)
 - <https://www.youtube.com/watch?v=DZwmZ8Usvnk>
 - Conditional Statements in Python:
 - <https://realpython.com/python-conditional-statements/>
 - Python Loops: A Comprehensive Guide:
 - <https://learnpython.com/blog/python-loop/>
 - Python 'For' Loops:
 - https://www.w3schools.com/python/python_for_loops.asp
 - Python Tutorial for Beginners – Loops and Iterations (10 Min Video)
 - <https://www.youtube.com/watch?v=6iF8Xb7Z3wQ>
 - Loops in Python - For, While, and Nested Loops:
 - <https://www.geeksforgeeks.org/loops-in-python/>
 - Python Data Structures Made Simple – A Beginner's Guide:
 - <https://python.plainenglish.io/python-data-structures-made-simple-a-beginners-guide-138e640a1640>
-

-
- 5. Data Structures — Python Documentation:
 - <https://docs.python.org/3/tutorial/datastructures.html>
 - Python Data Structures:
 - <https://www.geeksforgeeks.org/python-data-structures/>
 - Debugging Tips and Tricks for Python Beginners:
 - <https://python.plainenglish.io/debugging-tips-and-tricks-for-python-beginners-9f61c3a1ecf1>
 - Ultimate Guide to Python Debugging:
 - <https://towardsdatascience.com/ultimate-guide-to-python-debugging-854dea731e1b>
-

2 Function Deep Dive

Logging: A way to debug your code without making changes in code. You can use different levels of logging messages to filter and display information. You need to import and configure the logging module to use it.

Functions: A block of code that runs when called. A function may or may not return a value. A function may accept several or no parameters. A function can be accessed from within another function.

Arguments / Parameters: Values that you feed a function. A parameter is the variable listed inside the parentheses in the function definition. An argument is the value that is sent to the function when it is called. They can be of any data type. Arguments can be variables too. You can use default values for parameters.

Call by Reference: A function call is called by reference. The arguments passed to the functions are a reference to the original values. Any changes to those values in a function will not be reflected to the original variable.

Return Value: A return statement returns a value from the called function. A return value can be any data type. A function can return multiple values. A return statement terminates the flow of the function. A return value can also be boolean. A missing or empty return statement results in a None value of type NoneType being returned.

Scope: Scope rules determine the visibility of names and variables. Python uses the LEGB rule to find a name or variable. You can use global and nonlocal keywords to specify scope. Multiple variables can exist with the same variable name but different scope.

Decorator: A function that takes a function as an argument and returns a function. A decorator is syntactic sugar that allows one to augment functionality of existing functions. You can use the @ symbol to apply a decorator to a function.

Reference Links:

- Logging in Python:
 - <https://realpython.com/python-logging/>
- Logging – How to Log in Python:
 - <https://docs.python.org/3/howto/logging.html>
- Logging in Python for Absolute Beginners in 4 Mins:
 - <https://levelup.gitconnected.com/logging-in-python-for-absolute-beginners-in-4-minutes-c2fd6644fa5b>
- Python Functions:
 - https://www.w3schools.com/python/python_functions.asp
- Function Basics — Intro to Python:
 - <https://textbooks.cs.ksu.edu/intro-python/06-functions/02-basics/>
- Functions:
 - <https://www.pythonforbeginners.com/basics/python-functions-cheat-sheet>
- Python Function Arguments:
 - https://www.w3schools.com/python/gloss_python_function_arguments.asp
- Python Functions - How to Declare and Invoke with Parameters:
 - <https://www.freecodecamp.org/news/python-function-examples-how-to-declare-and-invoke-with-parameters-2/>
- Python for Beginners – Functions, Parameters, and Arguments:
 - <https://hackernoon.com/python-for-beginners-functions-parameters-and-arguments-n43c33mz>
- Deep Dive into Parameters and Arguments in Python:
 - <https://www.geeksforgeeks.org/deep-dive-into-parameters-and-arguments-in-python/>
- What is a Call by Value and Call by Reference in Python:
 - <https://www.scaler.com/topics/call-by-value-and-call-by-reference-in-python/>

-
- Call by Reference and Call by Value:
 - <https://www.geeksforgeeks.org/is-python-call-by-reference-or-call-by-value/>
 - Python return Statement – Usage and Best Practices:
 - <https://realpython.com/python-return-statement/>
 - Mastering Python return Values:
 - <https://skillapp.co/blog/mastering-python-return-values-a-comprehensive-guide-for-beginners/>
 - Python Scope:
 - https://www.w3schools.com/PYTHON/python_scope.asp
 - Understanding Python Scope:
 - <https://lset.uk/learning-resources/understanding-python-scope-a-comprehensive-guide-for-beginners/>
 - Python Decorators Explained for Beginners:
 - <https://www.freecodecamp.org/news/python-decorators-explained/>
 - Python Decorators Introduction:
 - <https://pythonbasics.org/decorators/>
-

3 Recursion, Again

Recursion is a concept in computer science where a **function calls itself** to solve a problem. It's *like* a loop, but instead of repeating a ~~block of code~~, the function repeats itself with a slightly different argument each time. Here's how it works:

1) Base Case: This is the *simplest* form of the problem that can be solved directly. It's like the stopping point for the recursion. For example, if we're counting down from 10, the base case could be when we reach 0.

2) Recursive Case: This is where the function calls itself, but with a smaller or simpler version of the original problem. Using the countdown example, the recursive case would be – to count down from $n-1$, where n is the current number.

3) Progress towards the Base Case: Each time the function calls itself, it should be making progress towards reaching the base case. In the countdown, we're getting closer to 0 with *each* recursive call.

Recursion and iteration are both programming concepts used to repeat *certain* operations, but they work in different ways:

Recursion:

A function is said to be recursive when it calls itself.

Recursion involves breaking down a problem into smaller and simpler sub-problems.

It requires a base case to stop the recursion.

Recursive functions have to keep the function records in memory and jump from one memory address to another to be invoked to pass parameters and return values.

Recursion can lead to elegant and concise code, but it can also lead to higher time complexity and stack overflow errors if not handled properly.

Iteration:

Iteration involves the use of loops (like for and while) to repeat a block of code.

The time complexity of iteration can be found by finding the number of cycles being repeated inside the loop.

Iterative solutions generally have a lower time complexity and are more efficient in terms of memory usage compared to recursive solutions.

However, iterative solutions might result in more lines of code compared to recursive solutions.

In summary, the choice *between* recursion and iteration often depends on the specific problem at hand, the efficiency requirements, and the need for code readability and maintainability.

Dynamic Programming: An optimization technique that builds on the concept of recursion. It uses a data structure to store the results of previous computations, which can

then be reused in later computations, thereby saving time at the expense of using more space. This technique is particularly useful when the problem involves overlapping subproblems.

Exception handling: A mechanism in programming to handle runtime errors. It prevents the program from crashing by providing a robust way to handle exceptions (*errors*) that may occur during the execution of the program. The structure typically involves:


Try Block: Contains the code that may raise an exception.

Except Block: Defines what to do when a specific exception is raised.

Else Block: Specifies what to do if no exception is raised.

Finally Block: Contains code that is always executed, whether an exception was raised or not.

Reference Links:

- Recursion in Python: An Introduction:
 - <https://realpython.com/python-recursion/>
- The Python 'return' Statement: Usage & Best Practices:
 - <https://realpython.com/python-return-statement/>
- Thinking Recursively in Python:
 - <https://realpython.com/python-thinking-recursively/>
- How to Implement a Python 'stack':
 - <https://realpython.com/how-to-implement-python-stack/>
- Namespaces & Scope in Python:
 - <https://realpython.com/python-namespaces-scope/>
- Recursion in Python:
 - <https://www.geeksforgeeks.org/recursion-in-python/>
- Python | Recursion Explained (video)
 -  Python: RECURSION Explained
- Recursive Functions in Python: Examples, Tips & Best Practices:

-
- <https://diveintopython.org/learn/functions/recursion>
 - Recursive Functions – Python Numerical Methods:
 - <https://pythonnumericalmethods.berkeley.edu/notebooks/chapter06.01-Recursive-Functions.html>
 - Recursive Functions – Simply Explained | Python in Plain English (need account)
 - <https://python.plainenglish.io/recursive-functions-simply-explained-7b13552f1393>
 - Python Recursion Example – Recursive Functions:
 - <https://www.askpython.com/python/python-recursion-function>
 - Recursion in Python Explained for Beginners (need account)
 - <https://python.plainenglish.io/recursion-in-python-explained-for-beginners-124b279f689c>
 - Recursion (computer science) — Wikipedia:
 - https://en.wikipedia.org/wiki/Recursion_%28computer_science%29
 - Intro to Recursion – Data Structure & Algorithm Tutorials:
 - <https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/>
 - What is 'Base Case' in Recursion?
 - <https://www.geeksforgeeks.org/what-is-base-case-in-recursion/>
 - Understanding Recursion in Python | A Step-by-Step Guide (need account)
 - <https://levelup.gitconnected.com/understanding-recursion-in-python-a-step-by-step-guide-2b4eb777f6a0>
 - Recursive Practice Problems with Solutions:
 - <https://www.geeksforgeeks.org/recursion-practice-problems-solutions/>
 - Reverse a 'Stack' Using Recursion:
 - <https://www.geeksforgeeks.org/reverse-a-stack-using-recursion/>
 - Sum of Natural Numbers Using Recursion:
 - <https://www.geeksforgeeks.org/sum-of-natural-numbers-using-recursion/>
 - Sort a 'Stack' Using Recursion:
 - <https://www.geeksforgeeks.org/sort-a-stack-using-recursion/>
 - Difference Between Recursion & Iteration:
 - <https://www.geeksforgeeks.org/difference-between-recursion-and-iteration/>
 - Recursion vs Iteration – 10 Differences & When to Use:
-

-
- <https://favtutor.com/blogs/recursion-vs-iteration>
 - Difference Between Recursion & Iteration:
 - <https://www.interviewkickstart.com/learn/difference-between-recursion-and-iteration>
 - Iteration vs Recursion PDF:
 - <https://turing.plymouth.edu/~zshen/Webfiles/notes/CS322/loopRecursionACT032021.pdf>
 - Programming Fundamentals - Recursion vs Iteration:
 - https://en.wikibooks.org/wiki/Programming_Fundamentals/Recursion_vs_Iteration
 - Recursion vs Iteration PDF:
 - <https://web.mit.edu/6.102/www/sp23/classes/11-recursive-data-types/recursion-and-iteration-review.html>
 - Big O Notation & Its Examples in Python:
 - <https://medium.com/analytics-vidhya/big-o-notations-and-its-examples-in-python-9d7e3c1ef09d>
 - Beginner's Guide to Big O Notation:
 - <https://www.freecodecamp.org/news/my-first-foray-into-technology-c5b6e83fe8f1/>
 - Dynamic Programming:
 - <https://www.geeksforgeeks.org/dynamic-programming/>
 - Dynamic Programming with Examples:
 - <https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/>
 - Dynamic Programming Stanford PDF:
 - <https://web.stanford.edu/class/cs97si/04-dynamic-programming.pdf>
 - Try and Except in Python:
 - <https://pythonbasics.org/try-except/>
 - Python Try Except:
 - <https://www.geeksforgeeks.org/python-try-except/>
 - Python Try and Except Statements - How to Handle Exceptions in Python:
 - <https://www.freecodecamp.org/news/python-try-and-except-statements-how-to-handle-exceptions-in-python/>
-

-
- Python Try Except: Best Practices and Examples:
 - <https://python.land/deep-dives/python-try-except>
 - Try, Except, else and Finally in Python:
 - <https://www.geeksforgeeks.org/try-except-else-and-finally-in-python/>
 - How to Handle Errors in Python:
 - <https://www.freecodecamp.org/news/how-to-handle-errors-in-python/>
 - How to Catch Multiple Exceptions in Python:
 - <https://realpython.com/python-catch-multiple-exceptions/>
 - What are Try/Except Statements in Python?
 - <https://betterprogramming.pub/how-to-start-using-try-statements-in-python-5043fe69058d>
 - Python 'finally' Keyword:
 - https://www.w3schools.com/python/ref_keyword_finally.asp
 - 'finally' Keyword in Python:
 - <https://www.geeksforgeeks.org/finally-keyword-in-python/>
 - Python 'finally' Keyword & Uses:
 - <https://www.pythonforbeginners.com/basics/python-finally-keyword-and-use>
[S](#)
 - Using 'return' With 'try' and 'finally' Blocks in Python:
 - <https://realpython.com/lessons/python-return-try-finally/>
 - How to Handle Exceptions in Python | A Detailed Visual Intro:
 - <https://www.freecodecamp.org/news/exception-handling-python/>
 - Python Exceptions and Errors:
 - <https://pynative.com/python-exceptions/>
 - Exception Handling in Python - Basic to Advanced:
 - <https://towardsdatascience.com/exception-handling-in-python-from-basic-to-advanced-then-tricks-9b495619730a>
 - Working with Carriage Return (/r) in Python:
 - <https://www.pythonpool.com/carriage-return-python/>
 - Understanding Python Carriage Return | A Comprehensive Guide:
 - <https://alexlynx.com/python-carriage-return/>
 - Python Exception Handling:
 - <https://www.geeksforgeeks.org/python-exception-handling/>
-

-
- Python Try Except:
 - https://www.w3schools.com/python/python_try_except.asp
 - Python Code Visualizer:
 - <https://pythontutor.com/visualize.html>
-

4 Object Oriented Programming (OOP)

This class covered *several* fundamental concepts in object-oriented programming (OOP):

Composition: This concept involves building complex objects by combining simpler ones. Composition allows objects to interact in a modular and flexible manner, promoting code reusability and maintainability.

Encapsulation: Encapsulation refers to the practice of bundling data (attributes) and methods (functions) that operate on the data into a single unit, known as a class. This hides the internal workings of an object, exposing only necessary interfaces for interacting with it. It enhances data protection and reduces unwanted interference.

Public vs. Private: In OOP, attributes and methods within a class can be designated as either public or private. Public members are accessible from outside the class, while private members are only accessible within the class itself. This enforces data encapsulation and controlled access to class internals.

Dunder Methods: Dunder methods (double underscore methods, also known as magic or special methods) are predefined methods in Python classes that enable customizing various behaviors. Examples include `__init__` for object initialization, `__str__` for human-readable string representation, and `__add__` for defining addition behavior.

Inheritance: Inheritance is a mechanism where a new class (subclass or derived class) is created based on an existing class (superclass or base class). The subclass inherits

attributes and methods from the superclass, allowing code reuse and hierarchical organization of classes. It promotes the "is-a" relationship between classes.

These concepts collectively form the foundation of object-oriented programming, enabling efficient code organization, reusability, and extensibility while promoting clean and maintainable code practices.

Reference links:

Some helpful supporting links for these concepts are:

- Short & sweet Dunder method info:
 - <https://dev.to/somespi/python-dunder-functions-what-they-are-and-how-to-use-them-1lom>
- Dunder info with code to follow along:Short & sweet Dunder method info:
 - <https://builtin.com/data-science/dunder-methods-python>
- A complete list of dunder methods:
 - <https://mathspp.com/blog/pydons/dunder-methods>
- A guide to inheritance:
 - <https://codefather.tech/blog/python-class-inheritance/>
- Python Class Constructors: Control Your Object Instantiation:
 - <https://realpython.com/python-class-constructor/>
- OOP Concepts for Beginners: What is Composition?
 - <https://stackify.com/oop-concepts-composition/>
- Inheritance and Composition in Python:
 - <https://www.geeksforgeeks.org/inheritance-and-composition-in-python/>
- Composing with Classes:
 - <https://realpython.com/lessons/compose-with-python-class/>
- Encapsulation in Python:
 - <https://www.geeksforgeeks.org/encapsulation-in-python/>
- Encapsulation in Python | A Comprehensive Guide:
 - <https://blog.stackademic.com/encapsulation-in-python-a-comprehensive-guide-with-class-methods-and-static-methods-5e336d5354d5>
- Getters & Setters: Manage Attributes in Python:

-
- <https://realpython.com/python-getter-setter/>
 - Getter & Setter in Python:
 - <https://www.geeksforgeeks.org/getter-and-setter-in-python/>
 - Public, Protected, Private Members in OOP:
 - <https://www.tutorialsteacher.com/python/public-private-protected-modifiers>
 - Access Modifiers in Python: Public, Private, Protected:
 - <https://www.geeksforgeeks.org/access-modifiers-in-python-public-private-and-protected/>

5 Collections of Objects

In this week's class, we delved into *various* advanced concepts in object-oriented programming (OOP) and data structures. Here's a recap of what we covered:

Types of Inheritance: We explored the different types of inheritance, including single inheritance (subclasses inherit from a single superclass), multiple inheritance (subclasses inherit from multiple superclasses), and multilevel inheritance (subclasses inherit in a hierarchical chain).

- **Multiple Inheritance:** We examined the challenges and benefits of multiple inheritance, where a subclass inherits attributes and methods from more than one superclass. This allows for complex class hierarchies but requires careful management of potential conflicts.
- **Multilevel Inheritance:** This topic introduced us to multilevel inheritance, where a subclass inherits from another subclass. This enables a multi-tiered hierarchy and facilitates code organization.
- **Method Resolution Order (MRO):** We delved into how Python handles the method resolution order in cases of multiple inheritance. Understanding MRO is crucial to comprehend the sequence in which methods are invoked from different parent classes.

Polymorphism: We explored polymorphism, which allows objects of different classes to be treated as objects of a common superclass. Polymorphism simplifies code by enabling a single interface for different classes.

Method Overriding: This concept involves modifying or extending a method inherited from a superclass in a subclass. It allows customization of behavior while maintaining the original method's signature.

Advanced Object Composition: We looked into more advanced techniques for object composition, building complex objects by combining simpler ones. This promotes modularity and code reusability.

Queue & Stack: We introduced the concepts of queues (FIFO - First In, First Out) and stacks (LIFO - Last In, First Out) as essential data structures for managing collections of elements in specific ways.

Super vs. Object Init: We compared the use of `super()` and `object.__init__()` methods in Python. `super()` facilitates calling methods from parent classes, while `object.__init__()` is used to directly call the constructor of the object's parent class.

Throughout the week, we deepened our understanding of OOP principles and their practical applications. These concepts are *foundational* for creating efficient, modular, and extensible code structures.

Reference links:

Session 1

- Multiple inheritance info:
 - <https://www.pythontutorial.net/python-oop/python-multiple-inheritance/>
 - <https://www.scaler.com/topics/multiple-inheritance-in-python/>
 - <https://www.geeksforgeeks.org/multiple-inheritance-in-python/>
 - <https://realpython.com/lessons/python-multiple-inheritance/>
 - <https://python.plainenglish.io/multiple-inheritance-in-python-a-basic-guide-with-examples-124ee08e7f62>

Python's built-in **super()** method returns a temporary object of the superclass to help you access its methods. Its purpose is to avoid using the base class name explicitly. It also enables your class to inherit from multiple base classes.

- Understanding Python's `super()` constructor for inheritance:

-
- <https://blog.finxter.com/python-super-function/> (exit out of pop up to view)
 - <https://medium.com/the-kickstarter/multiple-inheritance-in-python-84258756e04c>
 - <https://www.w3docs.com/snippets/python/how-does-pythons-super-work-with-multiple-inheritance.html>
 - https://aviadr1.github.io/learn-advanced-python/06_multiple_inheritance_and_super/multiple_inheritance_and_super.html
 - <https://youtu.be/AhS7LbHo3sw> (12 minute visual explanation of `super()`)
 - Multilevel inheritance guide:
 - <https://guidingcode.com/multi-level-inheritance-in-python/>
 - <https://www.codingninjas.com/studio/library/multilevel-inheritance-in-python>
 - What is the difference between multilevel and multiple inheritance?
 - <https://thisvsthat.io/multilevel-inheritance-vs-multiple-inheritance>
 - OOP inheritance types guide:
 - <https://learnpainless.com/inheritance-in-python/>
 - In-depth method resolution order (MRO) info:
 - <https://dotnettutorials.net/lesson/method-resolution-order-python/>

The word **polymorphism** means '*many forms*', and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

- Polymorphism in Python:
 - <https://pynative.com/python-polymorphism/>
- Overloading functions info:
 - <https://stackabuse.com/overloading-functions-and-operators-in-python/>
- Method overriding in Python:
 - <https://www.scaler.com/topics/method-overriding-in-python/>

Session 2

A **queue** in programming terms is an Abstract Data Type that stores the order in which items were added to the structure but only allows additions to the end of the queue while allowing removals from the front of the queue. In doing so, this follows a First-In First-Out (FIFO) data structure.

- Queues in Python:

-
- <https://towardsdatascience.com/a-complete-guide-to-queues-in-python-cd2baf310ad4>
 - (very good article, need free account to read)
 - Queue implementation beginner guide:
 - <https://python-programs.com/how-to-use-queue-a-beginners-guide/>
 - Working With Queue Data Structure in Python:
 - <https://www.simplilearn.com/tutorials/python-tutorial/queue-in-python#:~:text=Queue%20in%20Python%20is%20a,come%2Dfirst%2Dserve%20basis>
 - Queue in Python:
 - <https://www.geeksforgeeks.org/queue-in-python/>
 - Python's dequeue | Implement Queues and Stacks:
 - <https://realpython.com/python-deque/>
 - Queue Python Doc:
 - <https://docs.python.org/3/library/queue.html>

The **stack** data structure consists of a linear collection. The main distinguishing feature of a stack, however, is how data is stored and removed. Stacks use a *Last-In-First-Out* (LIFO) approach, where items are “popped” (retrieved and simultaneously removed) from newest to oldest.

- Understanding stacks in Python:
 - <https://entri.app/blog/beginners-guide-to-understand-stacks-in-python/>
- Super vs init:
 - <https://thelearning.dev/python-super-vs-baseinit-method>
- Super() with __init__() methods:
 - https://www.i2tutorials.com/python-super-with-__init__-methods/
- Stack and Queues in Python:
 - <https://www.geeksforgeeks.org/stack-and-queues-in-python/>
- Method Resolution Order in Python Inheritance:

-
- <https://www.geeksforgeeks.org/method-resolution-order-in-python-inheritance/>
 - What is `__mro__` in Python?
 - <https://towardsdev.com/what-is-mro-in-python-87c0c07f0c5a>

6 Graphs & Trees

During this class, we delved into **essential concepts** in computer programming using Python, focusing on *data structures* and *searching* algorithms. Here's a snapshot of what we learned:

Graphs: We explored graphs as a versatile data structure comprising nodes (vertices) and edges. Graphs model relationships between various entities and can be directed or undirected, cyclic or acyclic.

Trees: Trees are hierarchical data structures with a single root node and child nodes connected by edges. They find applications in representing hierarchical relationships and organizing data efficiently.

Binary Trees: Binary trees are a special type of tree where each node has at most two children: left and right. Binary trees are used in various algorithms and are the foundation of binary search trees.

Traversals: We discussed tree traversal methods that enable visiting all nodes in a tree. In-order, pre-order, and post-order traversals help extract and process data in a systematic manner.

Searching: Searching algorithms allow us to locate specific elements within data structures. Two common strategies are Breadth-First Search (BFS) and Depth-First Search (DFS).

-
- **BFS (Breadth-First Search):** BFS systematically explores nodes level by level, starting from the root. It's useful for finding shortest paths and exploring neighbors in graph-like structures.
 - **DFS (Depth-First Search):** DFS explores as far as possible along a branch before backtracking. It's helpful for tasks like maze-solving, topological sorting, and identifying connected components.

Throughout the class, we gained insights into the core principles of graphs, trees, and searching algorithms. These concepts are foundational for solving a wide range of problems in computer science, from data organization to pathfinding and network analysis.

Reference links:

Session 1

- Networkx intro doc:
 - <https://networkx.org/documentation/stable/reference/introduction.html>
- Networkx graphing doc:
 - <https://networkx.org/documentation/networkx-1.0/tutorial/tutorial.html>
- Networkx graph guide:
 - <https://www.datacamp.com/tutorial/networkx-python-graph-tutorial>
- Great beginner guide to graphs in Networkx:
 - <https://www.toptal.com/data-science/graph-data-science-python-networkx>
- Defaultdict tutorial & info:
 - <https://blog.devgenius.io/how-to-use-defaultdict-in-python-a-complete-tutorial-8886d6d71c0f>
 - <https://datagy.io/python-defaultdict/>
- Difference between graphs & trees:
 - <https://www.geeksforgeeks.org/difference-between-graph-and-tree/>

Session 2

- Tree traversals in Python:
 - <https://famtutor.com/blogs/tree-traversal-python-with-recursion>

-
- Tree Traversal Techniques in Python:
 - <https://www.geeksforgeeks.org/tree-traversal-techniques-in-python/>
 - Tree Traversal | Inorder, Preorder, Postorder:
 - <https://www.programiz.com/dsa/tree-traversal>
 - In-order traversal info:
 - <https://www.pythonforbeginners.com/data-structures/in-order-tree-traversal-in-python>
 - Post-order traversal info:
 - <https://www.pythonforbeginners.com/data-structures/postorder-tree-traversal-algorithm-in-python>
 - Postorder Traversal of Binary Tree:
 - <https://www.geeksforgeeks.org/postorder-traversal-of-binary-tree/>
 - Postorder Tree Traversal in Python:
 - <https://www.askpython.com/python/examples/postorder-tree-traversal-in-python>
 - Pre-order traversal info:
 - <https://www.pythonforbeginners.com/data-structures/preorder-tree-traversal-algorithm-in-python>
 - Iterative Preorder Traversal:
 - <https://www.geeksforgeeks.org/iterative-preorder-traversal/>
 - Preorder Tree Traversal in Python:
 - <https://www.askpython.com/python/examples/preorder-tree-traversal>
 - DFS of a Tree:
 - <https://www.geeksforgeeks.org/dfs-traversal-of-a-tree-using-recursion/>
 - Depth First Traversal in Python:
 - <https://www.pythonforbeginners.com/data-structures/depth-first-traversal-in-python>
 - Beginner's guide to BFS & DFS:
 - <https://leetcode.com/discuss/study-guide/1072548/A-Beginners-guide-to-BFS-and-DFS>
 - Depth-First Search guide with code:

-
- <https://favtutor.com/blogs/depth-first-search-python>
 - Breadth-First Search guide with code:
 - <https://favtutor.com/blogs/breadth-first-search-python>
 - Traversal Search (video)
 - <https://www.youtube.com/watch?v=WbzNRTTrX0g&list=PLhQjrBD2T381PopUTYtMSstgk-hsTGkVm&index=2>

7 Advanced File Handling

Session 1

In this class, we explored *fundamental concepts* related to **file handling**, reading, writing, and CSV (Comma-Separated Values) file manipulation. Here's a concise overview of what we covered:

Opening Files For Use: We learned how to open files in Python using the `open()` function. This function allows us to specify the file path and the mode in which the file will be accessed (read, write, append, etc.).

Reading Files: Reading files involves extracting data from them. We discussed methods like `read()`, `readline()`, and `readlines()` that enable us to read the file's content in different ways.

Writing Files: Writing files involves creating or overwriting content in a file. We explored how to use the `write()` method to add data to a file, and we covered appending data using the `append()` mode.

Cursors: File cursors (or pointers) keep track of the current position in a file. When reading or writing, the cursor indicates where the next operation will occur. We learned how to manipulate cursors using methods like `seek()`.

CSV Files: CSV files are a common format for storing tabular data. We delved into how to read and write CSV files using the `csv` module in Python. This module offers convenient methods for handling CSV-specific formatting.

Understanding file handling is **crucial** for interacting with external data and resources in programming. The ability to *efficiently* read from and write to files, along with knowledge of CSV file manipulation, equips us with the skills to handle data storage and retrieval effectively within Python programs.

Reference links:

- Opening files in Python (A long read, but explains concept very well):
 - <https://marketsplash.com/tutorials/python/how-to-open-a-file-in-python/>
- File opening in Python beginner's guide:
 - <https://awstip.com/opening-a-file-in-python-a-beginners-guide-129c70e85e63>
- File handling syntax reference guide:
 - https://www.w3schools.com/python/python_file_handling.asp
- File handling - easy beginner guide:
 - <https://dev.to/tusharsrivastava/handling-text-files-in-python-an-easy-guide-for-beginners-4egd>
- Understanding context managers:
 - <https://towardsdatascience.com/understanding-python-context-managers-for-absolute-beginners-4873b6249f16>
- Carriage return info guide:
 - <https://alexlynx.com/python-carriage-return/>
- How to read a file in Python:
 - <https://www.geeksforgeeks.org/how-to-read-from-a-file-in-python/>
- Advanced file handling:
 - https://www.w3schools.com/python/python_ref_file.asp
- Tutorial on handling files:
 - https://www.tutorialspoint.com/python/python_files_io.htm#
- The difference between read() and readlines()
 - <https://www.cstack.org/difference-r-w-python-example/#:~:text=Difference%20between%20r%2B%20and%20w%2B%20in%20Python%20We,is%20used%20to%20open%20the%20file%20for%20writing>
- How to write to file in Python:

-
- <https://learnpython.com/blog/write-to-file-python/>
 - Python file seek(), tell(), read(): move file cursor position:
 - <https://pynative.com/python-file-seek/>
 - Tutorial: How to Easily Read Files in Python (Text, CSV, JSON)
 - <https://www.dataquest.io/blog/read-file-python/>
 - This is good to save for reference when encountering different file types
 - CSV file handling:
 - <https://diveintopython.org/learn/file-handling/csv>
 - CSV Python documentation:
 - <https://docs.python.org/3/library/csv.html>
 - Guide to CSV module:
 - <https://learnpython.com/blog/guide-to-the-python-csv-module/>
 - String format:
 - <https://www.geeksforgeeks.org/string-alignment-in-python-f-string/>
 - Read & write simultaneously same file:
 - <https://stackoverflow.com/questions/48837560/reading-and-writing-to-the-same-file-simultaneously-in-python>
 - W+ info:
 - <https://stackoverflow.com/questions/48837560/reading-and-writing-to-the-same-file-simultaneously-in-python>

Session 2

This reference encapsulates the key concepts covered in our class, highlighting the significance and practical applications of each topic:

Exception Handling: Exception handling is a crucial programming practice that enables us to gracefully manage and recover from errors during program execution. By using `'try'`, `'except'`, `'finally'`, and `'raise'` statements, we can identify, handle, and even customize error messages. Exception handling enhances code robustness and prevents unexpected crashes.

Reading Files as Classes: Treating files as classes provides a structured approach to file manipulation. By creating custom classes for file reading, we encapsulate file operations

and logic, making code modular and reusable. This technique promotes cleaner code organization and allows us to manage file-specific functionality effectively.

Using JSON Files: JSON (JavaScript Object Notation) is a lightweight data interchange format commonly used for storing and exchanging structured data. In Python, the '`json`' module allows us to serialize (convert objects to JSON) and deserialize (convert JSON to objects) data seamlessly. JSON is ideal for configuration files, APIs, and data storage, offering a human-readable and machine-friendly format.

By mastering these concepts, we are equipped to develop more robust and organized Python programs. Exception handling safeguards our code, reading files as classes enhances code structure, and understanding JSON usage empowers us to work with data efficiently. Integrating these skills enhances our ability to create reliable and versatile applications.

Reference links:

- Python exceptions: A beginner's guide:
 - <https://realpython.com/python-exceptions/>

NOTE: Python treats all text in a text file as a string. If you read a number from a file and you want to carry out arithmetic operations, convert it to float using the `float()` function or integer using the `int()` function.

- How to Catch, Raise, and Print a Python Exception:
 - <https://www.coursera.org/tutorials/python-exception>
- Handling Exception operations:
 - <https://www.codingdrills.com/tutorial/python-tutorial/python-exceptions-file>
- Python library - built-in exception doc:
 - <https://docs.python.org/3/library/exceptions.html>
- Chapter 6: Files and exceptions - *very good* follow along!
 - https://www.softcover.io/read/e4cd0fd9/conversational-python/ch6_files_exceptions
- Handling permission error:
 - <https://www.askpython.com/python/examples/handling-error-13-permission-denied>

-
- Python - files and exceptions:
 - <https://dev.to/ahmedgouda/python-files-and-exceptions-2f5i>
 - Python Exception Handling w/ File Operations:
 - <http://www.geekswithgeeks.com/python/python-exception-handling-with-file-operations>
 - Python OS:
 - <https://www.geeksforgeeks.org/python-os-chmod-method/>
 - open() Python doc - everything you need to know
 - <https://docs.python.org/3/library/functions.html#open>
 - Python 'with open' statement guide:
 - <https://codefather.tech/blog/python-with-open/>
 - (A few overlay ads to close, but *realllllly* good explanation)
 - Python io module:
 - <https://www.askpython.com/python-modules/python-io-module>
 - Python doc for io module:
 - <https://docs.python.org/3/library/io.html#module-io>
 - *args and **kwargs in Python:
 - <https://www.geeksforgeeks.org/args-kwargs-python/>
 - Python's *args and **kwargs: A quick and easy guide with examples
 - <https://thecodecadence.medium.com/pythons-args-and-kwargs-a-quick-and-easy-guide-with-examples-b4e223f5ad37>
 - Add logic to your code (conditional logic video):
 - <https://realpython.com/lessons/add-logic-to-your-code/>
 - Logical operators in Python with examples:
 - <https://www.geeksforgeeks.org/python-logical-operators/>
 - Structuring your project (good all around info):
 - <https://docs.python-guide.org/writing/structure/>
 - Working with CSV files:
 - <https://www.geeksforgeeks.org/working-csv-files-python/>
 - Working with OOP guide:
 - <https://pythonguide.readthedocs.io/en/latest/python/oops.html>
 - Working with JSON data in Python:
 - <https://realpython.com/python-json/>

-
- What is JSON and why use it?
 - https://www.w3schools.com/whatis/whatis_json.asp
 - Working with JSON files:
 - <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON>
 - Python JSON doc:
 - <https://docs.python.org/3/library/json.html>

8 NumPy (Numerical Python)



Session 1

NumPy Introduction: NumPy is a crucial library for scientific computing and data analysis in Python. It introduces the concept of arrays, which are multi-dimensional data structures that can hold elements of the same data type. NumPy provides an extensive collection of functions and operations to perform complex mathematical computations efficiently on these arrays.

Here are some basic NumPy routines that are commonly used for array creation, manipulation, and computations:

Creating Arrays:

```
import numpy as np
```

```
# Create an array from a list
arr = np.array([1, 2, 3, 4, 5])
```

```
# Create an array of zeros
zeros_arr = np.zeros(5)
```

```
# Create an array of ones
```

```
ones_arr = np.ones(5)
```

```
# Create a range of values
range_arr = np.arange(0, 10, 2)
```

```
# Create a 2D identity matrix
identity_matrix = np.eye(3)
```

Array Operations:

```
import numpy as np
```

```
a = np.array([1, 2, 3])
```

```
b = np.array([4, 5, 6])
```

```
# Element-wise addition
sum_result = a + b
```

```
# Element-wise multiplication
product_result = a * b
```

# Dot product dot_product = np.dot(a, b)	std_dev = np.std(data)	
# Transpose of a matrix transposed_matrix = np.transpose(matrix)	# Variance of the array variance = np.var(data)	arr = np.array([[1, 2, 3], [4, 5, 6]])
Statistical Functions: import numpy as np data = np.array([10, 20, 30, 40, 50])	Broadcasting: import numpy as np array = np.array([1, 2, 3]) scalar = 2	# Reshape array to a different shape reshaped_arr = arr.reshape(3, 2)
# Mean of the array mean = np.mean(data)	# Element-wise multiplication with broadcasting result = array * scalar	Indexing and Slicing: import numpy as np arr = np.array([10, 20, 30, 40, 50]) # Access an element element = arr[2]
# Median of the array median = np.median(data)	Reshaping Arrays: import numpy as np	# Slice the array sliced_array = arr[1:4]
# Standard deviation of the array		

These are just a few basic routines that showcase the power and versatility of NumPy. As you explore further, you'll discover a wealth of additional functions and methods that enable you to manipulate and analyze data efficiently using NumPy arrays.

- NumPy User Guide PDF:
 - <https://numpy.org/doc/1.18/numpy-user.pdf>
- NumPy Cheat Sheet PDF:
 - <http://www.cheat-sheets.org/saved-copy/numpy-cheat-sheet.20210604.pdf>

Reading CSV Files with NumPy: NumPy simplifies the process of reading CSV (Comma-Separated Values) files. By using the `genfromtxt()` function, you can convert CSV data into NumPy arrays. This is particularly useful because NumPy arrays are suitable for handling large datasets and performing various numerical operations on the data. `np.loadtxt()` is suitable for reading relatively simple text files like CSVs. If you need more

advanced features, such as handling mixed data types, missing values, or column names, `np.genfromtxt()` provides more flexibility.

Statistics: NumPy facilitates statistical analysis by offering a wide range of functions to compute various statistical metrics. These functions include calculating the mean, median, standard deviation, variance, percentiles, and more. NumPy's ability to handle arrays efficiently makes it an ideal choice for processing and analyzing datasets in the field of statistics.

Here's an explanation of some common statistical functions in NumPy:

- **np.mean():** The `np.mean()` function computes the arithmetic mean (average) of the values in an array. It provides a measure of central tendency and is calculated by summing all values and dividing by the number of values.
- **np.median():** The `np.median()` function calculates the median, which is the middle value in a sorted dataset. It's a robust measure of central tendency that's less sensitive to outliers compared to the mean.
- **np.std():** The `np.std()` function calculates the standard deviation, which measures the spread or dispersion of data around the mean. A higher standard deviation indicates greater variability in the data.
- **np.var():** The `np.var()` function computes the variance, which is the average of the squared differences from the mean. Variance indicates how much the data points vary from the mean.
- **np.percentile():** The `np.percentile()` function calculates a specified percentile value of a dataset. Percentiles indicate the relative standing of a value within the dataset. For example, the 75th percentile is the value below which 75% of the data falls.
- **np.min()** and **np.max():** These functions return the minimum and maximum values in an array, respectively.
- **np.sum():** The `np.sum()` function calculates the sum of all values in an array. It's a versatile function that can also be used with specified axis arguments for summing along specific dimensions.
- **np.prod():** The `np.prod()` function computes the product of all values in an array.

-
- **np.corrcoef():** The np.corrcoef() function calculates the Pearson correlation coefficient, which measures the linear relationship between two variables. It's commonly used to quantify how well two variables change together.
 - **np.histogram():** The np.histogram() function computes a histogram of data, which represents the frequency distribution of values in intervals (bins). It's useful for visualizing the distribution of data.

These functions are just a subset of the statistical capabilities offered by NumPy. By utilizing these functions, you can efficiently analyze and summarize your data, gain insights into its characteristics, and make informed decisions based on statistical information.

- `np.sum(np_array_2d, axis = 0)`

Many functions in NumPy require that you specify an **axis** along which to apply a certain calculation. np_array_2d is the array created that we want to get np.sum() of.

Typically the following rule of thumb applies:

- **axis = 0:** Apply the calculation “column-wise” or *horizontal* or y
- **axis = 1:** Apply the calculation “row-wise” or *vertical* or x

By mastering NumPy, you'll gain the ability to efficiently work with large datasets, perform intricate numerical computations, and conduct statistical analyses with ease. Its intuitive array-based approach simplifies programming and enables you to tackle a wide range of scientific and data-related tasks in Python.

Reference Links:

To access NumPy and its functions, import it in your Python code like this:

```
import numpy as np
```

We shorten the imported name to np for better readability of code using NumPy.

This is a widely adopted convention that makes your code more readable for everyone working on it.

- NumPy: the absolute basics for beginners
 - https://numpy.org/doc/stable/user/absolute_beginners.html
- NumPy fundamentals:
 - <https://numpy.org/doc/stable/user/basics.html>

-
- What is the difference between a Python list & a NumPy array?
 - https://numpy.org/doc/stable/user/absolute_beginners.html#whats-the-difference-between-a-python-list-and-a-numpy-array

What is an array?

An array is a central data structure of the NumPy library. An array is a grid of values and it contains information about the raw data, how to locate an element, and how to interpret an element. It has a grid of elements that can be indexed in various ways. NumPy arrays must contain data all of the same type and the elements that are the same type are referred to as the array dtype. That means that if your NumPy array contains integers, *all* of the values must be integers. If it contains floating point numbers, *all* of the values must be floats.

- A Quick Introduction: NumPy Array
 - <https://www.sharpsightlabs.com/blog/numpy-array-python/>
- numpy.array documentation:
 - <https://numpy.org/doc/stable/reference/generated/numpy.array.html>
- Introducing NumPy Arrays (Chapter 2):
 - https://pythonnumericalmethods.berkeley.edu/notebooks/chapter02.07-Introducing_numpy_arrays.html
- NumPy Array Cookbook: Generating & Multiplying Arrays
 - <https://towardsdatascience.com/numpy-array-cookbook-generating-and-manipulating-arrays-in-python-2195c3988b09>
- 6 Ways to Read CSV File in NumPy:
 - <https://www.pythonpool.com/numpy-read-csv/>
- numpy.loadtxt() documentation:
 - <https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html>
- How to read CSV files with NumPy?
 - <https://www.geeksforgeeks.org/how-to-read-csv-files-with-numpy/>
- Statistics – NumPy documentation
 - <https://numpy.org/doc/stable/reference/routines.statistics.html>
- Basic Statistics in NumPy using Jupyter Notebook:
 - <https://www.twilio.com/blog/basic-statistics-python-numpy-jupyter-notebook-html>

Session 2

In this session, we delved into several fundamental concepts related to **array manipulation** using NumPy. Here's a concise overview of what we explored:

Indexing and Slicing: We began by understanding the principles of indexing and slicing in arrays. Indexing refers to *accessing individual elements* of an array, while slicing involves *extracting a portion* of an array. This knowledge equips us with the ability to access and work with specific data points within arrays, enabling efficient data extraction and manipulation.

`ndarrays` can be **indexed** using the standard Python `x[obj]` syntax, where `x` is the array and `obj` is the selection. There are different kinds of indexing available depending on `obj`: basic indexing, advanced indexing and field access.

Basic **slicing** extends Python's basic concept of slicing to N dimensions. Basic slicing occurs when `obj` is a slice object (constructed by `start:stop:step` notation inside of brackets), an integer, or a tuple of slice objects and integers.

Copy vs View: We then explored the critical distinction between copying and viewing arrays. A copy creates an *entirely new* array with its own data, whereas a view provides a *different* perspective on the original data without duplicating it. We comprehended how changes made to views can affect the original array, and how modifications to copies do not impact the original array. Understanding this difference allows us to manipulate and analyze data while being mindful of memory usage and data integrity.

Array Modification: Finally, we dived into the realm of array modification. We learned how to modify arrays using various functions and techniques. This involved altering individual elements, changing data types, reshaping arrays, and even performing element-wise mathematical operations. Array modification functions provided us with the tools to transform and adapt arrays according to specific needs.

Reference Links:

- Indexing Routines -- NumPy Doc
 - <https://numpy.org/doc/stable/reference/arrays.indexing.html>
- Mastering NumPy Array Indexing: A Comprehensive Guide

-
- <https://saturncloud.io/blog/mastering-numpy-array-indexing-a-comprehensive-guide/>
 - How to Find Index of Value in NumPy Array (with examples)
 - <https://www.geeksforgeeks.org/how-to-find-the-index-of-value-in-numpy-array/>
 - Slice NumPy Arrays like a Ninja <- great guide!
 - <https://towardsdatascience.com/slicing-numpy-arrays-like-a-ninja-e4910670ceb0>
 - Python list comprehension:
 - https://www.w3schools.com/python/python_lists_comprehension.asp
 - How to index, slice, and reshape NumPy arrays:
 - <https://machinelearningmastery.com/index-slice-reshape-numpy-arrays-machine-learning-python/>
 - NumPy slicing ndarray:
 - <https://note.nkmk.me/en/python-numpy-ndarray-slice/> (great note)
 - NumPy array slicing:
 - <https://www.programiz.com/python-programming/numpy/array-slicing>
 - Indexing on ndarrays:
 - <https://numpy.org/doc/stable/user/basics.indexing.html>
 - How to index ndarrays:
 - <https://numpy.org/doc/stable/user/how-to-index.html>

"The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.

The copy *owns* the data and any changes made to the copy will not affect the original array, and any changes made to the original array will not affect the copy.

The view *does not own* the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view."

- NumPy: Copy vs View
 - https://www.w3schools.com/python/numpy/numpy_copy_vs_view.asp
- Copies and Views NumPy Doc

-
- <https://numpy.org/doc/stable/user/basics.copies.html>
 - `numpy.copy()`
 - <https://numpy.org/doc/stable/reference/generated/numpy.copy.html>
 - `numpy.ndarray.view()`
 - <https://numpy.org/doc/stable/reference/generated/numpy.ndarray.view.html#numpy.ndarray.view>
 - Intro to NumPy Array Copy vs View
 - <https://skillupwards.com/python/python-numpy-copy-vs-view>
 - Python Copy and Deep Copy
 - <https://note.nkmk.me/en/python-copy-deepcopy/>
 - Array Manipulation Methods
 - <https://numpy.org/doc/stable/reference/routines.array-manipulation.html>
 - `numpy.ravel()`
 - <https://numpy.org/doc/stable/reference/generated/numpy.ravel.html>
 - `numpy.ndarray.flatten()`
 - <https://numpy.org/doc/stable/reference/generated/numpy.ndarray.flatten.html>
 - Flatten a NumPy array with `ravel()` and `flatten()`
 - <https://note.nkmk.me/en/python-numpy-ravel-flatten/>
 - NumPy `ravel()`
 - <https://www.programiz.com/python-programming/numpy/methods/ravel>
 - NumPy `flatten()`
 - <https://www.programiz.com/python-programming/numpy/methods/flatten>
 - Python NumPy - `flatten()` vs `ravel()`
 - <https://thispointer.com/python-numpy-flatten-vs-ravel/>
 - Copies and views NumPy manual:
 - <https://numpy.org/doc/stable/user/basics.copies.html>
 - How to transpose a NumPy array 1D, 2D, and 3D:
 - <https://www.slingacademy.com/article/how-to-transpose-a-numpy-array/>

-
- `numpy.transpose()`
 - <https://numpy.org/doc/stable/reference/generated/numpy.transpose.html>
 - `numpy.ndarray.transpose()`
 - <https://numpy.org/doc/stable/reference/generated/numpy.ndarray.transpose.html>
 - Difference between `resize()` and `reshape()` in NumPy:
 - <https://www.geeksforgeeks.org/difference-between-reshape-and-resize-method-in-numpy/>
 - `numpy.reshape()`
 - <https://numpy.org/doc/stable/reference/generated/numpy.reshape.html>
 - Using NumPy `reshape()` to change shape of an array:
 - <https://realpython.com/numpy-reshape/>
 - `numpy.resize()`
 - <https://numpy.org/doc/stable/reference/generated/numpy.resize.html>
 - NumPy `resize()` function:
 - <https://www.w3resource.com/numpy/manipulation/resize.php>

9 Pandas



Session 1

This week was a deep dive into the powerful Pandas library for data manipulation and analysis. Here's a concise summary of what we covered:

Pandas Series:

- We began by understanding Pandas Series, which are one-dimensional labeled arrays.
- Explored various attributes of Series to access information and metadata about the data.
- Learned modification functions for Series, enabling us to manipulate and transform data.
- Examined the importance of the Index and Name in Series, which enhance data organization and readability.
- Experimented with slicing techniques for extracting specific elements from Series.
- Discovered the `update()`, `rename()`, and `replace()` functions, which provide essential tools for data manipulation and cleaning within Series.

During our exploration of Pandas Series, we gained a comprehensive understanding of this one-dimensional labeled array. We started by uncovering key attributes of Series, which enable us to access metadata and information about our data efficiently. For instance, the `shape` attribute reveals the dimensions of our Series, and the `dtype` attribute informs us about the data type. We learned how to perform data manipulations using functions like `head()` to retrieve the first few elements of the Series, and `tail()` to retrieve the last few

elements. Descriptive *statistics* became accessible with the `describe()` function, offering insights into the data's central tendencies. Transposing a Series using `transpose()` allows for a quick change in orientation. Moreover, we explored slicing techniques to extract specific elements, facilitating a more targeted data analysis. We also dove into the `update()`, `rename()`, and `replace()` functions, enhancing our data cleaning and transformation capabilities.

Example Code:

```
import pandas as pd

# Creating a Pandas Series
data = pd.Series([10, 20, 30, 40, 50],
name='Sample Data')

# Accessing attributes
shape = data.shape
dtype = data.dtype

# Data manipulation
head_elements = data.head(3)

tail_elements = data.tail(2)
description = data.describe()
transposed_data = data.transpose()

# Slicing
subset = data[1:4]

# Modification functions
data.update(pd.Series([100, 200, 300,
400, 500]))
data.rename("Updated Data")
data.replace(300, 350)
```

→ Pandas.Series documentation:

◆ <https://pandas.pydata.org/docs/reference/api/pandas.Series.html#pandas-series>

These examples demonstrate how we leveraged Pandas Series to explore, manipulate, and analyze data efficiently, enhancing our data analysis toolkit.

Pandas Dataframes:

-
- We then moved on to Pandas Dataframes, which are two-dimensional tabular data structures.
 - Explored attributes of Dataframes to gain insights into the data's structure and characteristics.
 - Discussed how Dataframes can handle a wide range of data types, making them versatile for various data analysis tasks.
 - Understood how Dataframes index their elements, making it easy to access and manipulate data.
 - Learned to convert lists into Dataframes and import data from external sources like CSV files.

Throughout our exploration of Pandas Dataframes, we delved into the intricacies of this **two-dimensional** tabular data structure. Starting with an understanding of Dataframe attributes, we accessed metadata that shed light on the data's structure, including the shape attribute revealing the dimensions and columns to list column names. We appreciated the Dataframe's versatility in handling **various data types** and how it simplifies data indexing with intuitive labels. The ability to convert lists into Dataframes and import data from external sources like CSV files opened up a world of possibilities. We honed our skills in using descriptive functions such as `head()` and `tail()` to quickly inspect data, while `describe()` offered valuable statistical summaries. We learned about the `transpose()` function for changing the orientation of Dataframes. Lastly, we differentiated between `sort_index` and `sort_values` to efficiently reorganize and sort Dataframes.

Example Code:

```
import pandas as pd

# Creating a Pandas Dataframe
data = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 22],
    'City': ['New York', 'San Francisco', 'Los Angeles']
})
```

```
# Accessing Dataframe attributes
shape = data.shape
columns = data.columns
data_types = data.dtypes

# Converting a list into a Dataframe
my_list = [1, 2, 3, 4, 5]
df_from_list = pd.DataFrame(my_list,
                             columns=['Numbers'])

# Sorting Dataframes
sorted_by_index =
data.sort_index(ascending=False)

sorted_by_values =
data.sort_values(by='Age')
```

```
# Importing data from an external CSV file
csv_data = pd.read_csv('data.csv')

# Descriptive functions
head_data = data.head(2)
tail_data = data.tail(1)
description = data.describe()
transposed_data = data.transpose()
```

→ Pandas.DataFrame documentation:

◆ <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html#pandas-dataframe>

These examples illustrate how we harnessed Pandas Dataframes to efficiently manage, explore, and analyze data in a structured tabular format, laying the foundation for advanced data manipulation and insights.

Reading CSV Files into Pandas Dataframes:

- We also covered the essential skill of reading CSV files into Pandas Dataframes, a fundamental task in data analysis.
- Explored descriptive functions for Dataframes, such as `head()`, `tail()`, `describe()`, and `transpose()`, which help us understand and summarize data quickly.

-
- Differentiated between `sort_index` and `sort_values` to comprehend how to reorder and sort Dataframes effectively.

In our exploration of reading CSV files into Pandas Dataframes, we acquired essential skills for importing and working with external data sources. We began by learning to import CSV files into Dataframes using the `pd.read_csv()` function, a fundamental operation for data analysis. This function automatically converts the CSV data into a structured tabular format, making it accessible for further analysis. We then dove into descriptive functions like `head()` and `tail()` to quickly examine the imported data, providing an initial overview of its content. The `describe()` function emerged as a powerful tool to generate statistical summaries, aiding in the initial understanding of the dataset's central tendencies. Furthermore, we explored modification functions for Dataframes, such as sorting using `sort_index` and `sort_values`, allowing us to organize the data effectively.

Example Code:

<pre>import pandas as pd # Importing CSV data into a Pandas Dataframe df = pd.read_csv('data.csv') # Inspecting the first few rows of the imported data first_few_rows = df.head() # Inspecting the last row of the imported data</pre>	<pre>last_row = df.tail(1) # Generating statistical summaries summary = df.describe() # Sorting the Dataframe by index and values sorted_by_index = df.sort_index(ascending=False) sorted_by_values = df.sort_values(by='Age')</pre>
--	--

➔ Pandas.Read_CSV documentation:

- ◆ https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html#pandas-read-csv

These examples demonstrate our proficiency in reading external data, specifically CSV files, into Pandas Dataframes. These skills form the foundation for effective data analysis,

manipulation, and visualization, ultimately leading to valuable insights and informed decision-making in data-related tasks.

Reference Links:

What is pandas?

pandas is a data manipulation package in Python for tabular data. That is, data in the form of rows and columns, also known as DataFrames. Intuitively, you can think of a DataFrame as an Excel sheet.

pandas' functionality includes data transformations, like [sorting rows](#) and taking subsets, to calculating summary statistics such as the mean, reshaping DataFrames, and joining DataFrames together. pandas works well with other popular Python data science packages, often called the PyData ecosystem, including

- [NumPy](#) for numerical computing
- [Matplotlib](#), [Seaborn](#), [Plotly](#), and other data visualization packages
- [scikit-learn](#) for machine learning

- A Practical Intro to Pandas Series:

- <https://towardsdatascience.com/a-practical-introduction-to-pandas-series-9915521cdc69>

- Python for Beginners - Pandas Series:

- <https://www.pythonforbeginners.com/basics/pandas-series-data-structure-in-python>

- Simple Pandas Intro Guide:

- <https://www.datacamp.com/tutorial/pandas>

- Ultimate Guide to Pandas Library:

- <https://www.freecodecamp.org/news/the-ultimate-guide-to-the-pandas-library-for-data-science-in-python/>

- 10 Mins to Pandas Documentation:

- https://pandas.pydata.org/docs/user_guide/10min.html

Core components of pandas: Series and DataFrames

The primary two components of pandas are the `Series` and `DataFrame`.

A `Series` is essentially a column, and a `DataFrame` is a multi-dimensional table made up of a collection of Series.

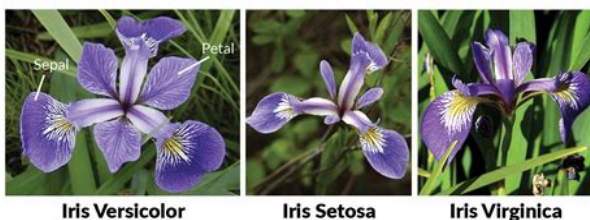
Series			Series			DataFrame	
	apples			oranges			
0	3		0	0	=	0	3
1	2	+	1	3		1	2
2	0		2	7		2	0
3	1		3	2		3	1

DataFrames and Series are quite similar in that many operations that you can do with one you can do with the other, such as filling in null values and calculating the mean.

- How to Create and Use a DataFrame:
 - <https://www.dataquest.io/blog/tutorial-how-to-create-and-use-a-pandas-dataframe/>
- Python Pandas Tutorial – Complete Intro for Beginners:
 - <https://www.learndatasci.com/tutorials/python-pandas-tutorial-complete-introduction-for-beginners/>
- Create a Pandas DataFrame in Python:
 - <https://www.pythonforbeginners.com/basics/create-pandas-dataframe-in-python>
- Python Pandas DataFrame:
 - <https://www.geeksforgeeks.org/python-pandas-dataframe/>
- Pandas DataFrames (very resourceful!)
 - <https://miamioh.edu/centers-institutes/center-for-analytics-data-science/students/coding-tutorials/python/pandas-dataframes.html>

Method	Usage
Values()	Return a list of all values in the dictionary
Update()	Updates the dictionary with the specified key-value pairs
setdefault()	Returns the value of the specified key. If the key does not exist insert the key, with the specified value
clear()	Removes all the elements from the dictionary
keys()	Returns a list containing the keys of the dictionary
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value

- How to Read CSV Files in Pandas:
 - <https://ecoagi.ai/articles/how-to-read-csv-files-pandas>
- Pandas Read CSV:
 - https://www.w3schools.com/python/pandas/pandas_csv.asp
- Pandas Read_CSV Tutorial:
 - <https://www.datacamp.com/tutorial/pandas-read-csv>
- Read CSV & Delimited Files in Pandas:
 - https://datagy.io/pandas-read_csv/



- Basics of Pandas Using Iris Dataset:
 - <https://www.geeksforgeeks.org/python-basics-of-pandas-using-iris-dataset/>
- Explanatory Analysis on Iris Dataset:
 - <https://www.geeksforgeeks.org/exploratory-data-analysis-on-iris-dataset/>

Session 2

Location Functions in Pandas: In this session, we delved into location functions in Pandas, particularly focusing on the `loc` and `iloc` functions. These functions are fundamental tools for selecting specific data within a Pandas DataFrame. `loc` is primarily used for label-based indexing, allowing us to access data by *specifying row* and *column labels*, while `iloc` is designed for **integer-based indexing**, enabling us to access data by providing **integer-based** row and column indices. `loc()` and `iloc()` serve as powerful methods for extracting and manipulating data within DataFrames, offering fine-grained control over row and column selection. We also learned about related functions like `at()` and `iat()`, which provide efficient **single-item** access. Understanding the nuances and differences among these selection functions equips us with the capability to precisely *locate* and *retrieve* data within Pandas DataFrames, a crucial skill for data analysis and manipulation.

Answers to Questions:

- *What does `loc[]` and `at[]` do?*

`loc[]`: `loc` is used for label-based indexing in Pandas DataFrames. It allows you to select specific rows and columns by specifying their labels or index names. For example, `df.loc[1:3, 'A':'C']` selects rows 1 to 3 and columns 'A' to 'C'.

at[]: at provides efficient single-item access in DataFrames by specifying both row and column labels. For example, `df.at[2, 'B']` retrieves the value at row 2 and column 'B'.

- *What does `iloc[]` and `iat[]` do?*

iloc[]: `iloc` is used for integer-based indexing in Pandas DataFrames. It allows you to select specific rows and columns by specifying their integer-based positions. For example, `df.iloc[1:3, 0:2]` selects rows 1 to 2 and columns 0 to 1.

iat[]: `iat` provides efficient single-item access in DataFrames by specifying both row and column integer positions. For example, `df.iat[2, 1]` retrieves the value at row 2 and column 1.

- *What is the difference between all selection functions?*

The main difference lies in the type of indexing they use:

`loc` and `at` use label-based indexing, allowing you to select data using row and column labels.

`iloc` and `iat` use integer-based indexing, enabling you to select data using integer positions.

Additionally, while `loc` and `iloc` allow you to select multiple rows and columns using slicing or lists of labels/positions, `at` and `iat` are specifically designed for single-item access, making them faster for accessing individual values.

In summary, `loc` and `iloc` are used for selecting multiple items using labels or integer positions, while `at` and `iat` are used for efficient single-item access. The choice between them depends on your specific data selection needs.

Example Code:

```
import pandas as pd

# Creating a sample DataFrame
data = {'A': [1, 2, 3, 4],
        'B': [10, 20, 30, 40],
        'C': ['apple', 'banana', 'cherry', 'date']}

df = pd.DataFrame(data)
```

```
# Using loc[] for label-based indexing
subset_loc = df.loc[1:2, ['A', 'C']] # Select rows 1 and 2, columns 'A' and 'C'

# Using iloc[] for integer-based indexing
subset_iloc = df.iloc[1:3, [0, 2]] # Select rows 1 and 2, columns 0 and 2

# Using at[] for single-item access by label
item_at = df.at[1, 'B'] # Accesses the item at row 1, column 'B'

# Using iat[] for single-item access by integer location
item_iat = df.iat[2, 0] # Accesses the item at row 2, column 0
```

These examples showcase the practical application of `.loc`, `.iloc`, `.at`, and `.iat` functions, enabling precise data extraction and manipulation within Pandas DataFrames, while understanding their distinctions ensures effective data handling. 👍

Dealing with Missing Data in Pandas: In this session, we tackled the concept of missing data in Pandas, a *common challenge* in data analysis. We began by understanding the significance of missing data and its impact on analysis, emphasizing the importance of recognizing and handling it **effectively**. We learned how to identify missing data in Pandas DataFrames, an essential step in data cleaning and preprocessing. To address missing data, we explored various functions. The `dropna()` function was introduced to remove rows or columns with missing values, providing flexibility in data cleaning. On the other hand, the `fillna()` function offered a way to fill missing values with specified replacements, enabling us to retain valuable data while mitigating gaps. The `isna()` function was utilized to generate Boolean masks, allowing us to identify the locations of missing data within DataFrames, a crucial step in data quality assessment. These functions collectively empower data analysts to handle missing data effectively, ensuring that data-driven decisions are based on complete and reliable information.

Answers to Questions:

- *What does the `dropna()` function do?*

The `dropna()` function in Pandas is used to remove rows or columns with missing values from a DataFrame. It provides flexibility by allowing you to specify the axis along which to drop missing values and the threshold for the number of non-missing values required to keep a row or column.

-
- *What does the `fillna()` function do?*

The `fillna()` function in Pandas is employed to fill missing values in a DataFrame with specified replacements. It enables data analysts to retain and complete data while mitigating gaps. You can specify the replacement value or use various methods like forward-fill or backward-fill to fill missing values.

- *What does the `isna()` function return?*

The `isna()` function in Pandas returns a DataFrame of the same shape as the input, where each element is a Boolean value. It indicates whether the corresponding element in the input DataFrame is missing (True) or not (False). This function is valuable for creating Boolean masks to identify the locations of missing data within a DataFrame, enabling precise handling of missing values.

Example Code:

```
import pandas as pd

# Read the 'mammograph.csv' file into a Pandas DataFrame
df = pd.read_csv('mammograph.csv')

# Checking for missing data using isna() function
missing_data_info = df.isna()

print("DataFrame with Missing Data Information:")
print(missing_data_info)

# Counting missing values in each column
missing_data_count = df.isna().sum()

print("\nCount of Missing Values in Each Column:")
print(missing_data_count)
```

In this example, we start by reading the 'mammograph.csv' file into a Pandas DataFrame using the `pd.read_csv()` function. Then, we use the `isna()` function to create a Boolean DataFrame where True indicates missing values, allowing us to visualize the locations of missing data in the dataset.

Additionally, we count the missing values in each column using `.sum()` on the Boolean DataFrame, providing us with the number of missing values for each column in the

'mammograph.csv' dataset. This information is crucial for data preprocessing and analysis when dealing with real-world datasets that often contain missing data.

Mini Project:

Here's a [template](#) for the mini project task:

Objective: The objective of this group mini project is to utilize the Pandas library for data analysis. You will work with the 'Team Summaries.csv' dataset, perform basic statistical analysis, and make arguments about the best NBA team of all time based on your findings.

Project Steps:

Dataset Import:

- Read in the 'Team Summaries.csv' dataset into a Pandas DataFrame using the `pd.read_csv()` function.
- Verify that the data loads correctly and inspect the dataset's structure.

Data Exploration:

- Begin by [exploring](#) the dataset to understand its contents:
 - Display the first few rows of the dataset using `head()`.
 - Use `info()` to check data types, null values, and non-null counts.
 - Calculate basic statistics with `describe()` to get an overview of key metrics.

Data Cleaning:

- Address [missing](#) data if any exist:
 - Identify columns with missing values using `isna()` and decide on the best approach (fill or drop).
- Check for [duplicate](#) entries and remove them if necessary using `drop_duplicates()`.

Statistical Analysis:

- Perform basic statistical analysis to identify trends and insights in the data. You might consider:
 - Season-wise performance metrics.
 - Wins, losses, and winning percentages.
 - Scoring statistics.
 - Defensive statistics.
 - Playoff success.

Determining the Best NBA Team:

- Based on your analysis, create arguments for what makes a team the "best."

Solution Example Code: *(One of MANY Ways)*

```
import pandas as pd

df = pd.read_csv('../Team Summaries.csv')

playoff = df.groupby('team') \
    .agg({'w': 'sum', 'l': 'sum', 'season': 'count'}) \
    .rename(columns={'w': 'total wins', 'l': 'total losses', 'season': 'seasons'})

playoff.dropna(inplace=True)

playoff['total number of games'] = playoff['total wins'] + playoff['total losses']
```

```
playoff['win percentage'] = playoff['total wins'] / playoff['total number of games']

print(playoff.sort_values(by='win percentage', ascending = False).head(5))

print(playoff.sort_values(by='total wins', ascending = False).head(5))

playoff['win percentage'].plot(kind='kde')
```

Understanding lambdas:

A lambda function in Python is a small, anonymous, and inline function. It's sometimes referred to as a "*lambda expression*." Lambda functions are typically used when you need a simple function for a short period and don't want to define a full-fledged named function using the `def` keyword. They are especially useful when you need to pass a function as an argument to another function, like in the case of `map()`, `filter()`, or `sort()`.

Here's the basic syntax of a lambda function:

lambda arguments: expression

- **lambda:** This keyword is used to define a lambda function.
- **arguments:** These are the input arguments or parameters that the lambda function takes.
- **expression:** This is a single expression that is evaluated and returned by the lambda function.

Lambda functions are limited in their capabilities compared to regular named functions defined with `def`. They are designed for simple operations and should not be used for complex tasks.

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

```
add = lambda x, y: x + y
result = add(5, 3) # result will be 8
```

Lambda functions are commonly used in scenarios where a short, throwaway function is needed. For instance, in list comprehensions or when sorting a list based on a specific criterion:

```
numbers = [3, 7, 1, 9, 4]
sorted_numbers = sorted(numbers, key=lambda x: x * -1) # Sort in reverse order
# sorted_numbers will be [9, 7, 4, 3, 1]
```

Lambda functions are concise and can make your code more readable when used appropriately, but it's essential to strike a balance and not overuse them, as complex logic is better suited for regular functions.

Reference Links:

- **(Very good)** Pandas location function guide:
<https://note.nkmk.me/en/python-pandas-at-iat-loc-iloc/>
- Pandas.DataFrame.loc() doc:
<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html>
- Pandas.DataFrame.iloc() doc:
<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iloc.html>
- Difference between loc() & iloc():
<https://www.geeksforgeeks.org/difference-between-loc-and-iloc-in-pandas-dataframe/>
- Pandas loc[] vs iloc[] guides:
 1. <https://www.nickmccullum.com/pandas-loc-iloc/>
 2. <https://stephenallwright.com/pandas-loc-vs-iloc/>
- Pandas.DataFrame.at() doc:
<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.at.html>
- Pandas.DataFrame.iat() doc:
<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iat.html>
- How to get and set Pandas cell values with at[] and iat[]
<https://practicaldatascience.co.uk/data-science/how-to-get-and-set-pandas-cell-values-with-at-and-iat>

-
- Efficient Ways to Print Full DataFrames:
https://www.golinuxcloud.com/pandas-print-entire-dataframe/#Efficient_ways_to_view_or_print_large_DataFrames_without_lag_or_system_crashes
 - Ways to Print DataFrames:
<https://www.geeksforgeeks.org/how-to-print-an-entire-pandas-dataframe-in-python/>
 - Working with missing data - Pandas doc
https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html
 - Missing Data in Pandas (great guide!)
<https://www.scaler.com/topics/pandas/missing-data-in-pandas/>
 - Handle Missing Values in DataFrames:
<https://stackabuse.com/python-how-to-handle-missing-dataframe-values-in-pandas/>
 - Missing Data - NaN Values Explained:
<https://www.bmc.com/blogs/pandas-nan-missing-data/>
 - How to Drop Rows with NaN Values?
<https://www.geeksforgeeks.org/how-to-drop-rows-with-nan-values-in-pandas-dataframe/>

10 Statistics & Visualizations

$$F = \frac{SSG}{\frac{SSB}{n_{groups}}} \quad cov(X, Y) = \frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})(y_i - \bar{y}) \quad df_{error} = (n_{blocks} - 1)(n_{groups} - 1)$$

$$\hat{x} = \frac{\sum X}{n} \quad S^2 = \frac{\sum (x - \bar{x})^2}{n - 1} \quad SE_{\hat{x}} = \frac{\sigma}{\sqrt{n}} \quad S^2 = \frac{\sum (x - \bar{x})^2}{n - 1}$$

$$P(x) = \frac{\lambda^x e^{-\lambda}}{x!} \quad X^2 = \sum \frac{(O - E)^2}{E} \quad f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

$$P = \frac{n!}{(n_1!)(n_2!) \dots (n_x!)} \times p_1^{n_1} \times p_2^{n_2} \dots p_x^{n_x} \quad \bar{x} = \frac{1}{n} \sum_{i=1}^n a_i$$

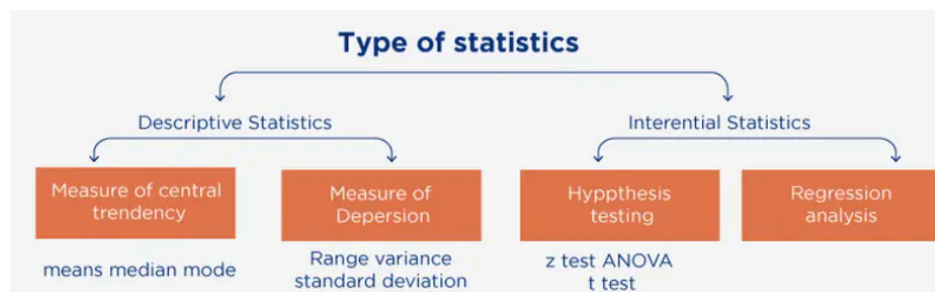
$$S = \sqrt{\frac{\sum (x - \bar{x})^2}{n - 1}} \quad P(x) = \frac{\lambda^x e^{-\lambda}}{x!} = \frac{8^3 e^{-8}}{3!} = 0.0287$$

$$P(x : n, p) = \binom{n}{x} (p)^x (1 - p)^{n-x} \quad \sigma^2 = \frac{\sum (x - \mu)^2}{N} \quad \sigma = \sqrt{\frac{\sum (x - \mu)^2}{N}}$$

$$\rho_{X,Y} = \frac{cov(X, Y)}{\sigma_X \sigma_Y} = \frac{\frac{1}{n} \sum (x - \bar{x})(y - \bar{y})}{\sqrt{\frac{\sum (x - \bar{x})^2}{n}} \sqrt{\frac{\sum (y - \bar{y})^2}{n}}} = \frac{\sum (x - \bar{x})(y - \bar{y})}{\sqrt{\sum (x - \bar{x})^2} \sqrt{\sum (y - \bar{y})^2}} \quad S^2 = \frac{\sum (x - \bar{x})^2}{n - 1} = \frac{SS}{df}$$

In this week's topic of statistics and visualization, we'll begin with the fundamental concept of **basic statistics**. Basic statistics is the foundation of data analysis, providing essential tools and techniques to understand *and* describe datasets. This includes measures of **central** tendency, which help us identify the typical or central value of a dataset, such as the mean, median, and mode. We'll explore how these measures can reveal key insights into data distribution and characteristics. Additionally, we'll delve into the concept of **feature scaling**, which is crucial for ensuring that data with varying scales and units can be effectively compared and analyzed. By the end of this topic, you'll have a solid understanding of these foundational statistical concepts, setting the stage for more *advanced* data analysis and visualization techniques.

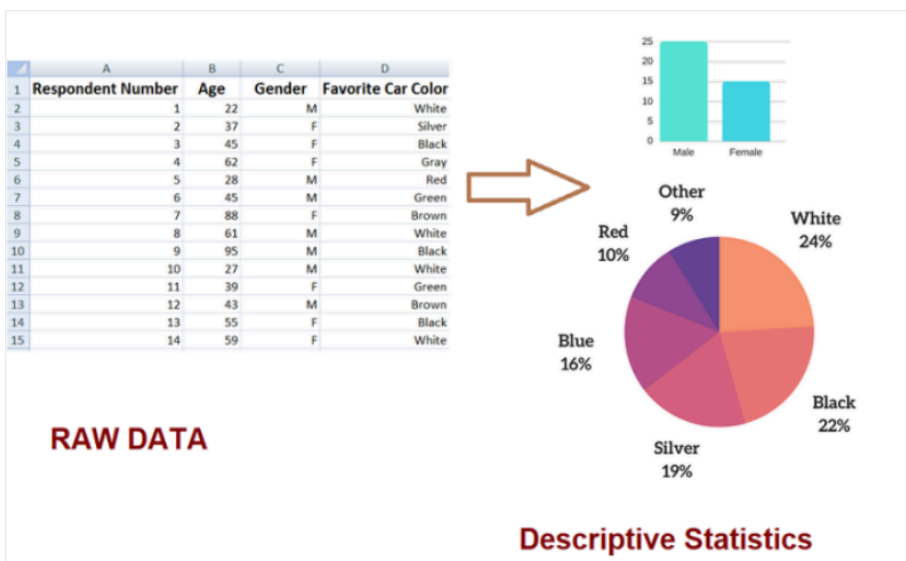
Basic Statistics



Basic statistics forms the cornerstone of data analysis, providing essential tools to make sense of datasets. We begin by comprehending the fundamental measures of central tendency: the mean,

median, and mode. The **mean** offers the average value, while the **median** identifies the central value, and the **mode** pinpoints the most frequent value within the data. These metrics illuminate the core characteristics of datasets and *guide* our analysis. Additionally, we explore **standard deviation**, a vital indicator of data's dispersion and variability, and the **Z-score**, a measure revealing how data points deviate from the **mean** in terms of **standard deviations**. We delve into **data normalization**, a technique that scales values into a shared range, and **standardization**, which aligns data to a standardized scale. These methods are indispensable for making data comparable and enhancing analysis. By mastering these foundational statistical concepts, beginners gain the tools needed to unearth insights and patterns within datasets confidently.

- **Descriptive Statistics:** Descriptive statistics are used to describe the essential features of the data in a study. They provide simple summaries about the sample and the measures. Together with simple graphics analysis, they form the basis of virtually every quantitative analysis of data. The below infographic provides a good summary of descriptive statistics:



Code Samples for Understanding:

Mean, Median, and Mode:

```
import statistics

data = [12, 15, 18, 22, 15, 17, 18, 15]

mean_value = statistics.mean(data)

median_value = statistics.median(data)
```

```
mode_value = statistics.mode(data)

print("Mean:", mean_value)

print("Median:", median_value)

print("Mode:", mode_value)
```

Standard Deviation:

```
import statistics

data = [12, 15, 18, 22, 15, 17, 18, 15]

std_deviation = statistics.stdev(data)

print("Standard Deviation:", std_deviation)
```

Z-Score:

```
import statistics

data = [12, 15, 18, 22, 15, 17, 18, 15]

x = 18

mean_value = statistics.mean(data)

std_deviation = statistics.stdev(data)

z_score = (x - mean_value) / std_deviation

print("Z-Score:", z_score)
```

Normalization:

```
data = [12, 15, 18, 22, 15, 17, 18, 15]

min_value = min(data)

max_value = max(data)

normalized_data = [(x - min_value) / (max_value - min_value) for x in data]

print("Normalized Data:", normalized_data)
```

Standardization:

```
data = [12, 15, 18, 22, 15, 17, 18, 15]

mean_value = statistics.mean(data)

std_deviation = statistics.stdev(data)

standardized_data = [(x - mean_value) / std_deviation for x in data]

print("Standardized Data:", standardized_data)
```

These code examples are designed to be beginner-friendly and demonstrate how to use these statistical functions in Python. You can replace the 'data' variable with your dataset for practical applications. 🚀

Questions in Jupyter Notebook:

The diagram shows the Z-score formula $Z = \frac{x - \mu}{\sigma}$. A green arrow points from the label 'Z-Score' to the variable 'Z'. A purple arrow points from the label 'Raw Score' to the variable 'x'. A dark blue arrow points from the label 'Mean' to the variable 'μ'. An orange arrow points from the label 'Standard Deviation' to the variable 'σ'.

How is the Z-score calculated?

- The Z-score, also known as the standard score, measures how many standard deviations a data point is away from the mean of a dataset. It is calculated using the formula:

$$Z = (X - \mu) / \sigma$$

Where:

Z is the Z-score.

X is the data point.

μ is the mean of the dataset.

σ is the standard deviation of the dataset.

The Z-score tells you how far a data point is from the mean in terms of standard deviations. A **positive** Z-score indicates that the data point is *above* the mean, while a **negative** Z-score indicates it is *below* the mean.

How is data normalized?

- Data normalization is the process of scaling data to fit within a specific range, often between 0 and 1. It is calculated using the formula:

$$X_{\text{normalized}} = X - X_{\text{min}} / X_{\text{max}} - X_{\text{min}}$$

Where:

X_{normalized} is the normalized value.

X is the original data point.

X_{\min} is the minimum value in the dataset.

X_{\max} is the maximum value in the dataset.

Normalization transforms data into a common scale, making it easier to compare and analyze data with different units or scales.

What is the use of standardization and normalization?

- **Standardization** (Z-score normalization): It transforms data into a standard scale with a mean of 0 and a standard deviation of 1. This is useful when you want to compare data points in terms of how many **standard deviations** they are from the mean. It's commonly used in machine learning algorithms like k-means clustering and principal component analysis (PCA).
- **Normalization** (Min-Max scaling): It scales data to a specific range, often between 0 and 1. This is beneficial when you want to ensure that data falls within a *specific range*, making it suitable for algorithms that rely on values in a specific interval. For example, neural networks often benefit from input data in the range [0, 1].

In our example code for this week, we applied both standardization and normalization to the 'speed' and 'velocity' datasets, showcasing how these techniques can transform data to facilitate analysis and comparison. *Standardization* makes data comparable in terms of standard deviations, while *normalization* scales data to a common range for easier interpretation and modeling.

Standardization and **normalization** are fundamental data preprocessing techniques that find extensive applications across various domains. **Here are real-life examples of their uses:**

Standardization:

- **Financial Analysis:** In the field of finance, standardization is often applied to financial metrics like *stock prices* or *returns*. It allows analysts to compare the volatility of different stocks by **scaling** them to a common standard deviation.
- **Machine Learning:** Many machine learning algorithms are sensitive to the scale of input features. *Standardization* ensures that **all** features have a mean of zero and a standard deviation of one, making them *compatible* with algorithms like k-means clustering or support vector machines.

-
- **Healthcare:** In medical research, *standardization* can be used to **normalize** measurements from different instruments or laboratories. For example, blood pressure readings taken by various devices can be standardized for analysis.
 - **Quality Control:** In manufacturing, *standardization* helps monitor and control product quality. It ensures that measurements taken from different machines or sensors are on a **common scale** for accurate quality assessment.

Normalization:

- **Image Processing:** In image analysis, pixel values in images are often *normalized* to a **range** of 0 to 1. This ensures that the brightness or color values are consistent, making it easier to apply image processing techniques.
- **Customer Ratings:** In recommendation systems, user ratings can be *normalized* to account for variations in individual rating behavior. This helps in providing accurate and fair recommendations to all users.
- **Environmental Monitoring:** In environmental science, data from various sensors and sources are often *normalized* to a **common scale** for analysis. For example, temperature data from different weather stations can be *normalized* for regional climate studies.
- **Social Sciences:** In surveys and social research, *normalizing* Likert scale responses (e.g., strongly agree to strongly disagree) allows researchers to compare responses across different survey questions and participants.
- **Text Mining:** In natural language processing, word frequencies or document lengths can be *normalized* to account for **variations** in text length, ensuring fair comparisons between documents or features.

Standardization and **normalization** are essential techniques that make data consistent, comparable, and suitable for analysis across diverse fields and applications. They help mitigate the impact of **varying scales**, units, or measurement methods, ensuring that data analysis and modeling produce reliable and meaningful results.

Reference Links:

- Mathematical Statistics - Python Library:
 - <https://docs.python.org/3/library/statistics.html>
- Descriptive Statistics:
 - <https://ethanweed.github.io/pythonbook/03.01-descriptives.html>
- Statistics for Data Science in Python (GREAT GUIDE)
 - <https://www.knowledgehut.com/blog/data-science/statistics-for-data-science-with-Python>
- Basic Statistics using Python:
 - <https://dev.to/amananandrai/basic-statistics-using-python-35l7>

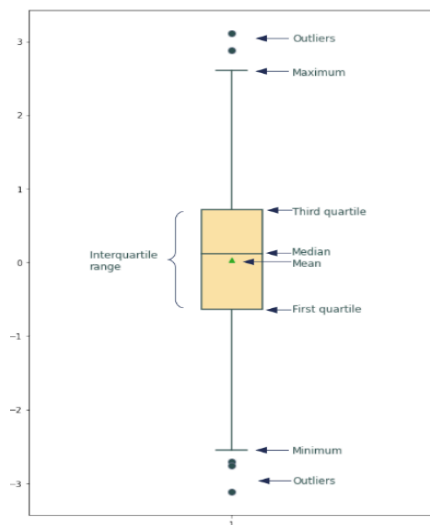
-
- Basic Statistics (delves deeper than this class, but great equation reference)
 - [https://eng.libretexts.org/Bookshelves/Industrial_and_Systems_Engineering/Chemical_Process_Dynamics_and_Controls_\(Woolf\)/13%3A_Statistics_and_Probability_Background/13.01%3A_Basic_statistics-_mean%2C_median%2C_average%2C_standard_deviation%2C_z-scores%2C_and_p-value](https://eng.libretexts.org/Bookshelves/Industrial_and_Systems_Engineering/Chemical_Process_Dynamics_and_Controls_(Woolf)/13%3A_Statistics_and_Probability_Background/13.01%3A_Basic_statistics-_mean%2C_median%2C_average%2C_standard_deviation%2C_z-scores%2C_and_p-value)
 - Python Statistics | mode() function:
 - <https://www.geeksforgeeks.org/python-statistics-mode-function/>
 - Python Statistics | median() function:
 - <https://www.geeksforgeeks.org/python-statistics-median/>
 - Python Statistics | mean() function:
 - <https://www.geeksforgeeks.org/python-statistics-mean-function/>
 - Calculating Standard Deviation | Comprehensive Guide:
 - <https://datagy.io/python-standard-deviation/>
 - Normalize Data in Python:
 - <https://www.statology.org/normalize-data-in-python/>
 - Practical Guide to Scaling & Normalization in Python:
 - <https://generativeai.pub/a-practical-guide-to-data-scaling-and-normalization-in-python-ffe7fa125313>
 - Simple Normalizing Guide:
 - <https://www.askpython.com/python/examples/normalize-data-in-python>
 - Normalization (Statistics) Wikipedia:
 - [https://en.wikipedia.org/wiki/Normalization_\(statistics\)](https://en.wikipedia.org/wiki/Normalization_(statistics))
 - How to Standardize Data in Pandas:
 - <https://www.geeksforgeeks.org/how-to-standardize-data-in-a-pandas-dataframe/>
 - Ways of Standardization (we're doing Min-Max):
 - <https://saturncloud.io/blog/how-to-standardize-columns-in-a-pandas-dataframe-using-python/>

Visualizations

Data visualization serves as a crucial tool in data analysis, offering a powerful means to convey insights, discover trends, and make data-driven decisions. In this segment, we delve into the significance of visualization and the techniques used to represent data visually, leveraging the capabilities of the `matplotlib`.`pyplot` library in Python.

-
- **Importance of Visualization:** Visualization aids in the exploration and interpretation of data. It transforms abstract numbers and statistics into intuitive graphical representations, allowing data analysts to uncover patterns, outliers, and correlations that might be less apparent in raw data.
 - **Visualization Techniques:** We explore various visualization techniques, such as box plots, scatter plots, histograms, bar plots, and pie charts, each serving specific purposes in data analysis.
 - **Relating Data to Visual Graphs:** The process of relating data to visual graphs involves selecting the most appropriate visualization method for a given dataset and research question. It entails understanding which type of graph best represents the data's characteristics and relationships.

Mastering data visualization techniques equips data analysts and scientists with the ability to translate complex data into clear and meaningful visuals. These visuals simplify data exploration, facilitate communication of insights, and enhance decision-making. Effective visualization is a cornerstone of data analysis, helping professionals extract valuable information and share it with clarity and impact.



A box plot, also known as a **box-and-whisker plot**, is a graphical representation of data distribution that provides a concise summary of key statistics. It is used for various purposes in data analysis:

1. *Understanding Data Distribution:*

- **Central Tendency:** A box plot reveals the **median**, which is the *middle value* of the dataset when arranged in ascending order. The median is represented by a line inside the box.

-
- **Variability:** It displays the **spread of the data** through the interquartile range (IQR), which is the range between the **first quartile** (25th percentile) and the **third quartile** (75th percentile). The IQR is enclosed within the box.
 - **Outliers:** Box plots indicate potential outliers, which are data points that *significantly differ* from the central tendency. Outliers are represented as individual points beyond the "whiskers" of the plot.

2. Comparing Distributions:

- Box plots are valuable for comparing the distributions of *multiple* datasets side by side. They provide a visual means to assess differences in central tendency, spread, and the presence of outliers.

3. Identifying Skewness:

- Skewness, which indicates the asymmetry of data, can be observed in box plots. A skewed distribution will have one whisker longer than the other.

4. Detecting Data Anomalies:

- Outliers appearing outside the whiskers can signify data anomalies or errors that require further investigation.

How to Interpret a Box Plot:

- The "**box**" represents the IQR and contains the middle 50% of the data.
- The "**whiskers**" extend from the box to the minimum and maximum non-outlier values within a defined range (often 1.5 times the IQR).
- The "**median**" line inside the box marks the middle value.
- **Outliers** are shown as individual points beyond the whiskers.

Box plots are particularly useful when dealing with datasets that may have varying distributions, outliers, or when comparing multiple datasets. They provide a visual summary of essential statistics and aid in making data-driven decisions, especially in fields like statistics, finance, and healthcare, where understanding data distribution is critical.

[matplotlib.pyplot.boxplot – Matplotlib](#)

Code 🐞 Example:

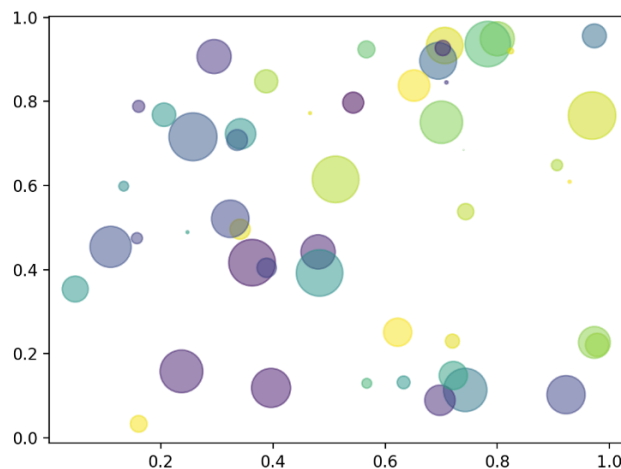
```
import matplotlib.pyplot as plt

# Data (replace with your dataset)
data = [10, 15, 20, 25, 30, 35, 40, 45, 50]

# Create a box plot
plt.boxplot(data)

# Add labels and title
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Box Plot Example')

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



A **scatter plot** is a graphical representation used to visualize the relationship or correlation between two *continuous* variables. This type of plot is particularly valuable for understanding how one variable **changes** concerning another and for identifying patterns, trends, or outliers in the data.

1. Key Characteristics and Uses:

- **Depicting Relationships:** Scatter plots display data points as individual dots on a two-dimensional plane. Each dot represents a data entry with values for **both** variables. By examining the distribution of these points, you can discern relationships between the variables.
- **Correlation Assessment:** Scatter plots are instrumental in assessing the **correlation** between variables. Correlation can be positive (both variables increase together), negative (one variable increases as the other decreases), or no correlation at all.
- **Identification of Outliers:** Outliers, data points that significantly deviate from the overall pattern, are readily apparent in scatter plots. These outliers can be important for understanding data anomalies or special cases.
- **Pattern Recognition:** Scatter plots can reveal *various* patterns, such as linear relationships, quadratic relationships, clusters, or no discernible pattern at all. This insight is essential for making data-driven decisions and forming hypotheses.

2. How to Interpret a Scatter Plot:

- Each dot on the plot represents a data point with values for two variables (usually represented on the x and y axes).
- The overall pattern of the dots provides insights into the relationship between the variables.
- If the dots form a recognizable pattern, it suggests a correlation or relationship.
- If the dots are scattered with no clear pattern, it suggests little to no correlation.

3. Common Use Cases:

- Scientific Research: Scientists often use scatter plots to analyze experimental data and explore relationships between variables.
- Economic Analysis: In finance and economics, scatter plots help analyze how various economic indicators correlate.

-
- Healthcare: Healthcare professionals may use scatter plots to examine the relationship between patient characteristics and health outcomes.
 - Quality Control: In manufacturing, scatter plots are used to detect patterns related to product quality.

Scatter plots are a fundamental tool in data analysis, aiding in the discovery of trends, relationships, and anomalies within datasets. They empower data analysts to draw meaningful conclusions and make informed decisions based on the visual representation of data.

[matplotlib.pyplot.scatter – Matplotlib](#)

Code 🐞 Example:

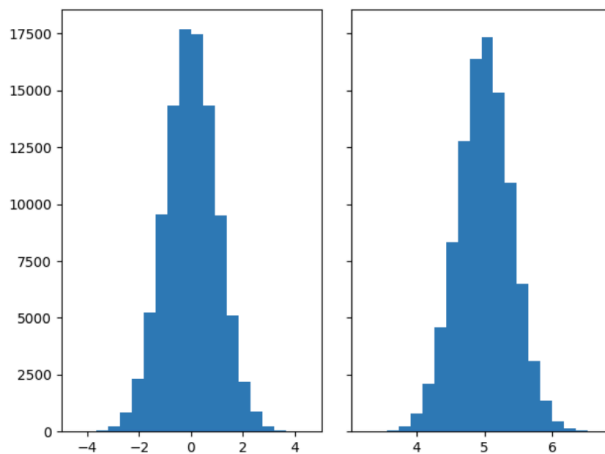
```
import matplotlib.pyplot as plt

# Data (replace with your values)
x_values = [1, 2, 3, 4, 5]
y_values = [10, 14, 8, 20, 12]

# Create a scatter plot
plt.scatter(x_values, y_values)

# Add labels and title
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Scatter Plot Example')

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



A **histogram** is a graphical representation used to visualize the distribution of a **continuous** or **discrete** dataset. It provides insights into how data is spread across different intervals or bins, making it an essential tool for understanding the frequency and patterns within a dataset.

1. Key Characteristics and Uses:

- **Data Distribution**: Histograms display the distribution of data by dividing the range of values into equal intervals or bins. Each bin represents a range of values, and the height of the bar over each bin represents the frequency or count of data points within that range.

-
- **Frequency Analysis:** By examining the heights of the bars, you can quickly identify which intervals contain more or fewer data points. This helps in understanding the central tendency and variability of the dataset.
 - **Skewness Detection:** Histograms can reveal whether a dataset is symmetric or skewed. A symmetric distribution will have a roughly equal distribution of data points on both sides of the center, while a skewed distribution will show a longer tail on one side.
 - **Outlier Identification:** Outliers, data points that significantly deviate from the main body of the data, are often visually apparent in histograms as bars that stand out from the rest.
 - **Data Binning:** Histograms allow you to choose the number of bins or intervals to represent data effectively. The choice of bin width can impact the interpretation of the data, and different binning strategies can be applied.

2. How to Interpret a Histogram:

- The **x-axis** represents the range of values in the dataset, divided into intervals (bins).
- The **y-axis** represents the frequency or count of data points within each bin.
- The **bars** in the histogram represent how many data points fall into each bin.
- The shape of the histogram provides insights into data distribution: a symmetric shape for normal distribution and skewness for non-normal distributions.

3. Common Use Cases:

- Statistical Analysis: Histograms are frequently used to assess the distribution of data in fields like statistics, economics, and social sciences.
- Quality Control: In manufacturing and engineering, histograms help monitor product quality and detect defects.
- Data Preprocessing: Data scientists use histograms to explore the distribution of features before selecting appropriate machine learning algorithms.
- Market Research: In marketing, histograms assist in analyzing customer demographics and preferences.

Histograms are a *powerful tool* for visualizing and understanding data distribution. They allow data analysts to uncover patterns, identify anomalies, and make informed decisions based on the distribution characteristics of a dataset.

[matplotlib.pyplot.hist – Matplotlib](#)

Code Example:

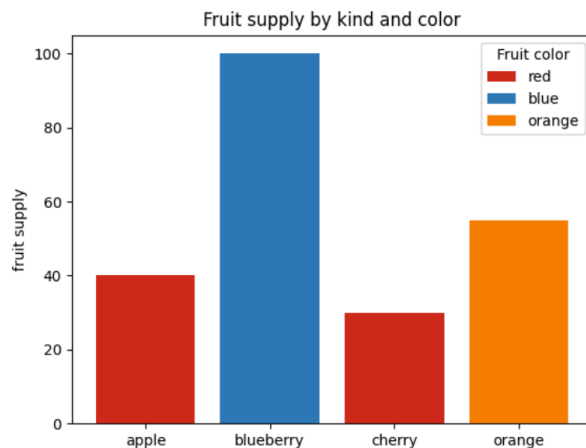
```
import matplotlib.pyplot as plt

# Data
data = [12, 15, 18, 22, 15, 17, 18, 15]

# Create a histogram
plt.hist(data, bins=5, edgecolor='black')

# Add labels and title
plt.xlabel('X-axis Label')
plt.ylabel('Frequency')
plt.title('Histogram Example')

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



A **bar plot**, also known as a **bar chart** or **bar graph**, is a graphical representation used to display categorical data. It is a versatile visualization tool that represents data as rectangular bars of varying lengths or heights, making it easy to compare values across different categories.

1. Key Characteristics and Uses:

- **Categorical Data Representation:** Bar plots are primarily used to represent and compare categorical data, such as different categories, groups, or labels.
- **Frequency or Count Display:** The length or height of each bar in a bar plot represents the frequency, count, or any numerical value associated with a specific category. It allows for a straightforward visual comparison of values.
- **Comparative Analysis:** Bar plots are effective for comparing data across categories. They make it easy to identify which categories have higher or lower values and visualize trends or disparities.
- **Bar Orientation:** Bar plots can be either vertical (columns) or horizontal (bars). The choice depends on the data and the preferred presentation style.

2. How to Interpret a Bar Plot:

- The **x-axis** typically represents the categorical data, such as categories, groups, or labels.
- The **y-axis** represents the numerical values associated with each category.
- Each **bar** represents a specific category and its corresponding numerical value.
- The **height** or **length** of each bar corresponds to the value it represents.

3. Common Use Cases:

- Market Research: Bar plots are frequently used to analyze and compare market share, customer preferences, or survey responses by category.
- Sales and Revenue Analysis: Businesses use bar plots to visualize sales figures, revenue breakdowns, and product performance.
- Education: In education, bar plots are used to compare student scores or performance by subject or grade.

-
- Social Sciences: Researchers use bar plots to display survey results, demographic data, or categorical variables.

Bar plots are an *essential* visualization tool for understanding and comparing **categorical data**. They provide a clear and intuitive representation of data distribution across categories, making it easier to identify trends, disparities, and patterns in the data.

[matplotlib.pyplot.bar – Matplotlib](#)

Code 🐞 Example:

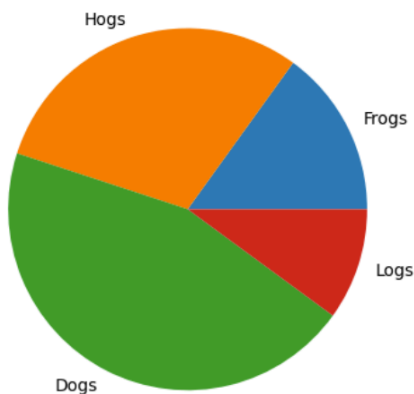
```
import matplotlib.pyplot as plt

# Data
categories = ['Category A', 'Category B', 'Category C', 'Category D']
values = [20, 35, 12, 27]

# Create a bar plot
plt.bar(categories, values)

# Add labels and title
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Plot Example')

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



A **pie chart** is a circular graphical representation used to illustrate the **proportion or distribution** of categorical data. It divides the *entire circle* into slices, with each slice representing a specific category or group. The **size** of each slice corresponds to the **proportion** of data it represents, making pie charts ideal for displaying parts of a whole.

1. Key Characteristics and Uses:

Proportion Display: Pie charts are used to show how individual parts contribute to a **whole**. Each slice (or sector) represents a category's share of the total data.

Percentage Representation: Typically, pie chart slices are labeled with *percentages*, indicating the proportion of the **whole** that each category represents.

Visual Comparison: Pie charts enable viewers to visually **compare** the relative sizes of different categories or groups. It's easy to identify which categories are more significant or smaller in relation to the whole.

2. How to Interpret a Pie Chart:

- The **entire circle** represents the whole dataset or 100%.
- Each **slice** represents a specific category or group.
- The size of each slice (its angle) is **proportional** to the proportion of data it represents.
- Slices are often labeled with percentages or category names to indicate their share of the whole.

3. Common Use Cases:

- Market Share Analysis: Businesses use pie charts to display the market share of products or services within an industry.
- Budget Allocation: Organizations use pie charts to illustrate how budget resources are distributed across different departments or expenses.
- Survey Results: Pie charts are commonly used to represent survey responses, showing the distribution of answers among multiple choices.
- Composition Analysis: Pie charts help analyze the composition of a whole, such as the allocation of time, resources, or expenses.

While pie charts are effective for displaying **proportions** and showing how parts relate to a whole, they are most suitable when dealing with a *limited number* of categories (typically less than five to seven). For larger datasets or when precise comparisons are necessary, other chart types like bar plots or stacked bar charts may be more appropriate.

[matplotlib.pyplot.pie – Matplotlib](#)

Code 🐘 Example:

```
import matplotlib.pyplot as plt

# Data
categories = ['Category A', 'Category B', 'Category C', 'Category D']
sizes = [15, 30, 45, 10]

# Create a pie chart
plt.pie(sizes, labels=categories, autopct='%1.1f%%', startangle=90)

# Add title
plt.title('Pie Chart Example')

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

Reference Links:

- Matplotlib Boxplot Guide:
 - <https://hyperskill.org/learn/step/16818>
- Matplotlib Pie Chart Info:
 - https://matplotlib.org/stable/gallery/pie_and_polar_charts/pie_features.html
- Matplotlib Scatter Plot Example Guide:
 - <https://stackabuse.com/matplotlib-scatterplot-tutorial-and-examples/>
- Plot a Pie Chart Using Matplotlib:
 - <https://www.geeksforgeeks.org/plot-a-pie-chart-in-python-using-matplotlib/>
- Matplotlib Pyplot Tutorial:
 - <https://matplotlib.org/stable/tutorials/introductory/pyplot.html>
- **Matplotlib Cheat Sheet Downloads:** 📄
 - <https://matplotlib.org/cheatsheets/>
- Matplotlib Data Visualization Code Examples:
 - <https://discovery.cs.illinois.edu/guides/Data-Visualization/simple-matplotlib/>
- Introduction to Plotting w/ Matplotlib in Python:
 - <https://www.datacamp.com/tutorial/matplotlib-tutorial-python>

I hope this helps! 😊

Tutorials:

Plotly Tutorial - Learn Plotly | Absolute Beginners:

<https://www.tutorialspoint.com/plotly/index.htm>

Matplotlib Tutorial - Learn Matplotlib (Simple Easy Learning) | Absolute Beginners:

<https://www.tutorialspoint.com/matplotlib/index.htm>

SQLite Tutorial - SQLite Getting Started | SQLite Sample Database:

<https://www.sqlitetutorial.net/sqlite-sample-database/>

SQLAlchemy - Database Toolkit for Python | SQL Toolkit & Object Relational Mapper:

<https://www.sqlalchemy.org/>

