

# Introduction to “Programming for Data Science”

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# — Short Biography

## **Phuong Nguyen**

University of L'Aquila, Italy

Email: [phuong.nguyen@univaq.it](mailto:phuong.nguyen@univaq.it)

Website: <https://www.disim.univaq.it/ThanhPhuong>

- PhD in Computer Science: University of Jena (Germany)
- Lecturer: FPT University, Duy Tan University (Vietnam)
- Postdoctoral researcher: Polytechnic University of Bari, University of L'Aquila (Italy)
- From January 2022: Assistant Professor, University of L'Aquila
- June 2023: ASN Qualification as Associate Professor (Issued by the Italian Ministry of University and Research)
- Research Interests: Machine Learning, Recommender Systems, Software Engineering

# — Agenda

- Introductory concepts of computer science.
- Introduction to architectures (hardware, software), operating systems, mathematical logics, algorithm and complexity.
- Principles of software engineering with a specific focus on requirement engineering, testing and debugging.
- Syntax and semantics: definitions and examples in Python.
- Basic python elements: expressions, variables, assignments, numeric types and strings. Control commands.

# — Agenda

- Python Structured types: lists. Iteration and Recursion.  
Functions, scope of variables and abstraction.
- Python Structured types: sequences, tuples,
- Python Structured types: dictionaries, sets. Exceptions and assertions. File management.
- Object-oriented Paradigm: Python classes and objects.
- Data reading, cleaning, plotting. Specific libraries (numpy, pandas, matplotlib, etc).

# — Teaching methods

- Lectures, seminars, and code demonstrations.
- The teaching focuses on learning-by-doing, and taking students through realistic projects.
- Extensive use of problem-based learning.
- Students are exposed to sufficient levels of detail and risk, thus gaining an appreciation for the challenges of solving actual data science problems.

# — Student involvement

- Students are encouraged to participate in the lecture by having questions/comments.
- Students can investigate a specific topic, and present it in the class.
- It is highly advised that students use GitHub to store their projects and to collaborate with other team members.

# Assignment and Exam

- Homework, project and oral exam.
- There will be a list of projects so that you can choose.
- Projects are related to the applications of Data Science.
- Reports should be written in English.
- Each group with 2 or 3 students works on a common project.
- Every member of the group should contribute to the project. I am going to check this during the exam (Please be prepared :-)).

# — Grading

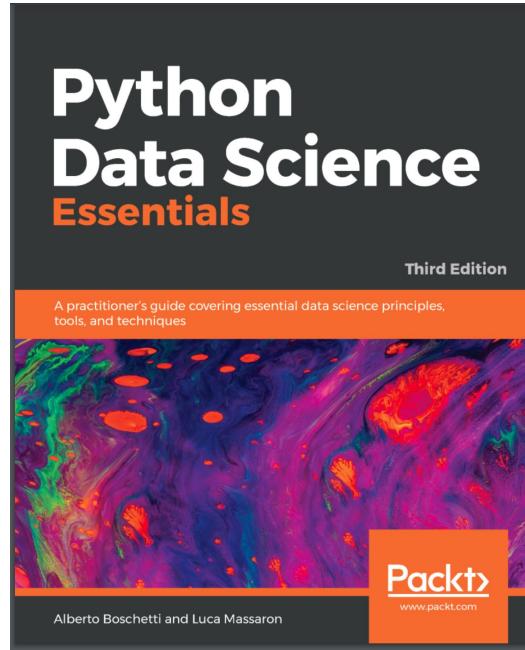
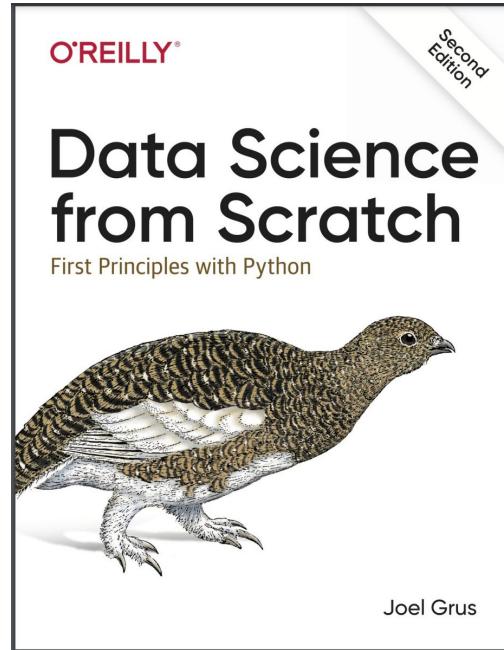
Task	Mark (%)
Selecting and adopting suitable techniques	20
<b>Designing and implementing the application components</b>	<b>40</b>
Performing evaluation of the application	20
Understanding the knowledge of the methodologies and technologies presented during the course	10
Participation (interaction) in the lectures	10

# — Libraries and platforms



- There are various libraries in Python, ready to be used for different tasks
- Students are encouraged to use these libraries/platforms for their projects

# References



- Joel Grus, “**Data Science from Scratch**” ([Link](#))
- Alberto Boschetti and Luca Massaron, “**Python Data Science Essentials**” ([Link](#))
- Datasets for working with data ([Link](#))
- Kaggle: A platform for with several Machine Learning problems

# Introduction to Data Science

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

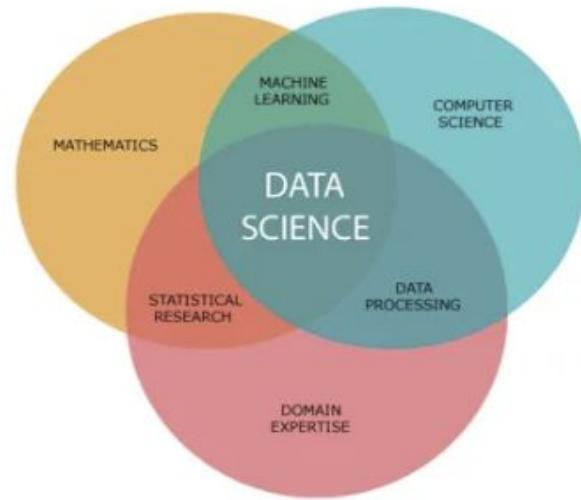
Università degli Studi dell'Aquila

## Chapter Goals

In this chapter you will learn:

- Introduction to Data Science
- Computer hardware, software and programming
- How to write and execute your first Python program
- How to diagnose and fix programming errors
- How to use pseudocode to describe an algorithm

# Data Science



- Data Science is a multidisciplinary field that involves extracting meaningful insights and knowledge from structured and unstructured data using a combination of techniques from statistics, mathematics, computer science, and domain-specific knowledge.
- It encompasses the process of collecting, processing, analyzing, and interpreting large volumes of data to solve complex problems, identify patterns, and make informed decisions.

## The need for Data Science

- Data Science enables organizations and individuals to make informed decisions, optimize processes, and uncover valuable insights from vast amounts of data.
- In today's digital age, where data is generated at an unprecedented scale from various sources (e.g., social media, sensors, business transactions), understanding and utilizing this data effectively is critical for competitive advantage, innovation, and efficiency.



Image source: <https://www.linkedin.com/pulse/title-demystifying-big-data-unveiling-power-analytics-shruti-kashyap->

# Data Scientists/Engineers

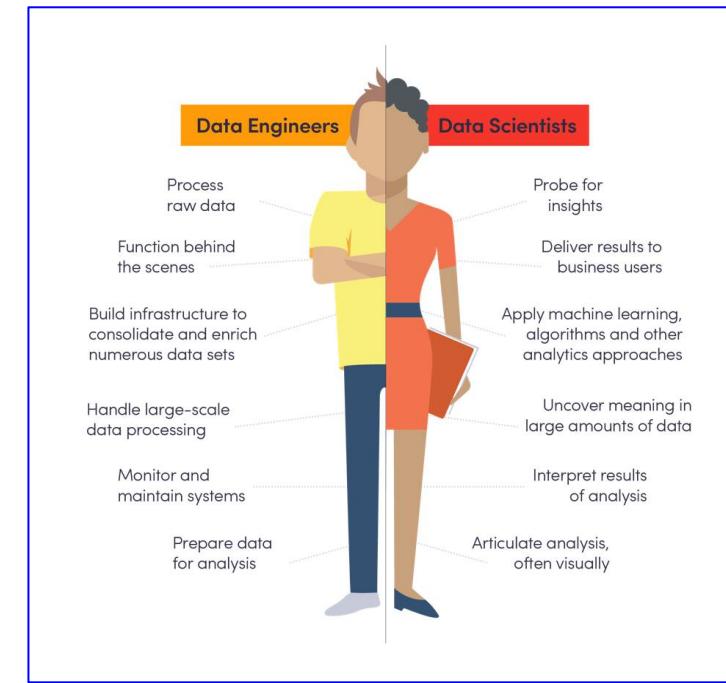


Image source: <https://www.astronomer.io/blog/why-every-data-scientist-needs-a-data-engineer/>

- Data Scientists: Focus on extracting insights from data, work with statistical models, machine learning algorithms, and data analysis techniques to find patterns, make predictions, and provide actionable business insights.
- A Data Scientist's primary job is to analyze data, interpret trends, and create data-driven solutions.

# Data Scientists/Engineers

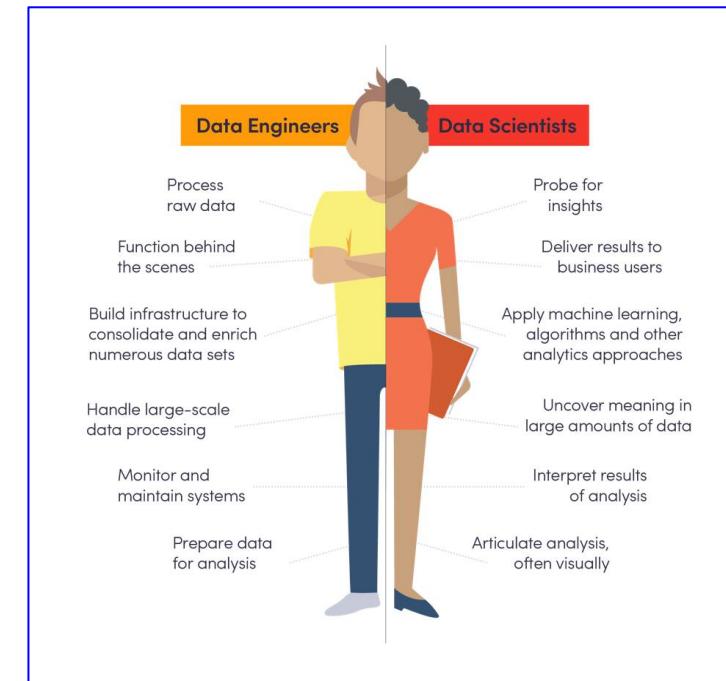


Image source: <https://www.astronomer.io/blog/why-every-data-scientist-needs-a-data-engineer/>

- Data Engineers: Focus on building the infrastructure for data generation, storage, and access, ensure that data pipelines are efficient, reliable, and scalable so that data can be easily accessed and processed by data scientists and analysts.
- A Data Engineer's role is to manage and optimize data flow and pipelines, ensuring that data is clean, well-structured, and readily available.

## Five ‘V’s in Big Data

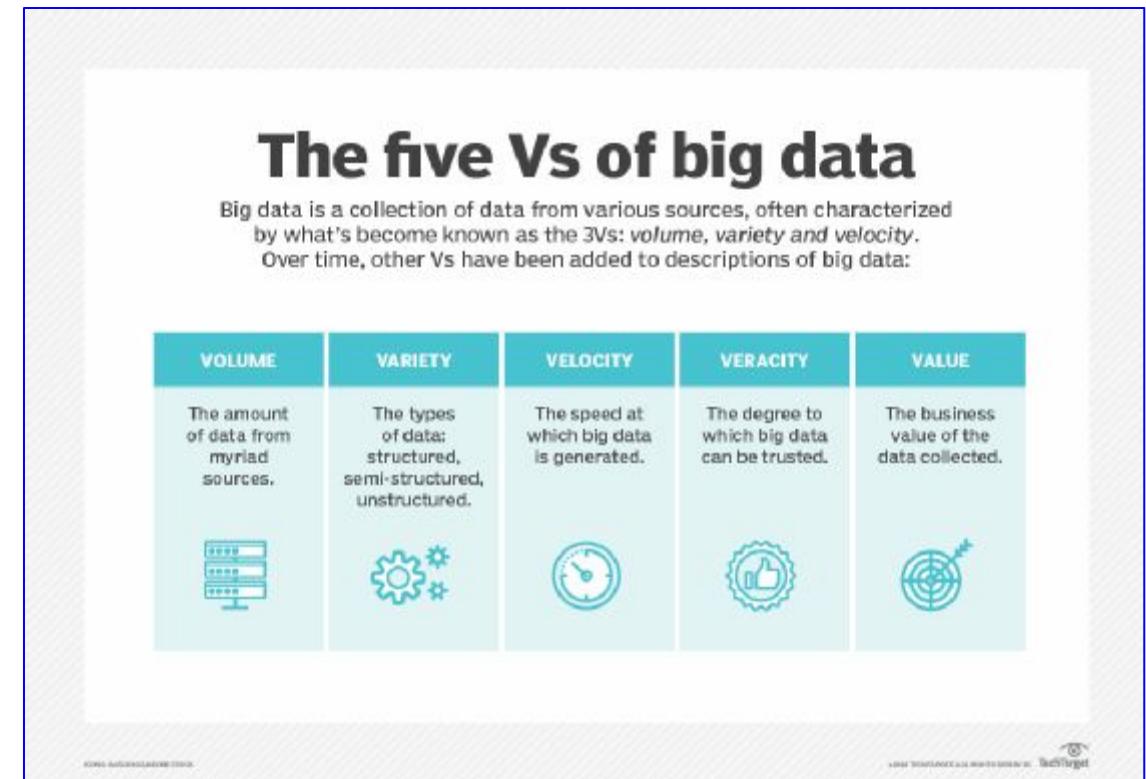


Image source:  
<https://www.marketingteacher.com/5vs-of-big-data-in-marketing/>

- **Volume** refers to the sheer size of the data being generated and stored. In the context of Big Data, we're dealing with massive amounts of data, often measured in terabytes (TB), petabytes (PB), or even zettabytes (ZB).

# Five ‘V’s in Big Data

# The five Vs of big data

- **Volume** refers to the sheer size of the data being generated and stored. In the context of Big Data, we're dealing with massive amounts of data, often measured in terabytes (TB), petabytes (PB), or even zettabytes (ZB).
  - Social media platforms like Facebook or Twitter generate petabytes of data daily from user posts, likes, comments, and shares. Sensors on industrial machines, IoT devices, or smartphones are also constantly generating massive data volumes.

# Five ‘V’s in Big Data

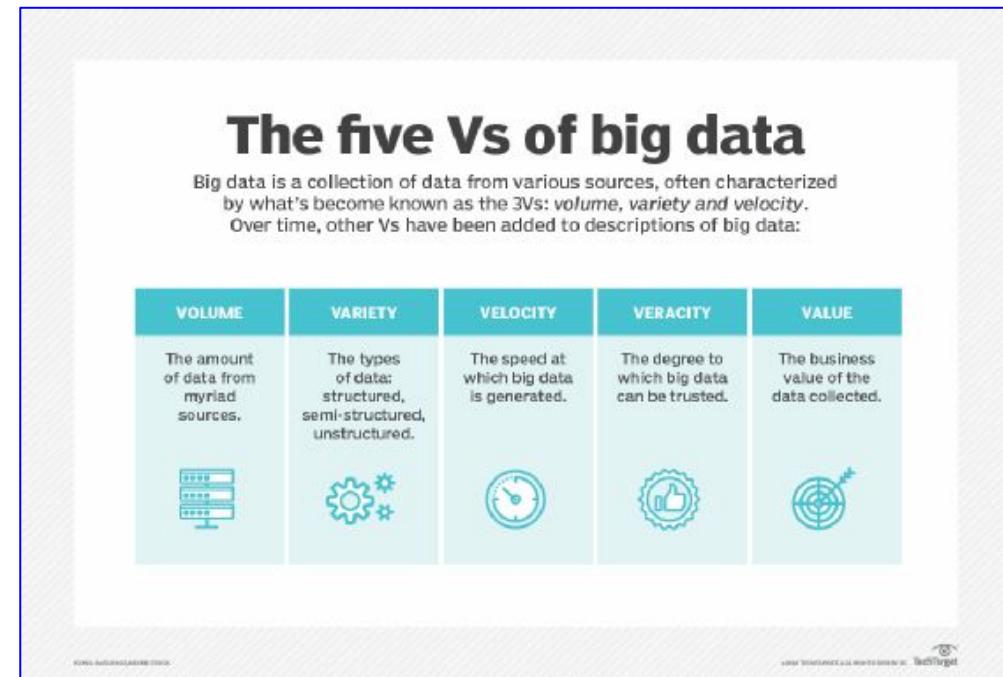
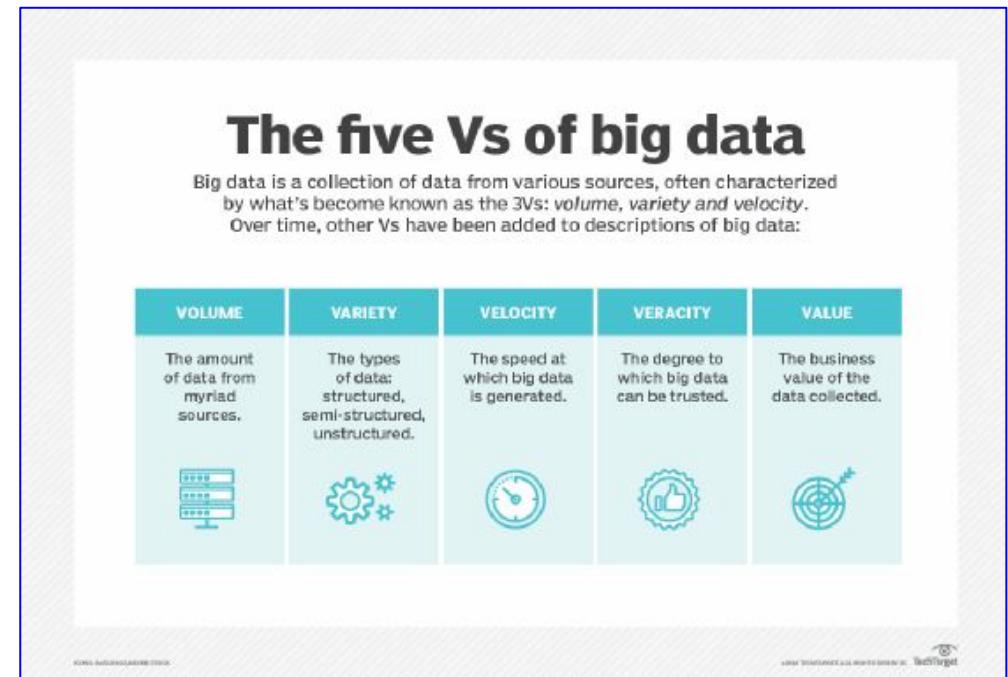


Image source:  
<https://www.marketingteacher.com/5vs-of-big-data-in-marketing/>

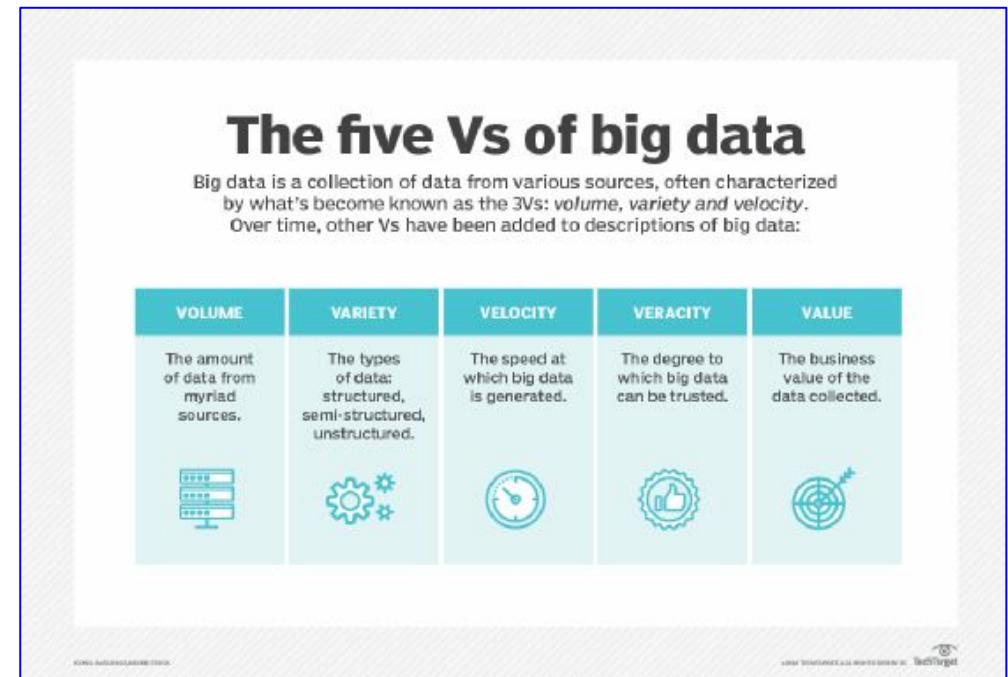
- **Variety** refers to the different types of data formats and sources. Big Data is not just structured (like in traditional databases), but also unstructured (like text, images, audio) and semi-structured (like JSON or XML files). This diversity adds complexity to how data is managed and processed.
- Example: A company may collect structured data from sales transactions (rows and columns in a database), semi-structured data from logs and emails, and unstructured data from customer reviews, social media posts, or images.

# Five ‘V’s in Big Data



- **Velocity** refers to the speed at which data is generated, processed, and analyzed. This could be real-time or near real-time data that needs to be handled continuously as it arrives.
- Example: Streaming data from stock markets, social media feeds, or IoT sensors generates data at an incredibly high speed that needs to be processed instantly to be valuable (e.g., real-time fraud detection in financial transactions).

# Five ‘V’s in Big Data



- **Veracity** refers to the trustworthiness, accuracy, and quality of the data. With Big Data, there is often uncertainty about the data's origin, accuracy, or completeness, making it challenging to derive meaningful insights from it.
- Example: Social media data, for instance, may contain a lot of noise (irrelevant, misleading, or inaccurate data). Similarly, sensors might generate faulty data due to malfunction or calibration issues.

# Five ‘V’s in Big Data

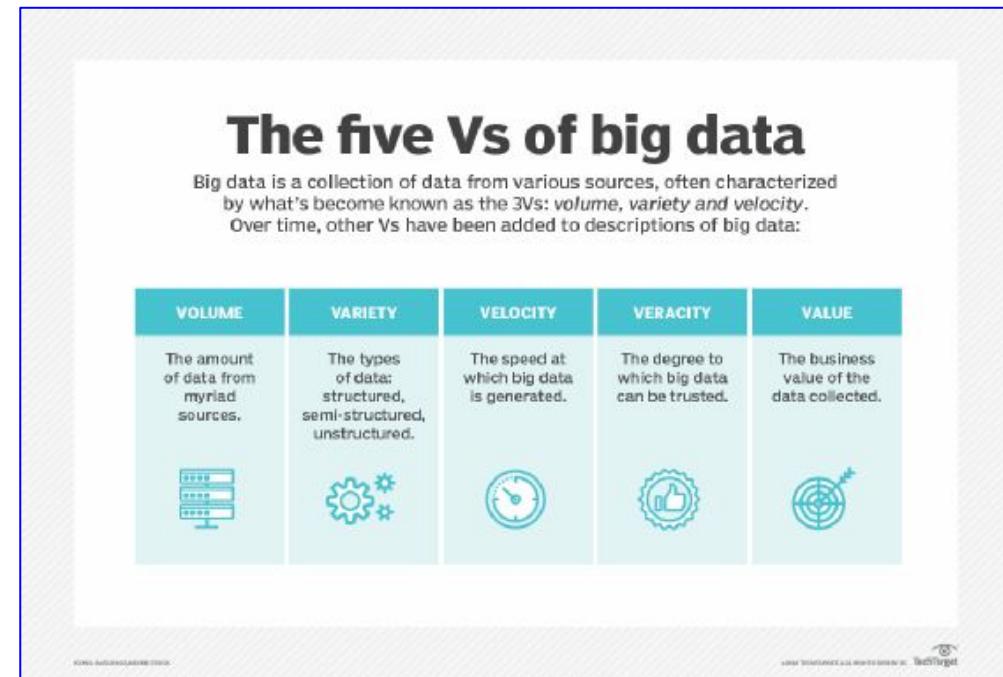


Image source:  
<https://www.marketingteacher.com/5vs-of-big-data-in-marketing/>

- **Value** refers to the worth or usefulness of the data once it's processed and analyzed. The ultimate goal of Big Data is to extract meaningful insights and drive value for businesses, governments, or individuals.
- Example: Retailers using data analytics to predict customer behavior, optimize inventory, and personalize marketing strategies. Financial institutions use Big Data analytics for fraud detection, and healthcare providers use it to improve patient care through predictive models.

## Key Components

- Data Collection: The first step in data science involves gathering data from various sources, such as databases, web scraping, sensors, or public datasets. The data can be structured (e.g., tables in a database) or unstructured (e.g., text, images, videos).
- Data Cleaning: Raw data is often messy and may contain missing values, duplicates, or errors. Data cleaning involves preprocessing and transforming the data to ensure it is accurate, consistent, and usable for analysis. This step may include tasks like filling missing values, removing outliers, and normalizing data.

## Key Components (2)

- Exploratory Data Analysis (EDA): EDA is the process of analyzing data sets to summarize their main characteristics, often using visual methods like graphs and charts. It helps data scientists understand the data's distribution, identify trends, and detect anomalies or patterns.
- Feature Engineering: It involves creating new features or modifying existing ones from raw data to improve the performance of machine learning models. It plays a critical role in boosting model accuracy.

## Key Components (3)

- Statistical Analysis: Data science relies heavily on statistics to make inferences and predictions from data. Techniques like hypothesis testing, regression analysis, and probability distributions are used to extract insights and validate assumptions.
- Machine Learning: This is a core aspect of data science that involves creating algorithms that can learn from data and make predictions or classifications.
- Machine learning models are trained on historical data and used for tasks such as forecasting, pattern recognition, and recommender systems.

## Key Components (3)

- Data Visualization: The graphical representation of data to communicate findings in a clear and intuitive way. Tools like Matplotlib, Seaborn, Tableau, and Power BI help data scientists create charts, graphs, and dashboards to illustrate trends and patterns.



# Software, Program, Algorithms

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# Executing a Program

## Computer Programs

- A computer program tells a computer the sequence of steps needed to complete a specific task
  - The program consists of a very large number of primitive (simple) instructions
- Computers can carry out a wide range of tasks because they can execute different programs
  - Each program is designed to direct the computer to work on a specific task

### ***Programming:***

The act of designing, implementing, and testing computer programs

## Introduction

- Running a program in a computer involves writing the program, translating it into machine-readable code, loading it into memory, and executing it through the CPU.
- The operating system plays a crucial role in managing resources, handling I/O, and multitasking to ensure smooth execution.
- Programs continue running until they complete their tasks or encounter an error, after which the OS cleans up the allocated resources.

## Process Overview

- Write Code: A developer writes the program in a high-level programming language.
- Compilation/Interpretation: The program is compiled into machine code or interpreted line-by-line.
- Loading: The OS loads the program and its dependencies into RAM.
- Execution: The CPU fetches, decodes, and executes the program's instructions.

## Compiling a Program

- Input/output operations and system calls are handled by the OS.
- Multitasking: If multiple programs are running, the OS switches between them efficiently.
- Memory Management: The OS manages memory allocation and virtual memory if needed.
- Interrupt Handling: The CPU handles external events or interrupts while running the program.
- Termination: The program terminates after completion, and resources are released.

## Compiling a Program

- Compilation or interpretation is the process of translating the human-readable code into machine-readable instructions (binary form).
- Compiled languages (e.g., C, C++) convert the entire code into an executable file (e.g., .exe in Windows or binary in Linux). The compiler generates machine code specific to the computer's architecture.
- Interpreted languages (e.g., Python, JavaScript) translate the code line-by-line at runtime via an interpreter.

## Loading into the Memory

- Once the program is compiled or interpreted, it needs to be loaded into RAM (Random Access Memory) for the CPU to execute it. This task is performed by the operating system (OS).
- The OS creates a process for the program, and the loader (a part of the OS) loads the program's binary into memory. It also loads necessary libraries or dependencies that the program might require.

## Executing a Program

- Program instructions and data (such as text, numbers, audio, or video) are stored in digital format
- When a program is started, it is brought into memory, where the CPU can read it.
- The CPU runs the program one instruction at a time.
  - The program may react to user input.
- The instructions and user input guide the program execution
  - The CPU reads data (including user input), modifies it, and writes it back to memory, the screen, or secondary storage.

# Software

## Software

- Software refers to the set of instructions, programs, or data that tell a computer how to perform specific tasks.
- Unlike hardware, which is the physical component of a computer, software is intangible and operates through hardware to carry out tasks. It encompasses all the non-physical components that interact with the hardware to process data and produce results.
- Software plays a crucial role in computing, from managing system resources to enabling user interaction through applications.

## Program and Software

- A program is a specific set of instructions written in a programming language that a computer can execute to perform a particular task. It is a single piece of executable code that serves a specific purpose.
- Software is a broader term that includes not only programs but also all related components, such as libraries, data files, and documentation, necessary for the proper functioning of a system.
- Software can consist of multiple programs and may involve system utilities, drivers, and user interfaces.

## System Software

- System software provides the foundation for running other software and managing hardware components. It is essential for the operation of a computer and is typically the first type of software installed when setting up a system.
- Operating System (OS): The OS manages the computer's hardware and software resources, acting as an intermediary between users and hardware. It handles tasks such as file management, memory allocation, process scheduling, and device control.
- Examples: Windows, macOS, Linux, Android, iOS.

## System Software

- **Device drivers:** They are specialized software that allow the operating system to communicate with hardware components like printers, graphics cards, network cards, etc. Without the correct drivers, hardware components cannot function properly.
- Example: The driver for a graphics card enables the OS to interact with the GPU and display graphics.
- **Firmware** is a type of low-level software embedded in hardware devices. It provides control over hardware functions and is typically stored in non-volatile memory like ROM or flash memory. Firmware updates can improve hardware performance or fix bugs.

## Application Software

- **Application software** is designed to help users perform specific tasks. These tasks can range from productivity, communication, and entertainment to more specialized areas such as design, research, and business management.
- **Productivity Software:** This category includes tools that allow users to create documents, presentations, spreadsheets, and databases. Examples: Microsoft Office (Word, Excel, PowerPoint), Google Docs, Apache OpenOffice.
- **Web Browsers:** They allow users to access and navigate the internet. They interpret HTML, CSS, and JavaScript to display web pages and handle online interactions. Examples: Google Chrome, Mozilla Firefox, Safari, Microsoft Edge.

## Application Software

- **Communication Software:** Software that facilitates communication through email, video calls, or instant messaging. Examples: Microsoft Outlook, Slack, Zoom, WhatsApp.
- **Multimedia Software:** These programs enable users to create, view, and edit multimedia content, including images, videos, and audio files. Examples: Adobe Photoshop, Final Cut Pro, VLC Media Player, Spotify.
- **Gaming Software:** Video games are a popular type of application software designed for entertainment. Gaming software can range from simple mobile games to complex, immersive video games. Examples: Fortnite, Minecraft, Call of Duty.
- **Mobile apps:** They are designed specifically for smartphones and tablets, offering a range of functionality from productivity to entertainment. They are typically downloaded from app stores. Examples: WhatsApp, Viber, Facebook, Google Maps, Google Drive.

## Software

- **Software** is typically realized as an application program
  - Microsoft Word is an example of software
  - Computer Games are software
  - Operating systems and device drivers are also software
- Software
  - Software is a sequence of instructions and decisions implemented in some language and translated to a form that can be executed or run on the computer.
- Computers execute very basic instructions in rapid succession
  - The basic instructions can be grouped together to perform complex tasks
- Programming is the act of designing and implementing computer programs

# Software, Program, Algorithms

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

## Introduction

- Running a program in a computer involves writing the program, translating it into machine-readable code, loading it into memory, and executing it through the CPU.
- The operating system plays a crucial role in managing resources, handling I/O, and multitasking to ensure smooth execution.
- Programs continue running until they complete their tasks or encounter an error, after which the OS cleans up the allocated resources.

# — Printing Variables

```
age = 18

if age >= 18:
    print("You are an adult.")
elif age >= 13:
    print("You are a teenager.")
else:
    print("You are a child.")
```

# Printing Variables

```
# Assign values to variables
name = "Alice"          # String
age = 25                 # Integer
height = 5.7              # Float
is_student = True         # Boolean

# Print the variables
print("Name:", name)
print("Age:", age)
print("Height:", height)
print("Is Student:", is_student)
```

# Arrays

# — Introduction

- In Python, arrays are used to store multiple values in a single variable.
- However, Python does not have built-in support for arrays as some other programming languages like C or Java do.
- Instead, Python uses lists for most array-like functionality, and if you need more specialized behavior (e.g., numerical operations), you can use the array module or libraries like NumPy.

# Lists

```
# Creating a list (array-like structure)
fruits = ["apple", "banana", "cherry"]

# Accessing elements
print(fruits[0]) # Output: apple

# Adding an element
fruits.append("orange")

# Removing an element
fruits.remove("banana")

# Printing the updated list
print(fruits) # Output: ['apple', 'cherry', 'orange']
```

- A list in Python can hold multiple values of different data types, and it's the most common data structure used for array-like behavior.

# — Key Characteristics of Lists

- Mutable: You can modify lists after creation (add, remove, or change elements).
- Heterogeneous: Lists can contain different types of data (e.g., integers, strings, floats).
- Dynamic: Lists can grow or shrink in size dynamically.



# NumPy Array

```
import numpy as np

# Creating a NumPy array
numbers = np.array([1, 2, 3, 4, 5])

# Accessing elements
print(numbers[2]) # Output: 3

# Performing operations on the array
numbers = numbers * 2 # Multiply each element by 2
print(numbers) # Output: [ 2  4  6  8 10]
```

- NumPy arrays are more powerful than the native Python lists or the array module.

# — Key Characteristics of NumPy Arrays

- Efficient and fast: NumPy arrays are much faster and more efficient than lists, especially for numerical operations on large datasets.
- Homogeneous data: All elements must be of the same data type (integers, floats, etc.).
- Supports multi-dimensional arrays: NumPy allows us to work with 2D arrays (matrices), 3D arrays, and beyond.

# — Multi-dimensional NumPy Array

```
import numpy as np

# Creating a 2D array (matrix)
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Accessing elements
print(matrix[1, 2]) # Output: 6 (second row, third column)

# Performing operations
matrix = matrix + 10 # Add 10 to every element in the array
print(matrix)
```

- NumPy arrays are more powerful than the native Python lists or the array module.

# — Arrays with the array Module

```
import array

# Creating an array of integers ('i' stands for integer type)
numbers = array.array('i', [10, 20, 30, 40])

# Accessing elements
print(numbers[1]) # Output: 20

# Adding an element
numbers.append(50)

# Removing an element
numbers.remove(20)

# Printing the updated array
print(numbers) # Output: array('i', [10, 30, 40, 50])
```

- If we specifically need an array with only one data type, we can use the array module. Arrays from this module are more memory-efficient than lists for large amounts of numerical data.

# — Conditionals

```
if condition:  
    # code block  
elif condition:  
    # code block  
else:  
    # code block
```

```
x = 5  
if x > 10:  
    print("x is greater than 10")  
elif x == 5:  
    print("x is 5")  
else:  
    print("x is less than 5")
```

- Conditionals are used to perform different actions based on conditions.

# — Loops (for, while)

```
for variable in sequence:  
    # code block
```

```
for i in range(5):  
    print(i)
```

- Loops are used to iterate over a sequence (e.g., list, string) or run a block of code repeatedly.

# — Functions

```
def function_name(parameters) :  
    # code block  
    return value
```

```
def greet(name) :  
    return f"Hello, {name}!"  
  
print(greet("Alice"))
```

- Functions allow you to define reusable blocks of code.

# Algorithms

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# — Introduction

- If you want a computer to perform a task, you start by writing an algorithm
- An **Algorithm** is:
  - a sequence (the order mattering) of actions to take to accomplish the given task
- An algorithm is like a recipe; it is a set of instructions written in a sequence that achieves a goal.
- For complex problems software developers write an algorithm before they attempt to write a computer program.
- Developing algorithms is a fundamental problem solving skill.

# — Overview

- An algorithm is a finite, well-defined sequence of computational instructions or steps designed to perform a specific task or solve a problem.
- Each step in an algorithm is precise and unambiguous, ensuring that the process can be executed either by a human or a machine, such as a computer.
- The algorithm takes input, processes it through a series of operations, and produces output, typically solving a computational problem or performing a designated function.
- Algorithms are foundational to computer science and are used to solve problems ranging from simple mathematical calculations to complex machine learning tasks.

# — Key characteristics

- Finiteness: It must terminate after a finite number of steps.
- Definiteness: Each step must be clear and unambiguous.
- Input: It takes zero or more inputs.
- Output: It produces one or more outputs.
- Effectiveness: The steps must be sufficiently simple to be performed, ideally by a machine.

## Problem Solving: Algorithm Design

- Algorithms are simply plans
  - Detailed plans that describe the steps to solve a specific problem
- You already know quite a few
  - Calculate the area of a circle
  - Find the length of the hypotenuse of a triangle
- Some problems are more complex and require more steps
  - Calculate PI to 100 decimal places
  - Calculate the trajectory of a missile

# Example

- **Algorithm FindMaximum(numbers)**

- Input: A list of numbers called "numbers"
- Output: The maximum number in the list
- Step 1: Set **max\_value** to the first element of the list (numbers[0])
- Step 2: For each element num in the list starting from the second element:
  - a. If num is greater than **max\_value**:
    - i. Set **max\_value** to num
- Step 3: Return **max\_value**

# Results

- Suppose we have the list `numbers = [3, 9, 2, 5, 6]`.
- The algorithm would:
  - Set `max_value = 3` (first element).
  - Compare 9 with `max_value` ( $9 > 3$ ), so update `max_value = 9`.
  - Compare 2 with `max_value` ( $2 < 9$ ), so no change.
  - Compare 5 with `max_value` ( $5 < 9$ ), so no change.
  - Compare 6 with `max_value` ( $6 < 9$ ), so no change.
- Finally, return `max_value = 9`, which is the maximum number in the list.

# Example

- **Problem:** Sort a list of numbers in ascending order.
- **Algorithm BubbleSort(numbers)**

Input: A list of numbers called "numbers"

Output: The list of numbers sorted in ascending order

Step 1: Set  $n$  to the length of the list "numbers"

Step 2: Repeat the following steps  $(n-1)$  times:

For  $i = 0$  to  $n-2$ :

If  $\text{numbers}[i] > \text{numbers}[i+1]$ :

Swap  $\text{numbers}[i]$  and  $\text{numbers}[i+1]$

Step 3: Return the sorted list "numbers"

# Explanation

- **Step 1: We get the length  $n$  of the list.**
- Step 2: We perform  $(n-1)$  passes over the list:
- For each pass, we go through the list from the first to the second-to-last element.
- Compare each element with the one next to it, and if it's larger, swap them.
- Step 3: After all passes, the list will be sorted in ascending order.

# Explanation

- Suppose we have the list `numbers = [5, 3, 8, 4, 2]`.
- First pass:
  - Compare 5 and 3 → Swap: [3, 5, 8, 4, 2]
  - Compare 5 and 8 → No swap
  - Compare 8 and 4 → Swap: [3, 5, 4, 8, 2]
  - Compare 8 and 2 → Swap: [3, 5, 4, 2, 8]

# Explanation

- **Second pass:**

- Compare 3 and 5 → No swap
- Compare 5 and 4 → Swap: [3, 4, 5, 2, 8]
- Compare 5 and 2 → Swap: [3, 4, 2, 5, 8]

- **Third pass:**

- Compare 3 and 4 → No swap
- Compare 4 and 2 → Swap: [3, 2, 4, 5, 8]

- **Fourth pass:**

- Compare 3 and 2 → Swap: [2, 3, 4, 5, 8]

## Algorithm: Formal Definition

An **algorithm** describes a sequence of steps that is:

1. Unambiguous
  - a. No “assumptions” are required to execute the algorithm
  - b. The algorithm uses precise instructions
2. Executable
  - a. The algorithm can be carried out in practice
3. Terminating
  - a. The algorithm will eventually come to an end, or halt

## A Simple Example

A simple algorithm to get yourself a drink of orange juice

For simplicity, the following are true:

You have a clean glass in the cabinet

You have orange juice in your refrigerator

So one valid algorithm is:

1. get a glass from your cabinet
2. go to the refrigerator and get the orange juice container
3. open the orange juice container
4. pour the orange juice from the container into the glass
5. put the orange juice container back in the refrigerator
6. drink your juice

# Variables

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# Introduction

- A variable is a symbolic name or identifier that is used to store data. It acts as a container for storing values, and these values can be of various data types, such as numbers, strings, lists, etc. Variables make it easy to refer to data and perform operations on them.
- Variables are dynamically typed: we do not need to specify the type when we create them. Variables can be reassigned, swapped, and used in different scopes (global and local). Python's flexible variable system makes it easy to handle data efficiently.

# Variable Types

- An integer is a whole number (positive, negative, or zero) without a decimal point.

```
a = 10      # Positive integer
b = -5      # Negative integer
c = 0       # Zero
```

- A float is a number that contains a decimal point or is written in exponential notation.

```
x = 3.14      # Floating-point number
y = -2.0      # Negative float
z = 1.5e3     # Scientific notation (1.5 * 10^3 = 1500.0)
```

# Variable Types

- A string is a sequence of characters enclosed within single ('...') or double ("...") quotation marks. Strings are used to represent text data.

```
name = "Alice"          # Double quotes
greeting = 'Hello'       # Single quotes
multiline_text = """This is
a multiline
string""""               # Triple quotes for multiline strings
```

## Operations

```
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name    # Concatenation
print(full_name)      # Output: John Doe

repeated = "Hello! " * 3
print(repeated)        # Output: Hello! Hello! Hello!
```

# Variable Types

- Booleans represent True or False values and are used for logical operations and conditions.

```
is_active = True      # Boolean True
is_logged_in = False # Boolean False
```

- Example:

```
a = True
b = False

print(a and b)  # Output: False
print(a or b)   # Output: True
print(not a)    # Output: False
```

# Type Checking

- We can check the type of a variable using the built-in `type()` function.

```
a = 10
b = 3.14
c = "Hello"
d = True

print(type(a)) # Output: <class 'int'>
print(type(b)) # Output: <class 'float'>
print(type(c)) # Output: <class 'str'>
print(type(d)) # Output: <class 'bool'>
```

# Dynamic Typing

```
x = 5          # x is an integer
y = "Hello"    # y is a string
z = 3.14       # z is a float
```

- Python is dynamically typed, which means we do not need to explicitly declare the type of a variable. Python automatically assigns a data type based on the value assigned to the variable.

# Variable Assignment

- Variables are assigned values using the = operator. The value on the right side of the = is assigned to the variable on the left.

```
name = "Alice"      # Assigning a string to variable 'name'  
age = 25           # Assigning an integer to variable 'age'  
height = 5.9        # Assigning a float to variable 'height'
```

- You can change the value of a variable by reassigning it.  
Variables can also change their type when reassigned.

```
x = 10      # x is an integer  
x = "Ten"    # Now x is a string (dynamic typing)
```

# Variables, Loops, and Arithmetic Operators

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# Naming Rules

- Variable names must start with a letter (a-z, A-Z) or an underscore (\_).
- They can be followed by letters, digits (0-9), or underscores.
- Variable names are case-sensitive, meaning **myVariable** and **myvariable** are different.
- Example of valid names: age, my\_variable, \_name, number1
- Invalid names: **1name**, my-variable (starts with a number or contains a hyphen).

# Naming Rules

- Must start with a letter or an underscore (\_):
  - A variable name cannot start with a number or special character.
  - Valid: my\_var, \_name, variable1
  - Invalid: 1variable, @var
- Can contain letters, numbers, and underscores (\_):
  - After the first character, you can use letters, numbers, and underscores.
  - Valid: var\_1, my\_variable\_2, \_temp\_value
  - Invalid: var@1, my-variable

# Naming Rules

- Case-sensitive:
  - Python treats variables with different cases as different variables.
  - Example: age and Age are two distinct variables.
- No spaces allowed:
  - We cannot use spaces in variable names. Use underscores (\_) instead for readability.
  - Invalid: my variable
  - Valid: my\_variable

# Naming Rules

## ■ Avoid Python keywords:

- Python has reserved words, known as keywords, which you cannot use as variable names.
- Example: **if**, **else**, **class**, **for**, **return**, etc. These are part of Python's syntax and have special meanings.
- Invalid: `if = 5, class = 'test'`

# Naming Rules

- No special characters:
  - Special characters like @, #, !, -, or \$ are not allowed in variable names.
  - Invalid: my@var, user-name
  - Valid: my\_var, user\_name

# — Descriptive Names

- Variable names should be meaningful and describe the value they hold. This makes your code more readable and easier to understand.
- Example:
  - Instead of `x = 5`, use `age = 5`.
  - Instead of `n = "Alice"`, use `name = "Alice"`.

# Best Practices

- Lowercase Letters for Variable Names.
  - By convention, variable names are written in lowercase letters with words separated by underscores (snake\_case).  
○ Example: total\_price, first\_name, user\_age
- Underscores for Readability:
  - If a variable name contains multiple words, use underscores to separate them. This improves the readability of the code.  
○ Example: student\_name, total\_sales\_amount

# Best Practices

- Uppercase Letters for Constants:
  - By convention, variables that are meant to be constants (whose values do not change) are written in all uppercase letters.
  - Example: PI = 3.14159, MAX\_VALUE = 100
- Avoid Single Letter Names:
  - Avoid using single-letter variable names like x, y, or z (unless for small loops or short-term calculations). Use more descriptive names, especially in larger programs.
  - Exception: i, j, k are commonly used in loop counters.

# Multiple Assignment and Swapping

- Python allows us to assign multiple variables in one line.

```
a, b, c = 1, 2, 3      # Assigns 1 to a, 2 to b, and 3 to c
```

- Python allows us to swap the values of two variables easily without needing a temporary variable.

```
x, y = 10, 20
x, y = y, x      # Now x is 20, and y is 10
```

# Global and Local Variables

- Variables declared inside a function are local variables, meaning they only exist within that function.
- Variables declared outside all functions are global variables, meaning they can be accessed throughout the program.

```
x = "global variable"

def my_function():
    x = "local variable"
    print(x)      # prints "local variable"

my_function()
print(x)          # prints "global variable"
```

# — Styles

- Snake Case: Words are separated by underscores, and everything is lowercase.
  - Example: total\_sales\_amount
- Camel Case: The first word is lowercase, and each subsequent word starts with an uppercase letter.
  - Example: totalSalesAmount



Image source: <https://cs50.harvard.edu/>

# — Styles

- Pascal Case: Each word starts with an uppercase letter. This is commonly used for class names.
  - Example: TotalSalesAmount



Image source: <https://khalilstemmler.com/blogs/camel-case-snake-case-pascal-case/>



# Styles



Image source: <https://cdcloudlogix.com/camel-case-vs-snake-case/>

Camel case	Snake case
{ "firstName": "John", "lastName": "Smith", "email": "john.smith@example.com", "createdAt": "2021-02-20T07:20:01", "updatedAt": "2021-02-20T07:20:01", "deletedAt": null }	{ "first_name": "John", "last_name": "Smith", "email": "john.smith@example.com", "created_at": "2021-02-20T07:20:01", "updated_at": "2021-02-20T07:20:01", "deleted_at": null }

Case Name	Example
	camelCase
	PascalCase
	snake_case
	kebab-case

by Helen Wall

# Naming in Special Context

- In loops, shorter variable names are acceptable, especially when the scope is limited. For example, i and j are commonly used in for loops:

```
for i in range(5):  
    print(i)
```

- In functions, use descriptive parameter names, as they act like variables within the function:

```
def calculate_total(price, tax_rate):  
    return price + price * tax_rate
```

# Summary

- Follow Python's naming rules (starting with letters/underscores, avoiding keywords and special characters).
- Use descriptive and readable variable names that follow `snake_case` convention for readability.
- Use uppercase letters for constants and avoid overwriting built-in functions.
- Stay consistent in your naming style, and use shorter names only in contexts where it makes sense, such as small loops.

# — Examples

## ■ Good variables

```
student_name = "Alice"      # Descriptive, follows snake_case convention
total_cost = 100.50         # Meaningful and easy to understand
MAX_CONNECTIONS = 5          # Uppercase for constants
```

## ■ Bad variables

```
x = "John"                  # Not descriptive
total$amount = 50            # Invalid due to special character $
If = "test"                  # Invalid because 'If' is a keyword
```

# — Examples

- A program to print out personal information.

```
# Assigning values to variables
name = "Alice"
age = 25
height = 5.7

# Printing variable values
print(f"My name is {name}, I am {age} years old, and my
height is {height} feet.")

# Reassigning variables
age = 26
print(f"Next year, I will be {age} years old.")
```

# Examples

## ■ A program to print out the elements of an array

```
# Define an array (list) of integers
numbers = [10, 20, 30, 40, 50]

# Use a for loop to print each element
for number in numbers:
    print(number)
```

```
# Define an array (list) of integers
numbers = [10, 20, 30, 40, 50]

# Use a while loop with an index
index = 0
while index < len(numbers):
    print(numbers[index])
    index += 1
```

- The index variable starts at 0 and increases by 1 on each iteration.
- `numbers[index]` is used to access each element in the list.
- The loop continues until the index reaches the length of the array (`len(numbers)`).

# Loops

# For Loop

- **for** is a control flow statement to iterate over a sequence of elements (e.g., a list, tuple, string, or range) and execute a block of code for each element.
- It is commonly used when we know the number of iterations in advance, as it iterates over each item in a given sequence.

```
for variable in sequence:  
    # Code block (executed for each element in the sequence)
```

- variable: A temporary variable that takes the value of each item, one at a time.
- sequence: A collection of items to iterate over, such as a list, tuple, string, range, or other iterable.

# — Examples

```
# Example: Iterating over a list of numbers
numbers = [10, 20, 30, 40]

for number in numbers:
    print(number)
```

- The list `numbers` contains four elements.
- The `for` loop iterates over each element in the list.
- On each iteration, the current element (stored in `number`) is printed.

# — Iterating over a list, text

## ■ A list

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

for fruit in fruits:
    print(fruit)
```

## ■ A text

```
text = "Hello"

for letter in text:
    print(letter)
```

# — While Loop

- The while loop in Python is used to repeatedly execute a block of code as long as a specified condition is True.
- It is often used when the number of iterations is not known beforehand and depends on some dynamic condition.

```
while condition:  
    # code block (executed repeatedly as long as the condition is True)
```

- condition: The loop keeps running as long as this condition evaluates to True. Once the condition becomes False, the loop terminates.
- Code block: This block of code is executed repeatedly until the condition is no longer true.

# Examples

```
# Example of a simple while loop
counter = 0

while counter < 5:
    print(f"Counter is: {counter}")
    counter += 1  # Increment the counter
```

- The loop starts with `counter = 0`.
- The condition `counter < 5` is True initially, so the code inside the loop is executed.
- After each iteration, the counter is incremented by 1 using `counter += 1`.
- Once `counter` reaches 5, the condition `counter < 5` becomes False, and the loop stops.

# — Endless loop

```
i = 1
while i > 0:
    print(i)
```

- If the condition never becomes False, the loop will run indefinitely, causing what's called an infinite loop.
- The condition  $i > 0$  is always True, so the loop will never stop unless you interrupt it manually.

# Example

```
# While loop to keep asking for user input until
# correct input is provided

password = ""

while password != "letmein":
    password = input("Enter the password: ")

print("Access granted!")
```

- The loop will continue to ask the user for input (Enter the password) until they type "letmein".
- When the correct password is entered, the condition password != "letmein" becomes False, and the loop terminates.

# Break

```
# While loop with break
i = 0

while i < 10:
    print(i)
    if i == 5: # Break the loop when i equals 5
        break
    i += 1
```

- The loop runs as long as  $i < 10$ .
- However, when  $i$  becomes 5, the if statement triggers the break, which immediately exits the loop, even though the condition is still True.

# Arithmetic Operations

# Introduction

- Arithmetic operations are used to perform mathematical calculations on numerical data types, such as integers (int) and floating-point numbers (float).
- Python provides a wide range of operators for performing arithmetic operations, such as addition, subtraction, multiplication, division, and more.

# Arithmetic Operations

```
x + y  # addition  
x - y  # subtraction  
x * y  # multiplication  
x / y  # division
```

```
x = 10  
y = 3  
print(x + y)  # Output: 13  
print(x - y)  # Output: 7  
print(x * y)  # Output: 30  
print(x / y)  # Output: 3.333...
```

- Python supports basic arithmetic operations such as addition, subtraction, multiplication, and division.

# Logical Operators

```
and # True if both conditions are true  
or # True if at least one condition is true  
not # Negates the condition
```

```
x = 10  
y = 5  
if x > 5 and y < 10:  
    print("Both conditions are true")
```

- Logical operators are used to combine conditional statements.

# Comparison Operators

```
and # True if both conditions are true  
or # True if at least one condition is true  
not # Negates the condition
```

```
x = 10  
y = 5  
if x > 5 and y < 10:  
    print("Both conditions are true")
```

- Logical operators are used to combine conditional statements.

# Basic Operations

## ■ Addition and Subtraction

```
a = 10
b = 5
result = a + b
print(result) # Output: 15
```

```
a = 10
b = 5
result = a - b
print(result) # Output: 5
```

## ■ Multiplication and Division

```
a = 10
b = 5
result = a * b
print(result) # Output: 50
```

```
a = 10
b = 5
result = a / b
print(result) # Output: 2.0
```

# Basic Operations

## ■ Floor Division

```
a = 10
b = 3
result = a // b
print(result) # Output: 3
```

## ■ Modulus and Exponentiation

```
a = 10
b = 3
result = a % b
print(result) # Output: 1
```

```
a = 2
b = 3
result = a ** b
print(result) # Output: 8
```

# Mixed variables

```
a = 5    # Integer
b = 2.5 # Float

result = a + b
print(result) # Output: 7.5
```

- When we perform arithmetic operations with both integers and floats, Python will automatically convert the result to a float.

# — Examples

```
x = 15
y = 4

# Addition
print(x + y) # Output: 19

# Subtraction
print(x - y) # Output: 11

# Multiplication
print(x * y) # Output: 60

# Division
print(x / y) # Output: 3.75

# Floor Division
print(x // y) # Output: 3

# Modulus
print(x % y) # Output: 3

# Exponentiation
print(x ** y) # Output: 50625
```

# Functions

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# — if condition

- the message is printed only if the age is greater than or equal to 18.

```
age = 18
if age >= 18:
    print("You are eligible to vote.")
```

- if the age is less than 18, the message in the else block is printed.

```
age = 16
if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote yet.")
```

# Using if-elif-else for Multiple Conditions

- This example checks multiple ranges of grade and prints a corresponding message.

```
grade = 85
if grade >= 90:
    print("You got an A!")
elif grade >= 80:
    print("You got a B!")
else:
    print("You need to improve.")
```

# Checking for Multiple Conditions with and and or

- In this example, both conditions (`age >= 18` and `is_registered == True`) must be True for the message "You can vote!" to be printed.

```
age = 20
is_registered = True

if age >= 18 and is_registered:
    print("You can vote!")
else:
    print("You cannot vote.")
```

- This checks if "apple" is present in the fruits list.

```
fruits = ["apple", "banana", "cherry"]
if "apple" in fruits:
    print("Apple is in the list.")
```

# Nested if Statements

- The nested if statement checks an additional condition after the age is evaluated.

```
age = 25
is_student = False

if age < 18:
    print("You are a minor.")
else:
    if is_student:
        print("You are an adult student.")
    else:
        print("You are an adult, not a student.")
```

# while

- The loop will keep asking the user to input a number until they provide one that is 10 or greater.

```
number = 0
while number < 10:
    number = int(input("Enter a number greater than or equal to 10: "))
    print("Thank you!")
```

- Using break

```
number = 1
while True:
    print(f"Current number: {number}")
    number += 1
    if number > 5:
        break # Exits the loop when the number exceeds 5
```



# — while with a Counter for Limited Attempts

```
attempts = 0
while attempts < 3:
    password = input("Enter the password: ")
    if password == "secret123":
        print("Access granted!")
        break
    attempts += 1
else:
    print("Too many attempts. Access denied.")
```

- The loop allows the user three attempts to enter the correct password. If they fail after three attempts, the else block runs and denies access.

# — for with else

```
for number in range(3):  
    print(number)  
else:  
    print("Loop finished!")
```

- After printing the numbers 0, 1, and 2, the else block executes, printing "Loop finished!"

# Nested for Loops

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
for row in matrix:  
    for value in row:  
        print(value, end=" ")  
    print() # For a new line after each row
```

- The outer loop iterates over each row in the matrix, while the inner loop prints each value in the row.

# Functions

# — Introduction

- Functions in Python are reusable blocks of code designed to perform specific tasks.
- They help to organize the code and make it more modular, readable, and efficient.
- Functions are defined using the **def** keyword, followed by the function name, parentheses, and a colon. Inside the function, the block of code to be executed is indented.

# — Introduction

- Functions are essential for organizing and structuring Python code. They promote reusability, make code more readable, and allow us to break down complex problems into smaller, manageable tasks.
- Whether using built-in functions, writing our own, or leveraging lambda expressions, understanding how to use functions effectively is key to mastering Python.

# — Types of functions

- Built-in Functions: Python has a variety of built-in functions like `print()`, `len()`, and `max()`.
- User-Defined Functions: These are the functions you define yourself using **def**.
- Anonymous Functions (Lambda Functions): These are small, unnamed functions created with the `lambda` keyword.

# Syntax

- Functions in Python are reusable blocks of code designed to perform specific tasks.

```
def function_name(parameters):  
    # Code block  
    return value # Optional, returns a result
```

- Example

```
def greet(name):  
    print(f"Hello, {name}!")
```

# Functions (2)

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# Calling a Function

- Functions can accept parameters, which are placeholders for the values that the function will operate on.

```
def greet(name="Guest"):  
    print(f"Hello, {name}!")  
  
greet()  # Output: Hello, Guest!  
greet("Bob")  # Output: Hello, Bob!
```

```
def add(a, b):  
    return a + b  
  
result = add(3, 4)  
print(result)  # Output: 7
```

# Function scope

- Functions in Python have their own scope, meaning variables defined inside a function are local to that function and cannot be accessed from outside it.

```
def my_function():
    x = 10  # Local variable
    print(x)

my_function()  # Output: 10
# print(x)  # Error: NameError: name 'x' is not defined
```

# Examples

# Recursive Function

- A recursive function that calculates the factorial of a number.

```
def factorial(n):  
    if n == 1:  
        return 1  
    else:  
        return n * factorial(n - 1)  
  
# Example usage  
print(factorial(5)) # Output: 120
```

# Function using for and if

- The for loop iterates over the numbers list, and the if statement checks whether each number is even. If it is even, it is added to total.

```
def sum_of_evens(numbers):
    total = 0
    for num in numbers:
        if num % 2 == 0:
            total += num
    return total

# Example usage
numbers = [1, 2, 3, 4, 5, 6]
print(sum_of_evens(numbers)) # Output: 12 (2 + 4 + 6)
```

# Function using while and if

- The for loop iterates over the numbers list, and the if statement checks whether each number is even. If it's even, it's added to total.

```
def count_negatives():
    count = 0
    while True:
        num = int(input("Enter a number (0 to stop): "))
        if num == 0:
            break
        elif num < 0:
            count += 1
    return count

# Example usage
print(count_negatives()) # Example output: If the user enters -1, -3, 5, 0 -> Output: 2
```

# — Returning Multiple Values

- A function that calculates both the quotient and remainder of a division and returns both values.

```
def divide(dividend, divisor):  
    quotient = dividend // divisor  
    remainder = dividend % divisor  
    return quotient, remainder  
  
# Example usage  
q, r = divide(10, 3)  
print(f"Quotient: {q}, Remainder: {r}") # Output:  
Quotient: 3, Remainder: 1
```

# Function with a for Loop

- A function that returns the sum of all elements in a list using a for loop.

```
def sum_list(numbers):  
    total = 0  
    for num in numbers:  
        total += num  
    return total  
  
# Example usage  
print(sum_list([1, 2, 3, 4])) # Output: 10
```

# Reverse a string and count vowels

- This function takes a string and returns its reverse.

```
def reverse_string(s):  
    return s[::-1]  
  
# Example usage  
print(reverse_string("Python")) # Output: nohtyP
```

- Counts the number of vowels (a, e, i, o, u) in a given string.

```
def count_vowels(s):  
    vowels = "aeiouAEIOU"  
    count = 0  
    for char in s:  
        if char in vowels:  
            count += 1  
    return count  
  
# Example usage  
print(count_vowels("hello world")) # Output: 3
```

# Find the Maximum in a List

- This function takes a list of numbers and returns the maximum value.

```
def find_max(numbers):  
    max_value = numbers[0]  
    for num in numbers:  
        if num > max_value:  
            max_value = num  
    return max_value  
  
# Example usage  
print(find_max([10, 20, 5, 8, 99, 56])) # Output: 99
```

# Function with if-else Statement

- A function that categorizes a number as positive, negative, or zero.

```
def categorize_number(n):  
    if n > 0:  
        return "Positive"  
    elif n < 0:  
        return "Negative"  
    else:  
        return "Zero"  
  
# Example usage  
print(categorize_number(5))    # Output: Positive  
print(categorize_number(-3))   # Output: Negative  
print(categorize_number(0))    # Output: Zero
```

# Finding the maximum value in a matrix

```
def find_max_value(matrix):
    # Initialize a variable to store the maximum value, set to the first element of the matrix
    max_value = matrix[0][0]

    # Iterate through each row in the matrix
    for row in matrix:
        # Iterate through each element in the row
        for element in row:
            # Update max_value if a larger element is found
            if element > max_value:
                max_value = element

    return max_value

# Example usage:
matrix = [
    [1, 4, 7],
    [2, 5, 8],
    [3, 6, 9]
]

max_val = find_max_value(matrix)
print(f"The maximum value in the matrix is: {max_val}")
```

# Explanation

- Initial Value: We start by setting the `max_value` variable to the first element in the matrix (`matrix[0][0]`).
- Nested Loops: The outer for loop iterates over each row of the matrix, and the inner for loop iterates over each element in that row.
- Comparison: If any element is greater than the current `max_value`, we update `max_value`.
- Return Value: Once all elements have been checked, the function returns the maximum value found.

# — Adding two matrices

```
def add_matrices(matrix1, matrix2):
    # Get the dimensions of the matrices
    rows = len(matrix1)
    cols = len(matrix1[0])

    # Initialize the result matrix with zeros
    result = [[0 for _ in range(cols)] for _ in
range(rows)]

    # Iterate through each element and add corresponding
elements
    for i in range(rows):
        for j in range(cols):
            result[i][j] = matrix1[i][j] + matrix2[i][j]

    return result
```

```
# Example usage:
matrix1 = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

matrix2 = [
    [9, 8, 7],
    [6, 5, 4],
    [3, 2, 1]
]

result = add_matrices(matrix1, matrix2)

# Print the result matrix
for row in result:
    print(row)
```

# Explanation

- Dimensions Check: The function assumes both matrices have the same dimensions.
- Result Matrix Initialization: We initialize an empty matrix result filled with zeros of the same size as matrix1 and matrix2.
- Nested Loops: We use two nested loops: the outer loop iterates through rows, and the inner loop iterates through columns, adding corresponding elements of matrix1 and matrix2.
- Return Value: The function returns the resulting matrix containing the sum of the two input matrices.

# Testing in Python

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# — Introduction

- Testing in Python is an essential practice that ensures code quality, reliability, and maintainability.
- Effective testing catches bugs early, verifies that new changes don't break existing functionality, and helps developers produce robust code.
- Python offers various tools and methodologies for testing, with popular libraries like **unittest**, **pytest**, and **doctest** widely used in different scenarios.

# Without Testing

- A practical example to illustrate how errors can go undetected without proper testing.
- Suppose we are developing a simple e-commerce checkout system that calculates the total price after applying a discount.
- Without testing, this function might introduce errors due to incorrect assumptions, edge cases, or unexpected input values.

# Example

```
def apply_discount(price, discount):  
    return price - (price * discount / 100)  
  
def calculate_total(items, discount):  
    total = sum(item["price"] * item["quantity"] for item in items)  
    total_with_discount = apply_discount(total, discount)  
    return total_with_discount
```

- If a 100% discount is applied, `apply_discount` should ideally set the price to zero. Without testing, this could lead to issues, such as the total going negative.
- The function assumes valid, positive prices and discounts. Without testing, we might miss that users can enter negative values, which would lead to incorrect calculations.

# Example

```
def apply_discount(price, discount):
    return price - (price * discount / 100)

def calculate_total(items, discount):
    total = sum(item["price"] * item["quantity"] for item in items)
    total_with_discount = apply_discount(total, discount)
    return total_with_discount
```

- If the items list is empty, the function should return zero. Without testing, this might raise an error or return an unexpected result.

# — Benefit of Testing

- Increased Code Reliability: Testing ensures that the checkout calculations are correct for various scenarios.
- Better User Experience: By preventing errors like negative totals, we ensure that users will not encounter unexpected behavior.
- Easier Maintenance: Tests make it safer to modify or refactor the code in the future because they catch regressions early.

# Unit Testing

- Purpose: Unit testing is focused on testing individual units or components in isolation (usually functions or methods).
- Tools: The most common tool is **unittest**, Python's built-in testing framework, along with pytest, which offers a simpler syntax and enhanced features.
- Example: Testing a single function that adds two numbers. Unit tests verify that each function behaves as expected for different inputs, including edge cases.

# Basic Testing with Assertions

- Assertions are the simplest way to test a function.
- We call the function with expected inputs and use the assert statement to check that the output matches the expected result.

```
def add(a, b):  
    return a + b  
  
# Testing with assert  
assert add(2, 3) == 5  
assert add(-1, 1) == 0
```

- Explanation: If an assert statement fails, it raises an AssertionError, signaling that the function did not return the expected result.

# — Assertions

- In Python's unittest framework (and also in pytest), assertions are used to verify conditions and check if the code behaves as expected.

**assertEqual(a, b)**: Checks if **a** is equal to **b**.

**assertNotEqual(a, b)**: Checks if **a** is not equal to **b**.

```
def test_numbers():
    assert 2 + 3 == 5
    assert 10 != 5 # This assertion will pass because 10 is not equal to 5
```

# True and False Assertions

- In Python's unittest framework (and also in pytest), assertions are used to verify conditions and check if the code behaves as expected.

**assertTrue(x)**: Checks if `x` is `True`.

**assertFalse(x)**: Checks if `x` is `False`.

```
def test_boolean():
    assert True
    assert not False
    assert 5 > 3 # True, so the assertion passes
    assert not (2 > 3) # False, so the assertion passes
```

# Using the `unittest` Library

- **unittest** is Python's built-in library for creating and running tests. It allows for organizing tests into classes, running multiple tests together, and reporting results in a structured way.
- **unittest** provides a structured approach to write, organize, and run tests to verify that individual pieces of code (often functions or methods) work as expected.

# Using the unittest Library

```
# Define the function to test
def multiply(a, b):
    return a * b

# Import unittest
import unittest

# Define the test case
class TestMathFunctions(unittest.TestCase):
    def test_multiply_positive_numbers(self):
        self.assertEqual(multiply(3, 4), 12)
    def test_multiply_with_zero(self):
        self.assertEqual(multiply(0, 10), 0)
    def test_multiply_negative_numbers(self):
        self.assertEqual(multiply(-1, 5), -5)
# Run the tests
if __name__ == "__main__":
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

- `argv=['first-arg-is-ignored']`: This ensures that the arguments passed by Google Colab don't interfere with unittest.
- `exit=False`: Prevents Colab from terminating the kernel after running the tests.

# assertGreater, assertGreaterEqual, assertLess, and assertLessEqual

```
import unittest

# Example functions to test
def get_discount(price):
    if price > 100:
        return 20
    return 5

def calculate_age(birth_year, current_year):
    return current_year - birth_year

class TestComparisonAssertions(unittest.TestCase):
    def test_assertGreater(self):
        # Assert that price over 100 gives a greater discount
        price = 150
        discount = get_discount(price)
        self.assertGreater(discount, 10, "Discount should be greater than 10 for price over 100")

    def test_assertGreaterEqual(self):
        # Assert that the discount for price exactly 100 is at least 5
        price = 100
        discount = get_discount(price)
        self.assertGreaterEqual(discount, 5, "Discount should be at least 5")
```

# assertGreater, assertGreaterEqual, assertLess, and assertLessEqual

```
def test_assertLess(self):
    # Assert that someone born after 2000 has an age less than 25
    age = calculate_age(2005, 2023)
    self.assertLess(age, 25, "Age should be less than 25 for birth year
after 2000")

def test_assertLessEqual(self):
    # Assert that someone born in 2000 has an age less than or equal to 23
    age = calculate_age(2000, 2023)
    self.assertLessEqual(age, 23, "Age should be 23 or less for birth year
2000")

# Running the tests
if __name__ == "__main__":
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

# Explanation

- **assertGreater(a, b)**: Checks if **a** is greater than **b**.
  - In **test\_assertGreater**, we test that **get\_discount(150)** gives a discount greater than **10** when the price is over **100**.
  
- **assertGreaterEqual(a, b)**: Checks if **a** is greater than or equal to **b**
  - In **test\_assertGreaterEqual**, we check that a price of exactly **100** gives a discount that is at least **5**.

# Explanation

- **assertLess(a, b)**: Checks if `a` is less than `b`.
  - In `test_assertLess`, we verify that the age calculated for someone born in 2005 (age 18 in 2023) is less than `25`.
  
- **assertLessEqual(a, b)**: Checks if `a` is less than or equal to `b`
  - In `test_assertLessEqual`, we confirm that someone born in 2000 has an age of `23` or less in 2023.

# — Types of assertions

Assertion	Description
<code>assert a == b</code>	Checks if <code>a</code> is equal to <code>b</code>
<code>assert a != b</code>	Checks if <code>a</code> is not equal to <code>b</code>
<code>assert a in b</code>	Checks if <code>a</code> is in <code>b</code> (useful for lists, dictionaries, strings)
<code>assert a not in b</code>	Checks if <code>a</code> is not in <code>b</code>
<code>assert a &gt; b</code>	Checks if <code>a</code> is greater than <code>b</code>
<code>assert a &gt;= b</code>	Checks if <code>a</code> is greater than or equal to <code>b</code>
<code>assert a &lt; b</code>	Checks if <code>a</code> is less than <code>b</code>
<code>assert a &lt;= b</code>	Checks if <code>a</code> is less than or equal to <code>b</code>
<code>assert isinstance(a, T)</code>	Checks if <code>a</code> is an instance of type <code>T</code>
<code>assert a is b</code>	Checks if <code>a</code> and <code>b</code> are the same object
<code>pytest.raises(Error)</code>	Asserts that a block of code raises a specified exception
<code>pytest.approx(value)</code>	Asserts that a floating-point result is approximately equal to <code>value</code>

# — pytest

- **pytest** is a popular, third-party testing framework for Python known for its simplicity, powerful features, and easy-to-read syntax.
- It is widely used in both small projects and large-scale software development due to its flexibility and extensibility.
- **pytest** is a powerful yet simple testing framework for Python, making it popular for projects of all sizes.
- With its straightforward syntax, built-in support for fixtures, parameterized tests, and an extensive plugin ecosystem, **pytest** enables both beginners and advanced users to write robust, scalable tests with minimal setup.

# Example with a bank account

# Bank Account Class

- A BankAccount class that includes methods to deposit, withdraw, and check the account balance.

```
# bank_account.py

class BankAccount:
    def __init__(self, initial_balance=0):
        self.balance = initial_balance

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")
        self.balance += amount

    def withdraw(self, amount):
        if amount > self.balance:
            raise ValueError("Insufficient funds")
        self.balance -= amount

    def get_balance(self):
        return self.balance
```

# Test Cases

```
# test_bank_account.py

import unittest
from bank_account import BankAccount

class TestBankAccount(unittest.TestCase):

    def setUp(self):
        """Set up a bank account with an initial balance for each test."""
        self.account = BankAccount(100)

    def test_deposit_increases_balance(self):
        """Check that depositing increases the balance."""
        self.account.deposit(50)
        self.assertGreater(self.account.get_balance(), 100, "Balance should be greater than initial balance after deposit")

    def test_withdraw_decreases_balance(self):
        """Check that withdrawing decreases the balance."""
        self.account.withdraw(30)
        self.assertLess(self.account.get_balance(), 100, "Balance should be less than initial balance after withdrawal")
```

# Test Cases

```
def test_minimum_balance(self):
    """Ensure minimum balance remains zero or above."""
    self.account.withdraw(100)
    self.assertGreaterEqual(self.account.get_balance(), 0, "Balance should be zero or more after full
withdrawal")

def test_deposit_boundaries(self):
    """Ensure that large deposits increase the balance appropriately."""
    self.account.deposit(1000)
    self.assertGreaterEqual(self.account.get_balance(), 1100, "Balance should be at least 1100 after large
deposit")

def test_withdraw_boundary(self):
    """Ensure that withdrawing up to the balance leaves the balance at zero."""
    self.account.withdraw(100)
    self.assertLessEqual(self.account.get_balance(), 0, "Balance should be zero or less after withdrawing
full balance")

def test_overdraft_prevention(self):
    """Ensure overdraft is prevented."""
    with self.assertRaises(ValueError, msg="Should raise ValueError when attempting to overdraft"):
        self.account.withdraw(150)

if __name__ == "__main__":
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

# Explanation

## ■ `test_deposit_increases_balance`:

- Checks that depositing an amount increases the balance above the initial balance of `100` using `assertGreater`.

## ■ `test_withdraw_decreases_balance`:

- Verifies that a withdrawal decreases the balance, ensuring it becomes less than the initial balance using `assertLess`.

## ■ `test_minimum_balance`:

- Ensures that the balance does not drop below zero after a full withdrawal using `assertGreaterEqual`. This test checks the lower boundary condition to prevent overdraft.

# Explanation

## ■ `test_deposit_boundaries`:

- Verifies that a large deposit brings the balance to at least `1100` using `assertGreaterEqual`. This ensures that the balance increases as expected for larger values.

## ■ `test_withdraw_boundary`:

- Ensures that withdrawing the exact balance brings the account to zero or less using `assertLessEqual`, confirming the balance is not negative.

## ■ `test_overdraft_prevention`:

- Uses `assertRaises` to check that a `ValueError` is raised when attempting to withdraw more than the available balance, preventing overdraft.

# Data Management

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# Application of Data Science

- Data Science is a dynamic field that continues to evolve with advancements in technology
- It plays a crucial role in extracting meaningful insights from the vast amounts of data generated in today's digital world
- Business Analytics: Predictive modeling and data-driven decision-making for businesses

# Application of Data Science

- Healthcare: Disease prediction, patient outcome analysis, and personalized medicine
- Finance: Fraud detection, risk assessment, and algorithmic trading
- E-commerce: Recommender systems, customer segmentation, and demand forecasting
- Social Media: Sentiment analysis, user behavior analysis, and content recommendation

# Libraries and Platforms

# — Libraries and platforms



- There are various libraries in Python, ready to be used for different tasks
- Students are encouraged to use these libraries/platforms for their projects

# Matplotlib



- Used for creating high-quality, publication-ready visualizations and plots
- Provides a comprehensive set of tools for creating various types of charts, graphs, and figures to visualize data
- Matplotlib is a fundamental library in the data science and scientific computing ecosystem and is often used in combination with libraries like NumPy and Pandas

# Matplotlib (2)

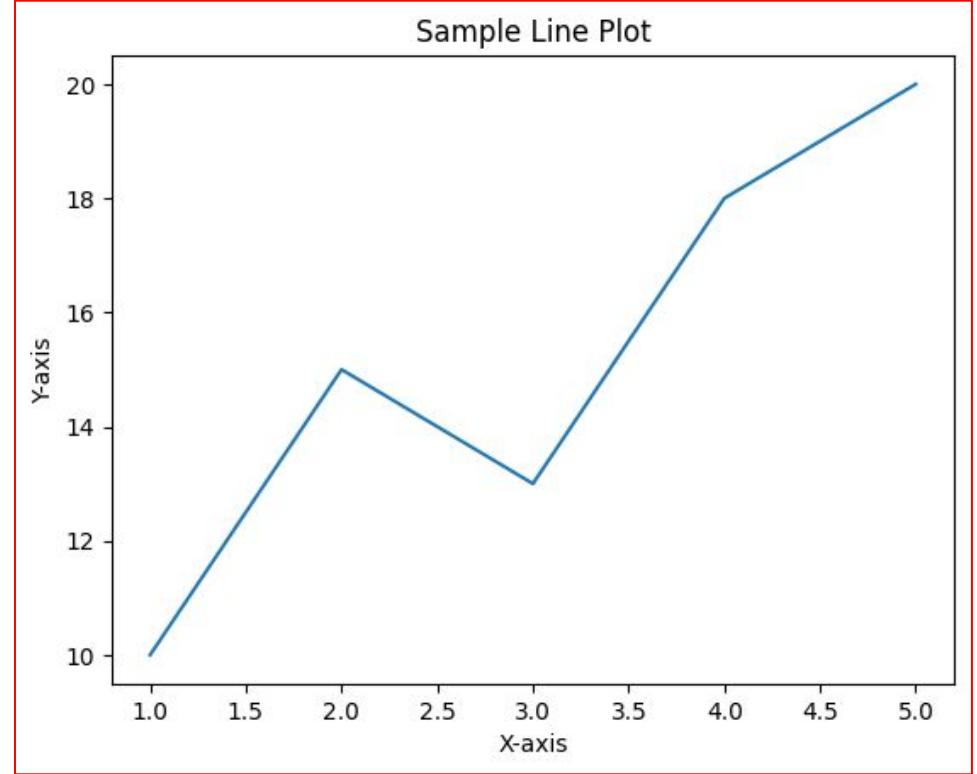
```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 15, 13, 18, 20]

# Create a line plot
plt.plot(x, y)

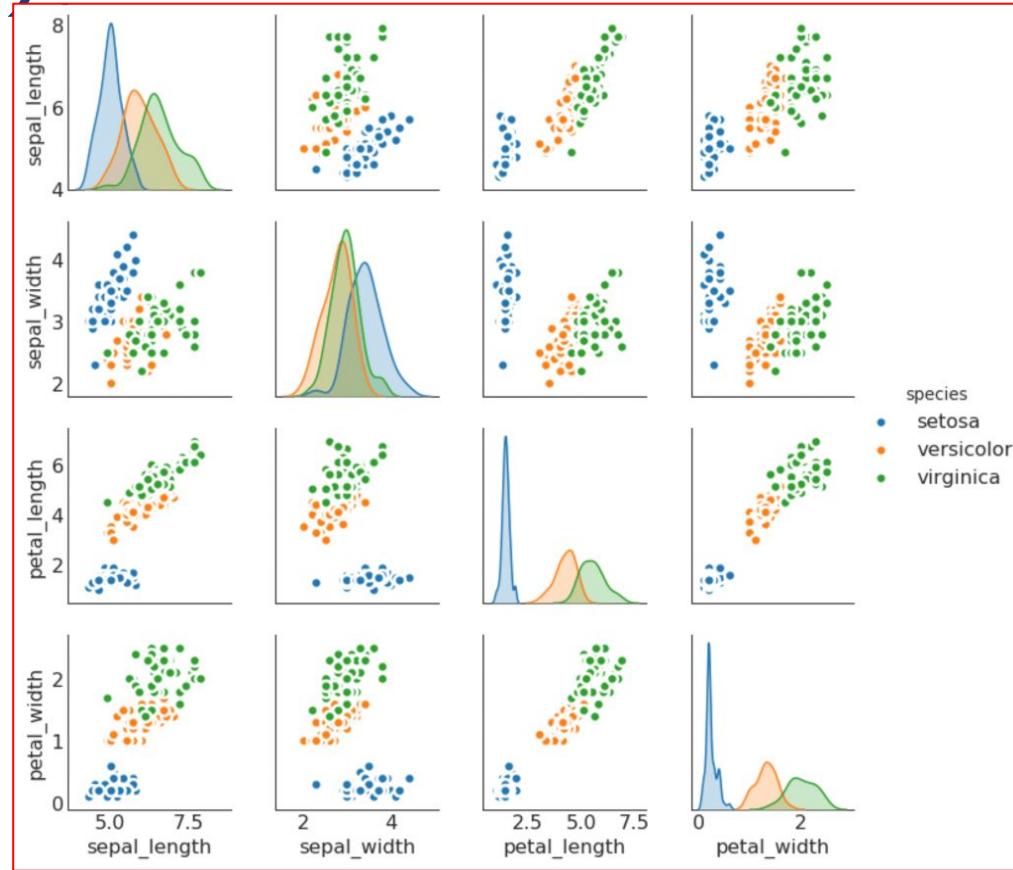
# Customize plot labels and title
plt.xlabel('Day')
plt.ylabel('Temperature')
plt.title('Temperature over the week')

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



- Highly extensible, allowing users to create custom plot types, combine multiple plots, or add new functionalities via plugins

# Matplotlib (3)



- Highly extensible, allowing users to create custom plot types, combine multiple plots, or add new functionalities via plugins

# Pandas



- Python Data Analysis Library: A popular and powerful Python library for data manipulation and analysis
- It provides easy-to-use data structures and functions for working with structured data, such as spreadsheets or SQL tables
- Pandas is widely used in data science, data analysis, and machine learning tasks

# Pandas (2)

```
import pandas as pd

# Read data from a CSV file into a DataFrame
df = pd.read_csv('data.csv')

# Display the first few rows of the DataFrame
print(df.head())

# Perform data selection and filtering
selected_data = df[df['Category'] == 'Science']

# Calculate statistics
mean_price = selected_data['Price'].mean()
print(f'Mean price of Science books: ${mean_price:.2f}')
```

## ■ Read and manipulate data from a CSV file

# Numpy



- Stands for “Numerical Python,” a fundamental Python library for numerical and scientific computing
- Provides support for arrays and matrices, along with a collection of mathematical functions to operate on these arrays
- A foundational library in the Python ecosystem, and serves as the basis for many other scientific and data science libraries

# Numpy (2)

```
import numpy as np

# Create NumPy arrays
arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([5, 4, 3, 2, 1])

# Perform element-wise addition
result = arr1 + arr2

# Compute the mean of the array
mean_value = np.mean(result)

print(result)  # Output: [6 6 6 6 6]
print(mean_value)  # Output: 6.0
```

- Primary data structure is the NumPy array (or ndarray)-a multi-dimensional, homogeneously typed array
- Arrays are more efficient for numerical operations than standard Python lists

# Scikit-learn



- A versatile and widely-used Python library for machine learning and data mining
- Provides a rich set of tools for various machine learning tasks, including classification, regression, clustering, dimensionality reduction, model selection, and more
- Built on top of other foundational libraries like NumPy and SciPy, and it offers a consistent API for working with machine learning algorithms

# — Google Colab



- A platform for running Machine Learning, Python code
- It does not require the installation of various software packages
- Advantages: It allows developers to run Python notebooks easily
- Limited access to GPU; Developers can run a notebook for a maximum period of 12 hours

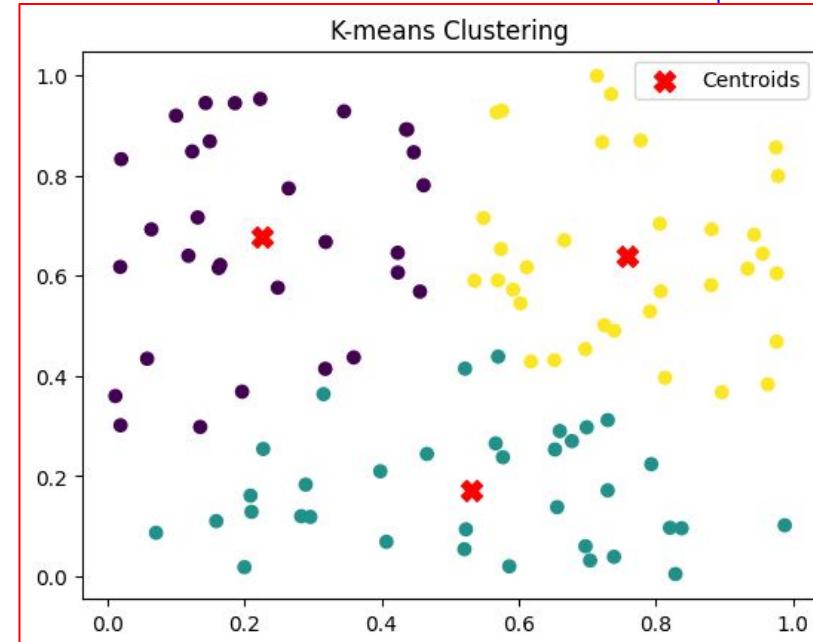
More information: <https://research.google.com/colaboratory/faq.html#resource-limits>

# Scikit-learn (2)

```

import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
# Generate synthetic data for clustering
np.random.seed(0)
X = np.random.rand(100, 2)
# Create a K-means model with 3 clusters
kmeans = KMeans(n_clusters= 3)
# Fit the model to the data
kmeans.fit(X)
# Get cluster assignments for each data point
labels = kmeans.labels_
# Get cluster centroids
centroids = kmeans.cluster_centers_
# Plot the data points and cluster centroids
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.scatter(centroids[:, 0], centroids[:, 1], c='red', s=100, marker='X', label='Centroids')
plt.title('K-means Clustering')
plt.legend()
plt.show()

```



# — Seaborn



- A popular Python data visualization library built on top of Matplotlib
- It provides an interface for creating informative and attractive statistical graphics
- Seaborn is particularly well-suited for visualizing complex datasets and statistical relationships in a simple and efficient way



# Seaborn (2)

```

import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

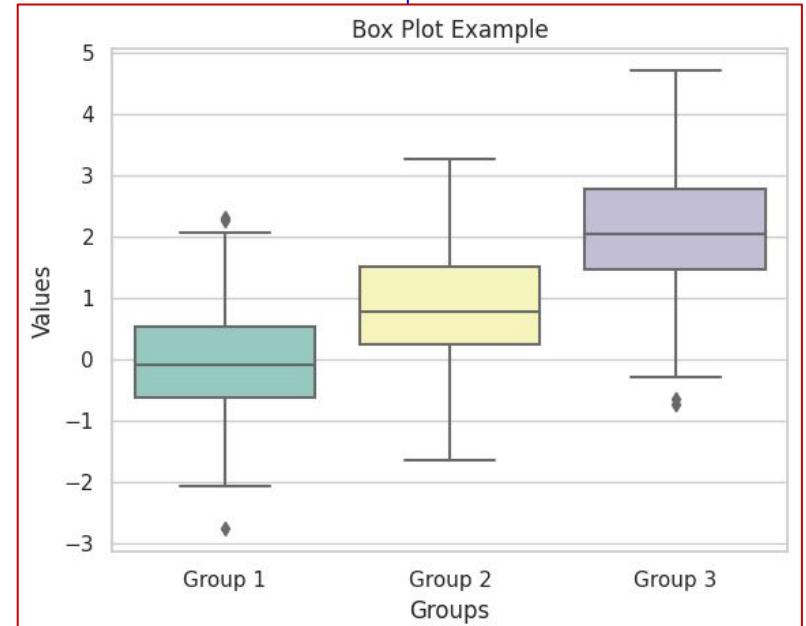
# Generate random data for three groups
group1 = np.random.normal(0, 1, 100)
group2 = np.random.normal(1, 1, 100)
group3 = np.random.normal(2, 1, 100)

# Combine the data into a single DataFrame
data = np.concatenate([group1, group2, group3])
labels = ['Group 1'] * 100 + ['Group 2'] * 100 + ['Group 3'] * 100

# Create a box plot (replace sns.violinplot with sns.boxplot)
sns.set(style="whitegrid") # Set the style
sns.boxplot(x=labels, y=data, palette="Set3") # Create the box plot

# Customize the plot
plt.title("Box Plot Example")
plt.xlabel("Groups")
plt.ylabel("Values")
# Show the plot
plt.show()

```



# — Introduction

- Data management involves tasks related to the organization, storage, retrieval, and manipulation of data
- Python provides several libraries and tools for effective data management
- Moreover, Python offers built-in data structures like lists, tuples, dictionaries, and sets
- Libraries like NumPy and Pandas provide advanced data structures like arrays and DataFrames, which are particularly useful for numerical and tabular data, respectively

# Data Visualization

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# Data Analysis Process

- Requirement specification.
- Data collection.
- Data preprocessing.
- Data analysis.
- Data storytelling.

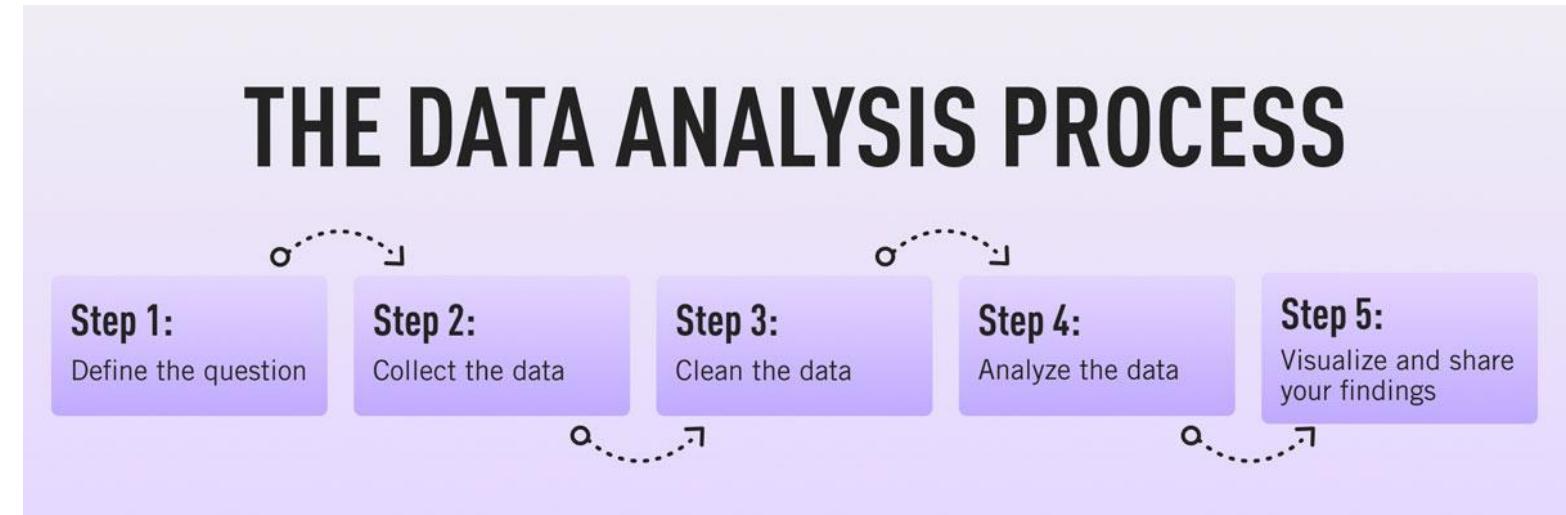
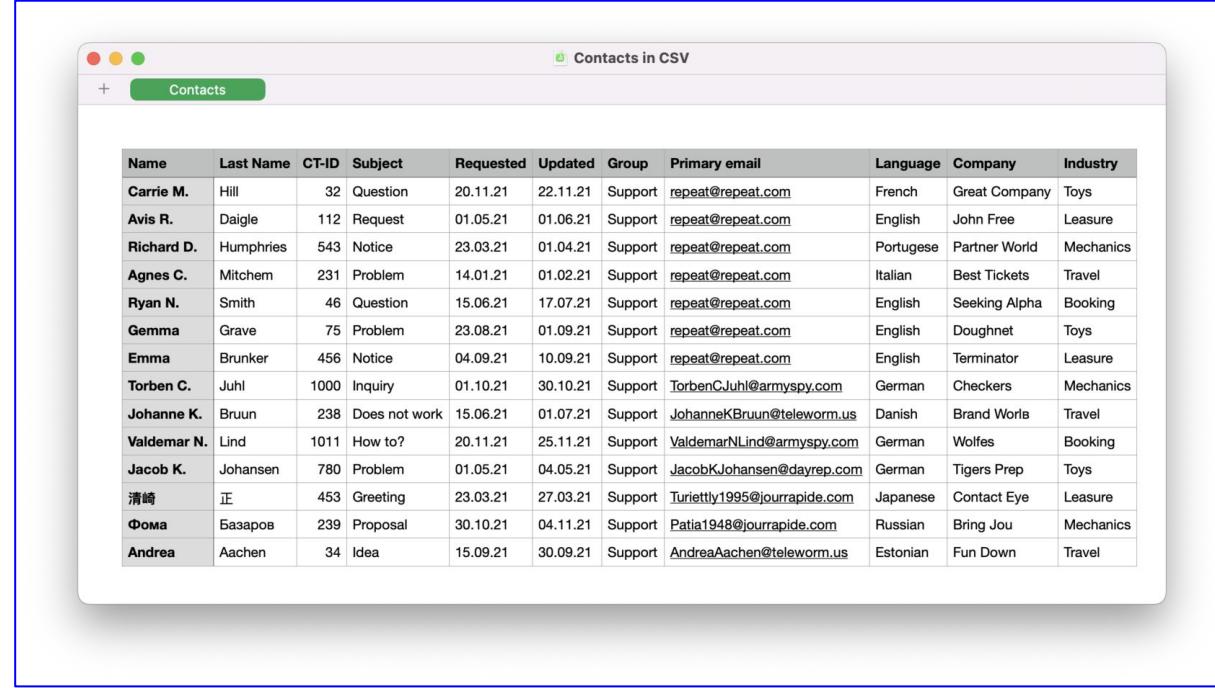


Image source: <https://careerfoundry.com/en/blog/data-analytics/the-data-analysis-process-step-by-step/>

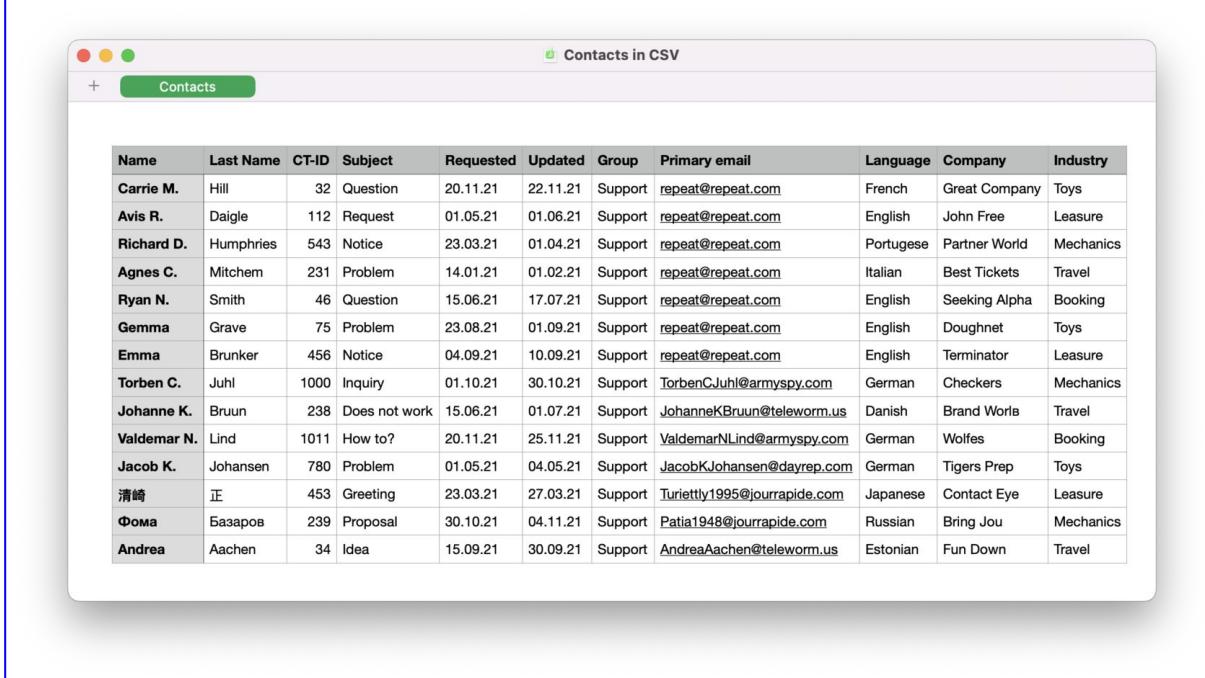
# The CSV Format



Name	Last Name	CT-ID	Subject	Requested	Updated	Group	Primary email	Language	Company	Industry
Carrie M.	Hill	32	Question	20.11.21	22.11.21	Support	repeat@repeat.com	French	Great Company	Toys
Avis R.	Daigle	112	Request	01.05.21	01.06.21	Support	repeat@repeat.com	English	John Free	Leasure
Richard D.	Humphries	543	Notice	23.03.21	01.04.21	Support	repeat@repeat.com	Portugese	Partner World	Mechanics
Agnes C.	Mitchem	231	Problem	14.01.21	01.02.21	Support	repeat@repeat.com	Italian	Best Tickets	Travel
Ryan N.	Smith	46	Question	15.06.21	17.07.21	Support	repeat@repeat.com	English	Seeking Alpha	Booking
Gemma	Grave	75	Problem	23.08.21	01.09.21	Support	repeat@repeat.com	English	Doughnet	Toys
Emma	Brunker	456	Notice	04.09.21	10.09.21	Support	repeat@repeat.com	English	Terminator	Leasure
Torben C.	Juhl	1000	Inquiry	01.10.21	30.10.21	Support	TorbenCJuhl@armyspy.com	German	Checkers	Mechanics
Johanne K.	Bruun	238	Does not work	15.06.21	01.07.21	Support	JohanneKBruun@teleworm.us	Danish	Brand Worls	Travel
Valdemar N.	Lind	1011	How to?	20.11.21	25.11.21	Support	ValdemarNLind@armyspy.com	German	Wolfs	Booking
Jacob K.	Johansen	780	Problem	01.05.21	04.05.21	Support	JacobKJohansen@dayrep.com	German	Tigers Prep	Toys
清崎	正	453	Greeting	23.03.21	27.03.21	Support	Turietylly1995@jourrapide.com	Japanese	Contact Eye	Leasure
Фома	Базаров	239	Proposal	30.10.21	04.11.21	Support	Patia1948@jourrapide.com	Russian	Bring Jou	Mechanics
Andrea	Aachen	34	Idea	15.09.21	30.09.21	Support	AndreaAachen@teleworm.us	Estonian	Fun Down	Travel

- Comma-Separated Values: A plain text file format that stores tabular data (numbers and text) in plain text form, where each line of the file is a data record, and fields within each record are separated by commas.
- Delimiter: While the default delimiter is a comma, other delimiters like semicolons (;) or tabs (\t) can also be used, depending on the application or region.

# CSV



Name	Last Name	CT-ID	Subject	Requested	Updated	Group	Primary email	Language	Company	Industry
Carrie M.	Hill	32	Question	20.11.21	22.11.21	Support	repeat@repeat.com	French	Great Company	Toys
Avis R.	Daigle	112	Request	01.05.21	01.06.21	Support	repeat@repeat.com	English	John Free	Leasure
Richard D.	Humphries	543	Notice	23.03.21	01.04.21	Support	repeat@repeat.com	Portugese	Partner World	Mechanics
Agnes C.	Mitchem	231	Problem	14.01.21	01.02.21	Support	repeat@repeat.com	Italian	Best Tickets	Travel
Ryan N.	Smith	46	Question	15.06.21	17.07.21	Support	repeat@repeat.com	English	Seeking Alpha	Booking
Gemma	Grave	75	Problem	23.08.21	01.09.21	Support	repeat@repeat.com	English	Doughnet	Toys
Emma	Brunker	456	Notice	04.09.21	10.09.21	Support	repeat@repeat.com	English	Terminator	Leasure
Torben C.	Juhl	1000	Inquiry	01.10.21	30.10.21	Support	TorbenCJuhl@armyspy.com	German	Checkers	Mechanics
Johanne K.	Bruun	238	Does not work	15.06.21	01.07.21	Support	JohanneKBruun@teleworm.us	Danish	Brand Worls	Travel
Valdemar N.	Lind	1011	How to?	20.11.21	25.11.21	Support	ValdemarNLind@armyspy.com	German	Wolfs	Booking
Jacob K.	Johansen	780	Problem	01.05.21	04.05.21	Support	JacobKJohansen@dayrep.com	German	Tigers Prep	Toys
清崎 正		453	Greeting	23.03.21	27.03.21	Support	Turietty1995@journapide.com	Japanese	Contact Eye	Leasure
Фома	Базаров	239	Proposal	30.10.21	04.11.21	Support	Patia1948@journapide.com	Russian	Bring Jou	Mechanics
Andrea	Aachen	34	Idea	15.09.21	30.09.21	Support	AndreaAachen@teleworm.us	Estonian	Fun Down	Travel

- Header Row: CSV files often have a header row at the beginning, which contains the names of the columns.
- CSV files are widely supported and can be opened and edited using spreadsheet software like Microsoft Excel or Google Sheet.

# File Input/Output

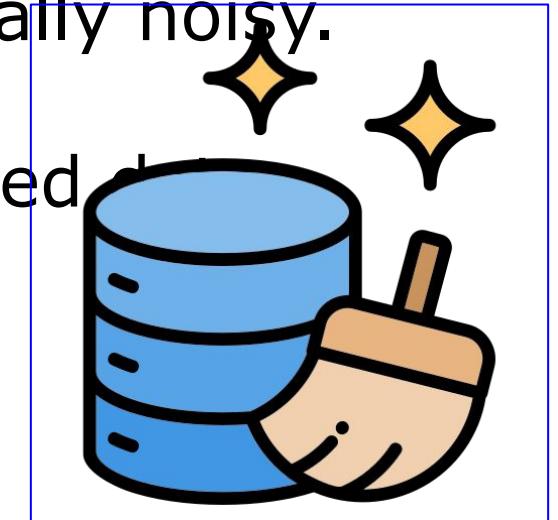
```
import csv
with open('data.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

```
from google.colab import drive
drive.mount('/content/drive/', force_remount=False)
DATA_PATH = '/content/drive/My Drive/Colab Notebooks/MNIST/'
from os.path import join
input_file = join(DATA_PATH, 'data.csv')
input_data = pd.read_csv(input_file, header = None)
```

- Python supports various file formats for reading and writing data.
- The `open()` function is used for basic file I/O. Additionally, libraries such as `csv` (for CSV files), `json` (for JSON files), `pickle` (for serialization), and others offer specialized functions for working with specific file formats.

# Data Cleaning

- Data comes from third-party sources is generally noisy.
- Various steps are required to clean the collected data.
- It is necessary to remove duplicates, outliers.
- Sampling data when it is imbalanced.



# Data Cleaning

```
import pandas as pd

# Create a DataFrame from a CSV file
df = pd.read_csv('data.csv')

# Perform data manipulation tasks
df_cleaned = df.dropna()
df_filtered = df[df['column'] > 10]
```

- Python provides libraries for cleaning and transforming data.
- Pandas is used a powerful data manipulation tool, including functions for handling missing data, filtering, merging, grouping, and reshaping data.

# Data Cleaning

- Identify missing values: data is usually noisy with a lot of blank fields.
- Options for handling missing data include removing rows or columns, filling missing values with a specific value, or interpolating missing values.
- Identify and remove duplicate rows: many rows are duplicate, and need to be removed to keep just one of them.

# Data Cleaning and Transformation

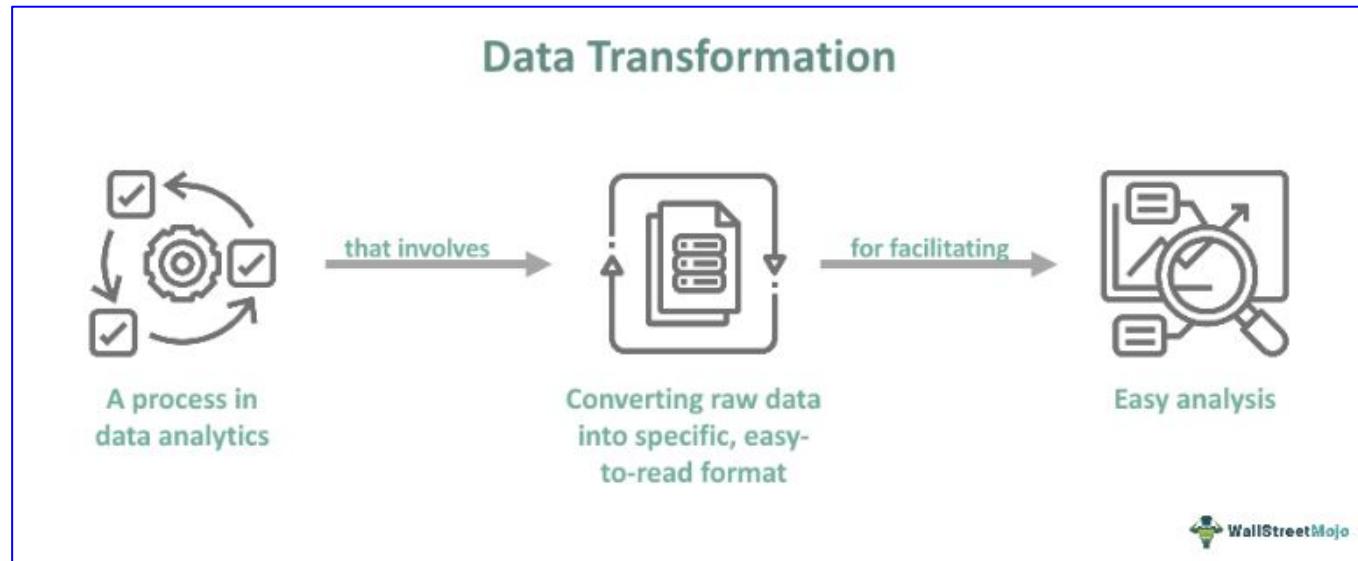


Image source: <https://www.wallstreetmojo.com/data-transformation/>

- Data cleaning and transformation are crucial steps in the data analysis process to ensure that the data is in a usable and meaningful format.
- The goal is to identify and handle missing or inconsistent data, perform necessary transformations, and prepare the data for analysis.

# Data Transformation

- Rename columns using the `rename()` method.
- Create new columns based on existing ones.
- Convert data types.
- Identify and handle outliers using statistical methods.
- Options include removing outliers or transforming the data.

# Data Transformation (2)

- Encode categorical variables using techniques like one-hot encoding.
- Group data using **groupby()** and perform aggregations.
- Convert date columns to the datetime format.
- Extract features like day, month, or year.

# Data Visualization

- Data visualization is a crucial aspect of data analysis and interpretation. It involves creating graphical representations of data to uncover patterns, trends, and insights.
- Effective data visualization enhances the understanding of complex datasets and facilitates better decision-making.
- In Python, the **Matplotlib** and **Seaborn** libraries are commonly used for data visualization.

# Data Visualization

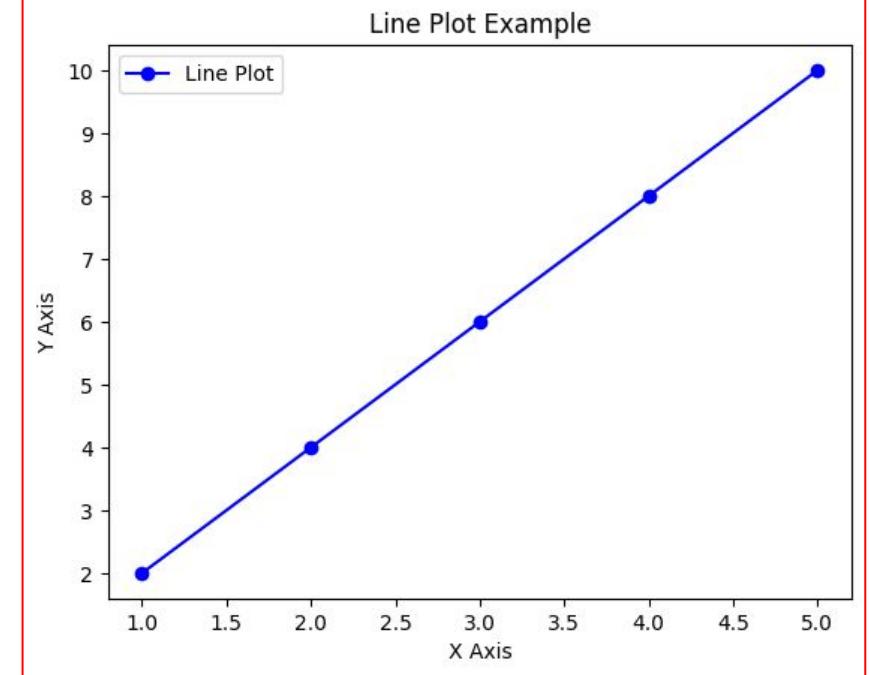
- Line Plots: Show the trend of a numerical variable over time.
- Bar Charts: Compare the values of different categories.
- Histograms: Display the distribution of a continuous variable.
- Scatter Plots: Explore relationships between two numerical variables.
- Pie Charts: Illustrate the proportion of each category in a whole.

# Line Plots

```
import matplotlib.pyplot as plt

# Sample data
x_values = [1, 2, 3, 4, 5]
y_values = [2, 4, 6, 8, 10]

# Line plot
plt.plot(x_values, y_values, marker= 'o', linestyle= '-',
color='blue', label='Line Plot')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('Line Plot Example')
plt.legend()
plt.show()
```



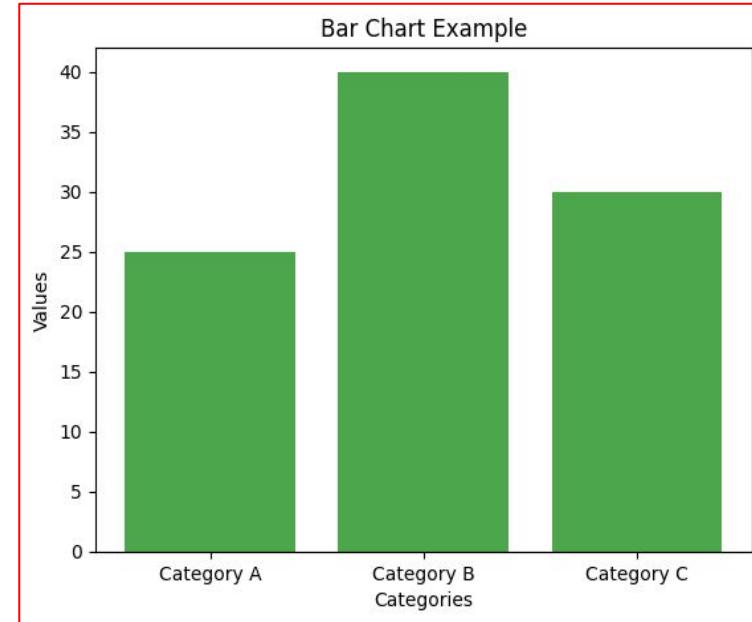
- Show the trend of a numerical variable over time

# Bar Chart

```
import matplotlib.pyplot as plt

# Sample data
categories = [ 'Category A' , 'Category B' , 'Category C' ]
values = [ 25 , 40 , 30 ]

# Bar chart
plt.bar(categories, values, color= 'green' , alpha=0.7)
plt.xlabel( 'Categories' )
plt.ylabel( 'Values' )
plt.title( 'Bar Chart Example' )
plt.show()
```



## Bar Charts: Compare the values of different categories

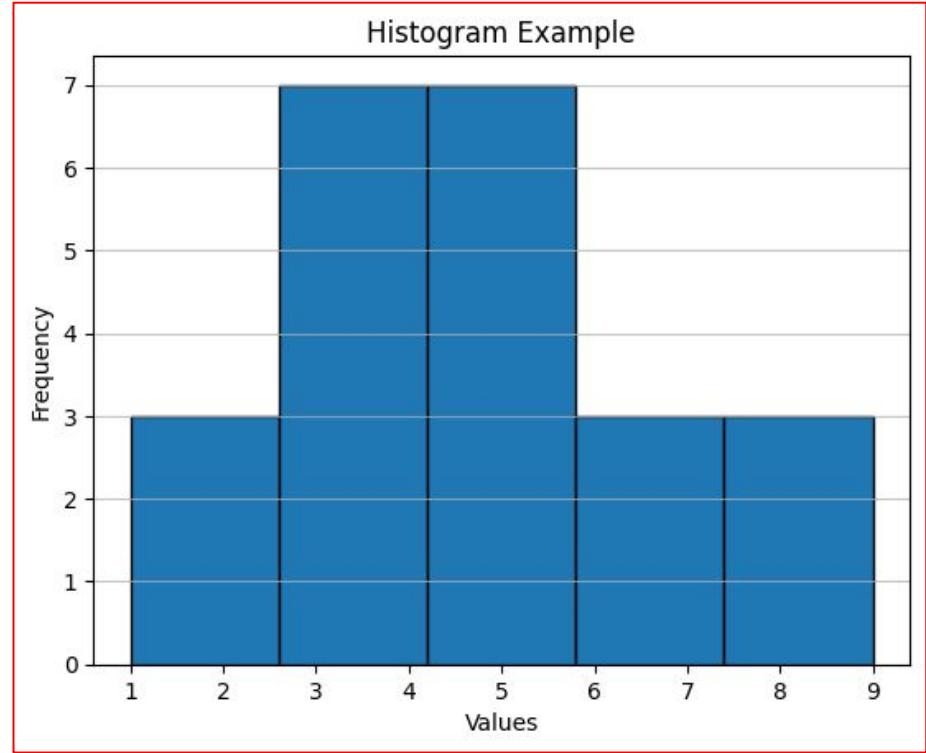
# Histograms

```
import matplotlib.pyplot as plt

# Sample data
data = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5,
5, 5, 6, 6, 6, 8, 8, 9]

# Create a histogram
plt.hist(data, bins=5, edgecolor='black') # 'bins'
determine the number of bins
plt.title('Histogram Example') # 'bins'
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)

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



- Visualizes the underlying frequency distribution of a set of continuous or discrete data
- In a histogram, data is divided into a set of bins, and the number of data points falling into each bin is represented by the height of a bar

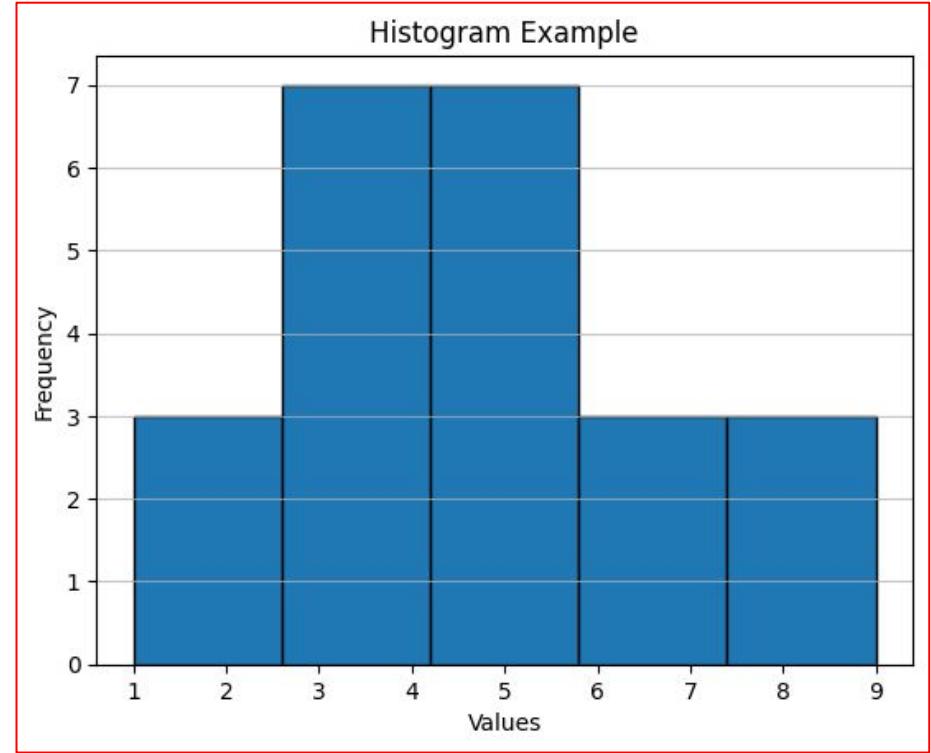
# Histograms

```
import matplotlib.pyplot as plt

# Sample data
data = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 8, 8, 9]

# Create a histogram
plt.hist(data, bins=5, edgecolor='black') # 'bins'
determine the number of bins
plt.title('Histogram Example')
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)

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



- Visualizes the underlying frequency distribution of a set of continuous or discrete data
- In a histogram, data is divided into a set of bins, and the number of data points falling into each bin is represented by the height of a bar

# — randn() function

```
import numpy as np  
np.random.randn(d0, d1, ..., dn)
```

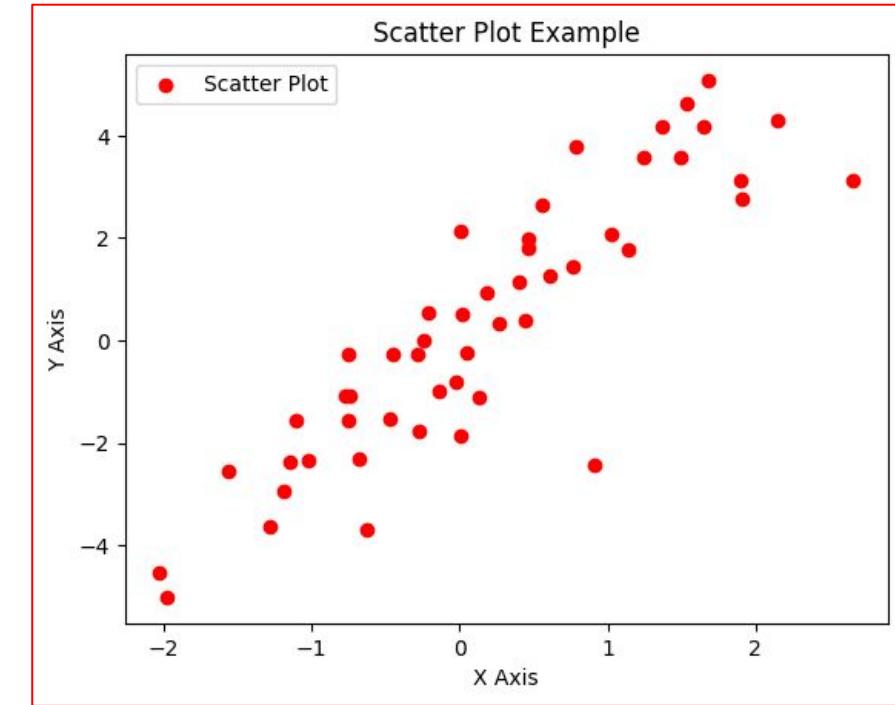
- Generates random numbers from a standard normal distribution with mean 0 and standard deviation 1, not from the interval [0, 1).
- Parameters: d0, d1, ..., dn: Optional. These are the dimensions of the returned array. If no argument is given, a single random value is generated.
- Return Value: Returns an array of random numbers with the specified dimensions, sampled from the standard normal distribution (mean = 0, standard deviation = 1).

# Scatter Plot

```
import matplotlib.pyplot as plt
import numpy as np

# Sample data
x_values = np.random.randn( 50 )
y_values = 2 * x_values + np.random.randn( 50 )

# Scatter plot
plt.scatter(x_values, y_values, color= 'red', marker= 'o',
label='Scatter Plot')
plt.xlabel( 'X Axis' )
plt.ylabel( 'Y Axis' )
plt.title( 'Scatter Plot Example' )
plt.legend()
plt.show()
```



- Explore relationships between two numerical variables.

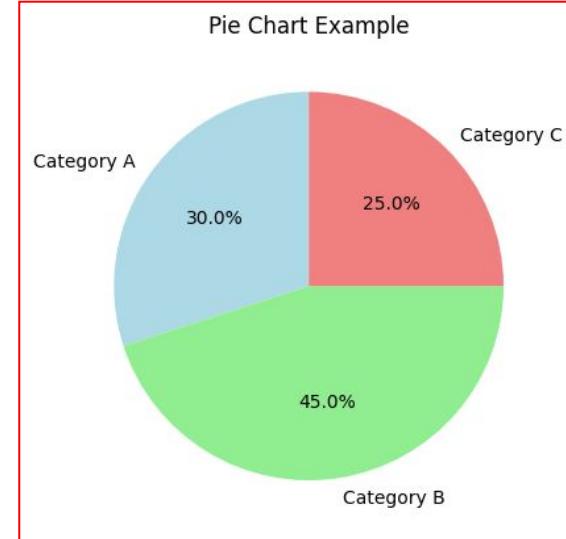


# Scatter Plot

```
import matplotlib.pyplot as plt

# Sample data
labels = ['Category A', 'Category B', 'Category C']
sizes = [30, 45, 25]

# Pie chart
plt.pie(sizes, labels=labels, autopct='%.1f%%', startangle=90,
colors=['lightblue', 'lightgreen', 'lightcoral'])
plt.title('Pie Chart Example')
plt.show()
```



- Illustrate the proportion of each category in a whole.

# Example of Data Storytelling

# Example 1

```
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
# Generate fictional dataset
data = {'Month': ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov',
'Dec'],
'Revenue': [200000, 220000, 240000, 210000, 230000, 250000, 260000, 270000, 240000, 230000,
220000, 250000]}
df = pd.DataFrame(data)
# Analyze the data
average_revenue = df['Revenue'].mean()
max_revenue_month = df.loc[df['Revenue'].idxmax(), 'Month']
```

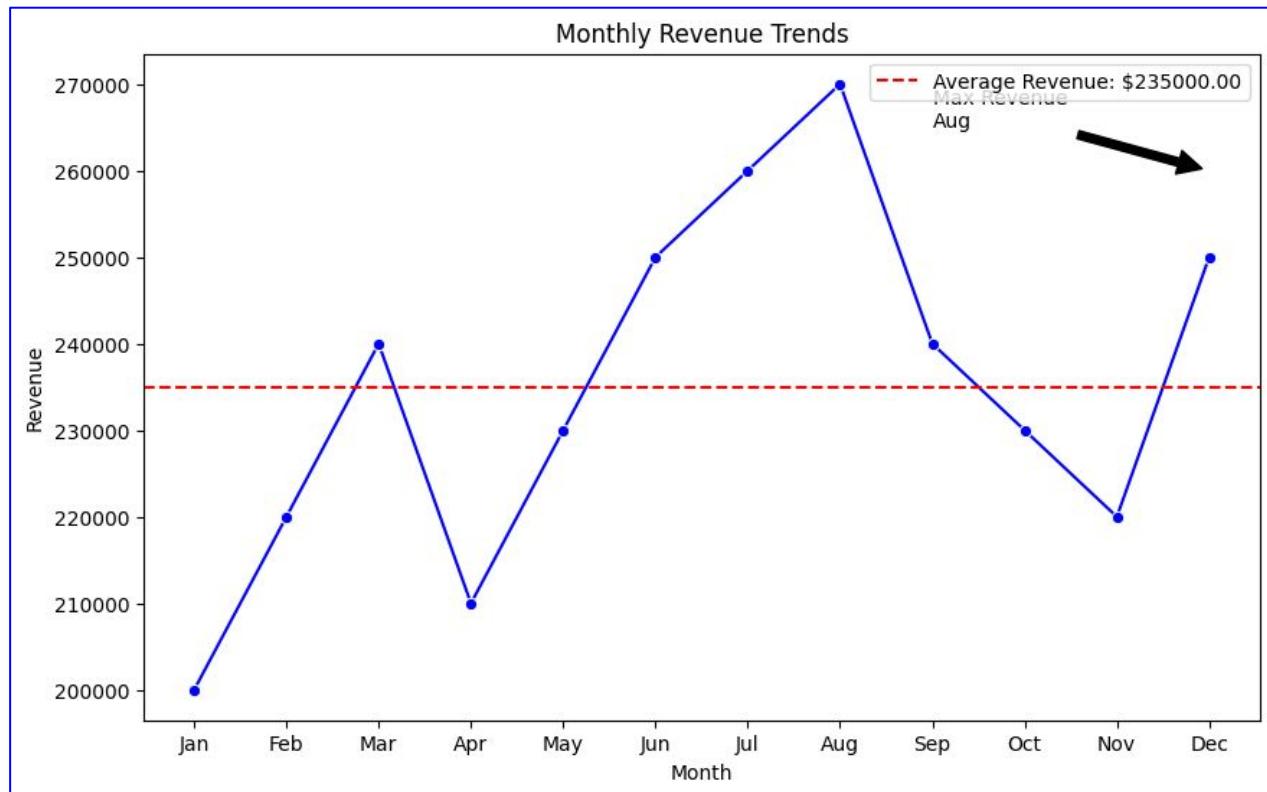


# Example 1

```
# Visualize the data
plt.figure(figsize=(10, 6))
sns.lineplot(x='Month', y='Revenue', data=df, marker='o', color='blue')
plt.title('Monthly Revenue Trends')
plt.xlabel('Month')
plt.ylabel('Revenue')
plt.axhline(average_revenue, color='red', linestyle='--', label=f'Average Revenue:
${average_revenue:.2f}')
plt.annotate(f'Max Revenue\n{max_revenue_month}', xy=(11, 260000), xytext=(8, 265000),
            arrowprops=dict(facecolor='black', shrink=0.05))
plt.legend()
plt.show()

# Tell the story
print(f"The average monthly revenue is ${average_revenue:.2f}.")
print(f"The month with the highest revenue is {max_revenue_month}.")
```

# Example 1



- The average monthly revenue is \$235,000.00
- The month with the highest revenue is Aug.

## Example 2

- A dataset containing information about the sales performance of a company over several quarters, we want to create a data story to understand the trends and insights from the data
- The dataset (**sales\_data.csv**) contains the following columns:
  - Quarter: The quarter in which the sales were recorded.
  - Revenue: The revenue generated in that quarter.
  - Expenses: The expenses incurred in that quarter.
  - Profit: The profit calculated as Revenue - Expenses

# Example 2: Code (Part 1)

```
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
# Load the dataset
from google.colab import drive
drive.mount('/content/drive/', force_remount=False)
DATA_PATH = '/content/drive/My Drive/Colab Notebooks/Data Science/'
from os.path import join
data_file = join(DATA_PATH,'sales_data.csv')
sales_data = pd.read_csv(data_file)
# Explore the dataset
print(sales_data.head())
# Data Exploration and Visualization
# Plotting Revenue, Expenses, and Profit over Quarters
plt.figure(figsize=(12, 6))
sns.lineplot(x='Quarter', y='Revenue', data=sales_data, label='Revenue', marker='o')
sns.lineplot(x='Quarter', y='Expenses', data=sales_data, label='Expenses', marker='o')
sns.lineplot(x='Quarter', y='Profit', data=sales_data, label='Profit', marker='o')
plt.title('Sales Performance Over Quarters')
plt.xlabel('Quarter')
plt.ylabel('Amount ($)')
plt.legend()
plt.grid(True)
plt.show()
```

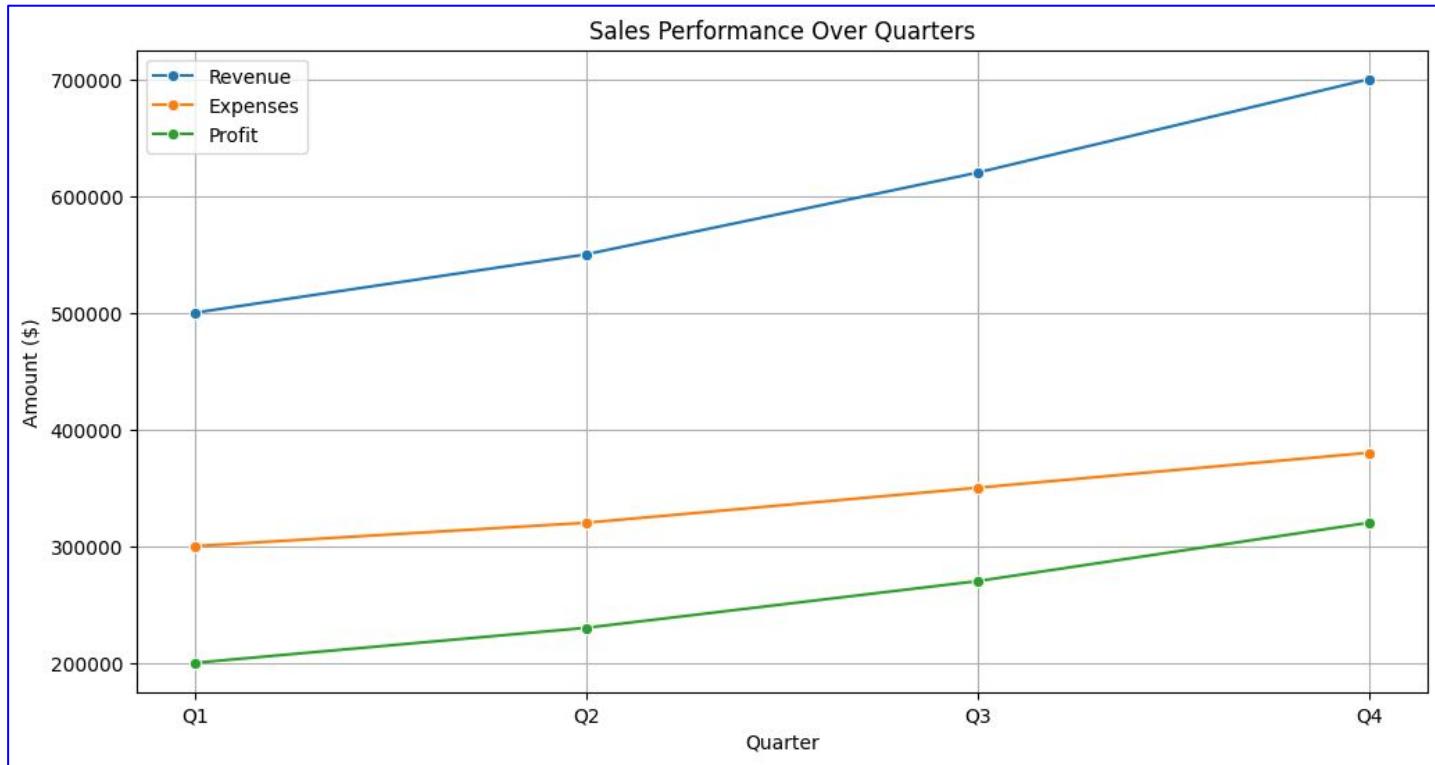
# Example 2: Code (Part 2)

```
# Analyzing Profit Margin
sales_data['Profit_Margin'] = (sales_data['Profit'] /
sales_data['Revenue']) * 100

# Plotting Profit Margin
plt.figure(figsize=(10, 5))
sns.barplot(x='Quarter', y='Profit_Margin', data=sales_data,
palette='viridis')
plt.title('Profit Margin Over Quarters')
plt.xlabel('Quarter')
plt.ylabel('Profit Margin (%)')
plt.ylim(0, 30)
plt.show()

# Summary and Conclusion
average_profit_margin = sales_data[Profit_Margin].mean()
print(f"Average Profit Margin: {average_profit_margin:.2f}%)")
```

# Example 2: Insight



- Revenue, Expenses, and Profit show an increasing trend over the quarters
- The company experienced the highest profit margin in Q3
- The average profit margin across quarters is  $\{average\_profit\_margin:.2f\}\%$

## — Example 2: Recommendations

- Explore the factors contributing to the high profit margin in Q3 for potential strategies.
- Evaluate the effectiveness of cost-cutting measures in improving overall profitability.

## Example 3

- Exploring and visualizing sales data to understand trends over time, such as monthly sales growth.
- Observing steady sales growth, with some months experiencing higher spikes (e.g., from February to March). The growth stabilizes toward the end of the year.
- Seasonal trends or external factors may drive sales fluctuations. This insight could guide sales strategy and inventory planning.

# Example 3: Code (Part 1)

## Load and Explore the Data

```
import pandas as pd
import numpy as np

# Sample sales data
data = {
    'Date': pd.date_range(start='2023-01-01', periods=12, freq='M'),
    'Sales': [200, 250, 300, 310, 400, 380, 420, 450, 470, 500, 520, 530]
}
df = pd.DataFrame(data)
df['Month'] = df['Date'].dt.strftime('%b')
df['Month_Num'] = df['Date'].dt.month
print(df.head())
```

## Calculate Monthly Sales Growth

```
df['Sales_Growth'] = df['Sales'].pct_change() * 100 # Percentage growth
df['Sales_Growth'].fillna(0, inplace=True) # Fill NaN for the first month with 0
print(df)
```

# Example 3: Code (Part 2)

Visualize Monthly Sales and Growth

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

# Line plot for sales and bar plot for sales growth
fig, ax1 = plt.subplots(figsize=(10, 6))

# Plot sales trend
sns.lineplot(data=df, x='Month_Num', y='Sales', marker='o', color='b', ax=ax1)
ax1.set_ylabel("Sales ($)", color='b')
ax1.set_xlabel("Month")
ax1.set_xticks(df['Month_Num'])
ax1.set_xticklabels(df['Month'], rotation=45)

# Overlay sales growth with a secondary y-axis
ax2 = ax1.twinx()
sns.barplot(data=df, x='Month_Num', y='Sales_Growth', alpha=0.6, color='orange', ax=ax2)
ax2.set_ylabel("Sales Growth (%)", color='orange')

plt.title("Monthly Sales and Sales Growth")
plt.show()
```



# Introduction to Anaconda Programming for Data Science 2024/25

Giordano d'Aloisio

[giordano.daloisio@univaq.it](mailto:giordano.daloisio@univaq.it)

# Anaconda Download

- Go to this link and download Anaconda
- Follow all the default selections of the installer

<https://www.anaconda.com/download/success>

# How Python Works

- Python is an interpreted language
- To run a Python code, we only need the Python file and the Python interpreter

```
● ○ ●  
import sys  
  
a = sys[0]  
b = sys[1]  
  
print(f"The sum of {a} and {b} is {a+b}")
```

Python file  
(sum.py)



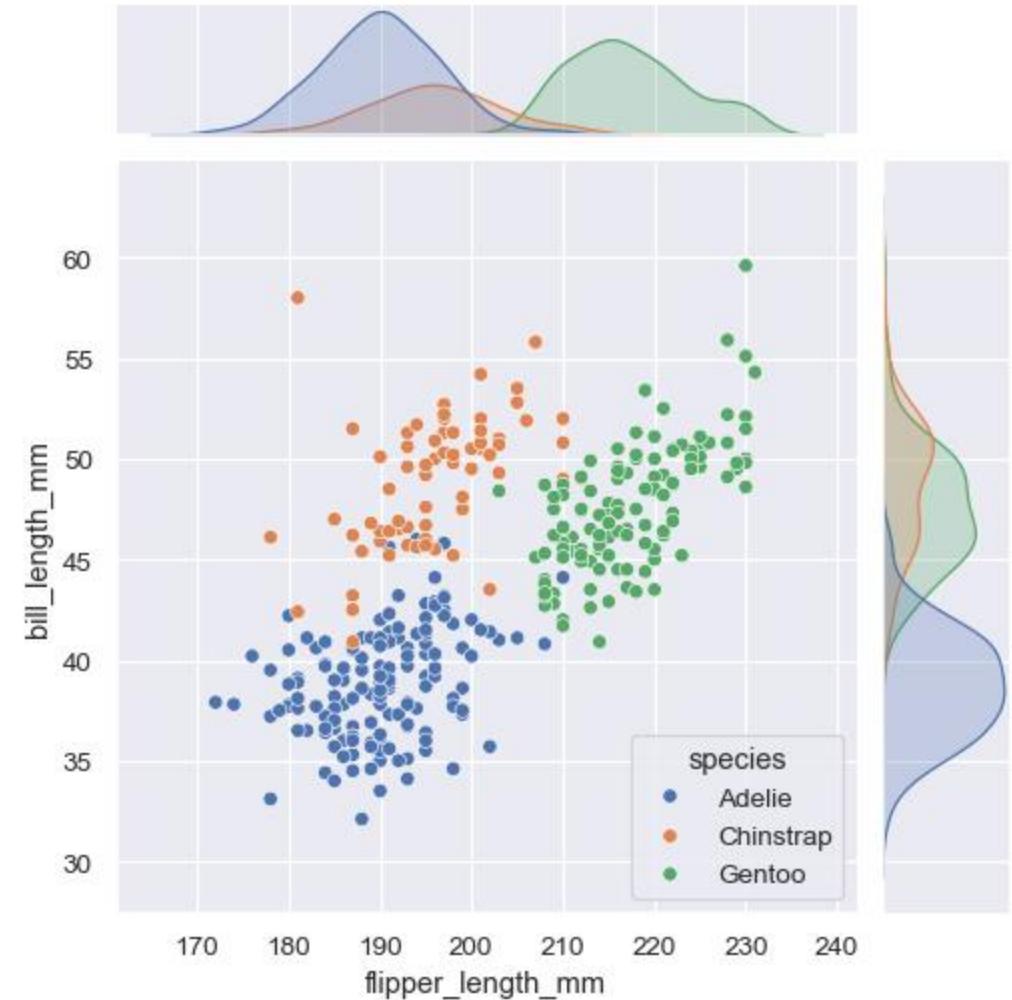
Python  
interpreter

```
● ○ ●  
~ $ python sum.py 2 3  
The sum of 2 and 3 is 5
```

Solution  
(in the shell)

# Why Python alone is not enough?

- In Data Science (but not only) Python alone is not enough
- Additional libraries are usually required
- For instance, to read and process a file or to make complex plots



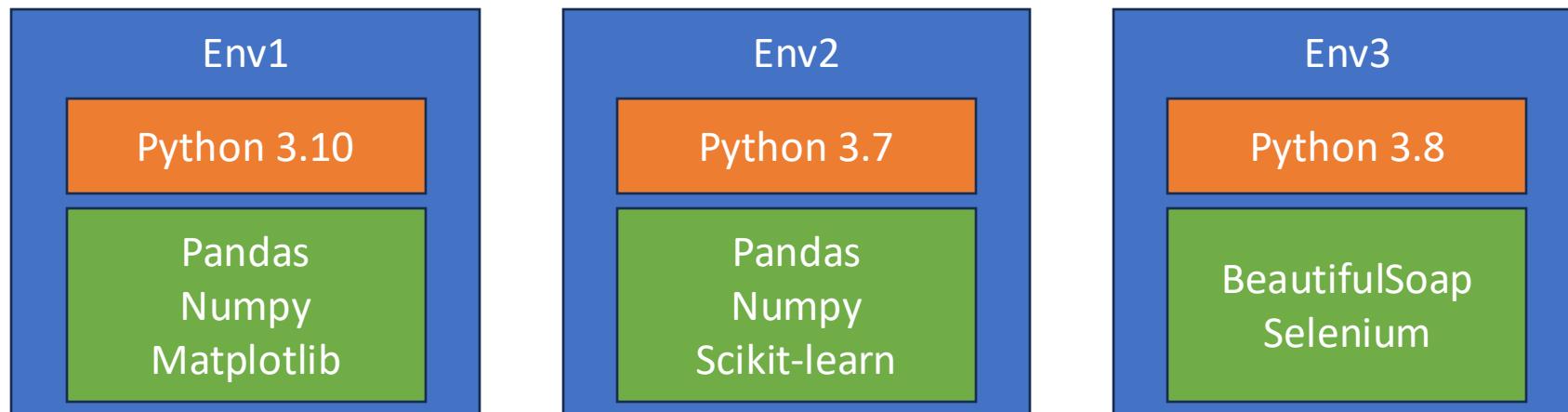
# Why Anaconda

- Many Python libraries are required in Data Science (e.g., Numpy, Pandas, Matplotlib, Scikit-learn, ...)
- However, different libraries can require different versions of the same dependency
- A wrong management of the libraries (*spaghetti dependencies*) can destroy a Python environment



# Why Anaconda

- **Anaconda** is a tool specifically designed to address the *spaghetti dependencies* problem in Data Science
- Allows to:
  - Install different versions of Python
  - Install Python libraries
  - Create several isolated virtual environments with a specific Python version and set of libraries



# Conda

- Conda is an open-source **package** and **environment** management system
- Conda allows to install, run, and update packages and their dependencies.
- It also allows to create, save, load, and switch between environments.
- Can be used through the **conda** command in the command line or via the *Anaconda Navigator* graphical interface

```
[bash-3.2$ conda install pandas
Collecting package metadata (current_repodata.json) : done
Solving environment: done

## Package Plan ##

environment location: /opt/anaconda3

added / updated specs:
  - pandas

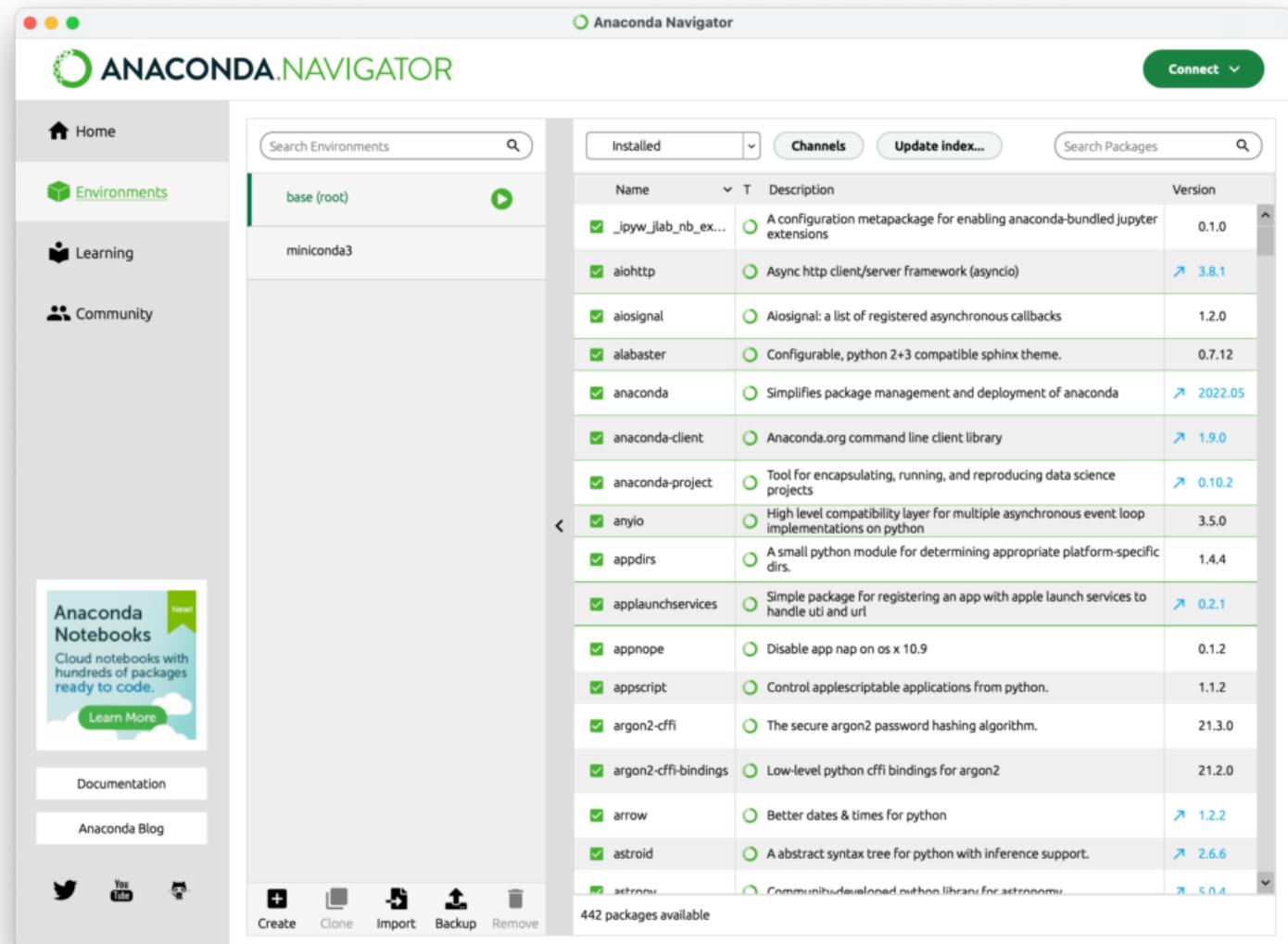
The following packages will be downloaded:

package          | build
-----
pandas- 1.        | py39he9d5cce_1    9.4 MB
-----
                                         Total:      9.4 MB

The following packages will be UPDATED:
```

# Anaconda Navigator

- Graphical interface for conda
- Allows to create, delete and manage environments
- Allows to install libraries inside an environment



# Dictionaries and Sets

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

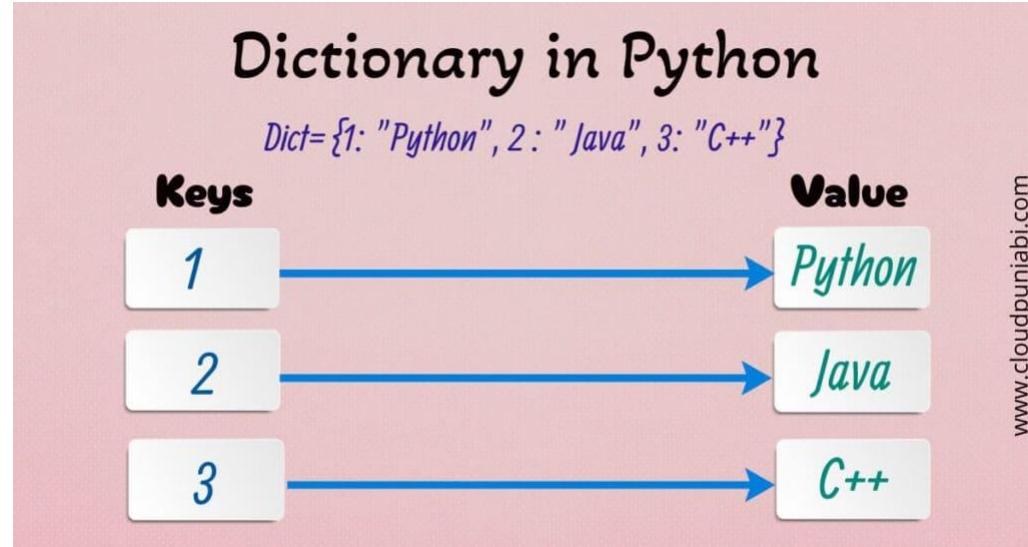
Università degli Studi dell'Aquila

# Dictionaries

# Dictionaries

- Dictionary is a built-in data structure used to store collections of key-value pairs.
- Dictionaries are also known as associative arrays or hash maps in other programming languages (e.g., Java).
- They are highly versatile and are widely used for organizing and retrieving data efficiently.
- Dictionaries are defined by enclosing a comma-separated list of key-value pairs in curly braces '{}':

# Dictionaries



- Dictionaries consist of key-value pairs, where each key is associated with a corresponding value.
- Keys are unique within a dictionary, meaning that no two keys can have the same name. Nevertheless, values can be duplicates.

# Syntax

```
# Creating a dictionary
my_dict = { 'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}

# Accessing values using keys
print(my_dict['key1']) # Output: 'value1'
```

- Unordered: Items in a dictionary are not stored in any specific order. As of Python 3.7+, the insertion order of items is maintained, but we should not rely on this behavior for ordering.
- Dictionaries are mutable, i.e., we can modify their content by adding, removing, or updating key-value pairs.



# Operations

## ■ Accessing values:

```
# Accessing values using keys
value = my_dict['key2']
print(value) # Output: 'value2'
```

## ■ Adding a Key-Value Pair:

```
# Adding a new key-value pair
my_dict['new_key'] = 'new_value'
print(my_dict)
# Output: {'key1': 'value1', 'key2': 'value2', 'key3': 'value3', 'new_key':
'new_value'}
```

# Operations (2)

## ■ Update a value:

```
# Updating the value of an existing key
my_dict['key1'] = 'updated_value'
print(my_dict)
# Output: {'key1': 'updated_value', 'key2': 'value2', 'key3': 'value3', 'new_key': 'new_value'}
```

## ■ Removing a Key-Value Pair:

```
# Removing a key-value pair
del my_dict['key3']
print(my_dict)
# Output: {'key1': 'updated_value', 'key2': 'value2', 'new_key': 'new_value'}
```

# Operations (3)

## ■ Update a value:

```
# Checking if a key exists
if 'key1' in my_dict:
    print("Key 'key1' exists!")
```

## ■ Getting a List of Keys or Values:

```
# Getting a list of keys
keys = my_dict.keys()
print(keys) # Output: dict_keys(['key1', 'key2', 'new_key'])

# Getting a list of values
values = my_dict.values()
print(values) # Output: dict_values(['updated_value', 'value2', 'new_value'])
```

# Operations (4)

## ■ Traversing over the values:

```
# Iterating over keys and values
for key, value in my_dict.items():
    print(f"Key: {key}, Value: {value}")
```

## ■ Merging two dictionaries:

```
dict1 = {'key1': 'value1', 'key2': 'value2'}
dict2 = {'key2': 'updated_value2', 'key3': 'value3'}

# Concatenate dictionaries using the update method
dict1.update(dict2)

print("Merged Dictionary:", dict1)
```

# Unpacking

## ■ Combining two dictionaries (Python 3.5 or newer):

```
dict1 = {'key1': 'value1', 'key2': 'value2'}  
dict2 = {'key2': 'updated_value2', 'key3': 'value3'}  
  
# Concatenate dictionaries using dictionary unpacking  
merged_dict = {**dict1, **dict2}  
  
print("Merged Dictionary:", merged_dict)
```

## ■ Combining two dictionaries (Python 3.9 or newer):

```
dict1 = {'key1': 'value1', 'key2': 'value2'}  
dict2 = {'key2': 'updated_value2', 'key3': 'value3'}  
  
# Concatenate dictionaries using the | operator  
merged_dict = dict1 | dict2  
  
print("Merged Dictionary:", merged_dict)
```

# Sets

- Built-in data type used to store a collection of unique and unordered elements.
- Sets are defined using curly braces {} or the **set()** constructor.
- Uniqueness: Sets do not allow duplicate elements. If you try to add the same element multiple times, it will only be stored once in the set.

# Operations

## ■ Create a new set

```
my_set = {1, 2, 3, 4, 5}
```

## ■ Add a new element

```
my_set.add(6)  
# my_set: {1, 2, 3, 4, 5, 6}
```

## ■ Remove an element from set

```
my_set.remove(3)  
# my_set: {1, 2, 4, 5, 6}
```

## ■ Checking

```
print(2 in my_set) # Output: True  
print(3 in my_set) # Output: False
```

# — Union, Intersection, and Difference

```
# Set operations
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
union_set = set1 | set2 # Union of two sets
intersection_set = set1 & set2 # Intersection of two sets
difference_set = set1 - set2 # Difference between sets

# Checking for membership
if 5 in my_set:
    print("5 is in the set")

# Length of a set
set_length = len(my_set)

# Printing the sets
print(my_set) # Output: {1, 2, 4, 5, 6}
```

# Example

```
# Creating a set
my_set = {1, 2, 3, 4, 5}
# Adding an element
my_set.add(6)
# Removing an element
my_set.remove(3)
# Set operations
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
union_set = set1 | set2 # Union of two sets
intersection_set = set1 & set2 # Intersection of two sets
difference_set = set1 - set2 # Difference between sets

# Checking for membership
if 5 in my_set:
    print("5 is in the set")

# Length of a set
set_length = len(my_set)

# Printing the sets
print(my_set) # Output: {1, 2, 4, 5, 6}
```

# Sets of strings

```
# Define two sets of strings
fruits_set1 = {"apple", "banana", "cherry", "orange"}
fruits_set2 = {"banana", "kiwi", "mango", "cherry"}

# Union of sets
union_fruits = fruits_set1 | fruits_set2
print("Union of sets:", union_fruits)

# Intersection of sets
intersection_fruits = fruits_set1 & fruits_set2
print("Intersection of sets:", intersection_fruits)

# Difference between sets
difference_fruits = fruits_set1 - fruits_set2
print("Difference between sets (fruits_set1 - fruits_set2):", difference_fruits)

# Check if a specific fruit is in either set
if "apple" in fruits_set1:
    print("Apple is in fruits_set1")
if "mango" in fruits_set2:
    print("Mango is in fruits_set2")
```

# — Intersections of more than two sets

```
# Define three sets of integers
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4, 5, 6, 7}
set3 = {5, 6, 7, 8, 9}

# Find the intersection of the three sets
intersection_result = set1 & set2 & set3

# Alternatively, you can use the .intersection() method
# intersection_result = set1.intersection(set2, set3)

print("Intersection of the three sets:", intersection_result)
```

- It is possible to compute intersection for more than two sets.
- Results: elements that are common in all the considered sets.

# — Immutable and Hashable

- Immutable: Sets that cannot be changed (They are fixed).
- A hash is a function that takes in an input (or ‘message’) and returns a fixed-size string of characters, which is typically a hexadecimal number.
- The output, commonly referred to as the hash value or hash code, is unique to the specific input.
- “hashable” means that an object has a hash value associated with it, and it can be used as a key in a dictionary or an element in a set.

# Frozenset

- It is an immutable, hashable set, similar to a regular set, but with the key difference that it is not possible to change its elements once it is created
- Operations that modify sets, such as `add()`, `remove()`, and `pop()`, are not applicable to frozensets
- This immutability makes frozensets suitable for situations where you need a fixed set of elements that should not change

# Frozenset

- Similar to sets, frozensets contain only unique elements. If someone attempts to create a frozenset with duplicate elements, then the duplicates will be automatically removed
- Hashability: Unlike regular sets, frozensets are hashable, i.e., we can use frozensets as elements of other sets or as keys in dictionaries because they have a stable hash value
- Creation: We can create a frozenset by using the **frozenset()** constructor and passing an iterable (e.g., a list or another set) as an argument

# Frozenset

```
# Create a regular set
my_set = {1, 2, 3, 4, 5}

# Create a frozenset
my_frozenset = frozenset([1, 2, 3, 4, 5])

# Attempt to modify a frozenset (this will raise an error)
# my_frozenset.add(6) # Raises an error

# Using frozensets as keys in a dictionary
employee_info = {
    frozenset({"John", "Doe"}) : 30,
    frozenset({"Alice", "Smith"}) : 25,
}

# Printing the frozensets
print("Regular set:", my_set)
print("Frozenset:", my_frozenset)
```

# — Hashable

```
# Creating a frozenset
frozen_set = frozenset([1, 2, 3])

# Checking hashability
print(hash(frozen_set)) # You can hash a frozenset

frozen_set_2 = frozenset([1, 2, 3])

print(hash(frozen_set_2))

# Attempting to hash a regular set (which is mutable)
regular_set = {1, 2, 3}
# print(hash(regular_set)) # This would raise an error because sets
are not hashable
```

- Hashability: To be used as a key in a dictionary or an element in a set, an object needs to be hashable
- Hashable objects have a fixed hash value that doesn't change during their lifetime

# Examples

# — Student Records

- Create a dictionary representing a student, and then perform various operations like accessing values, modifying values, adding new key-value pairs, and iterating over the dictionary
- Additionally, we demonstrate dictionary comprehension and working with nested dictionaries

# — Student Records

```
# Creating a Dictionary
student = {
    'name': 'Alice',
    'age': 20,
    'major': 'Computer Science',
    'grades': {'math': 90, 'english': 85, 'history': 92}
}

# Accessing Values
print("Name:", student['name'])
print("Age:", student['age'])
print("Major:", student['major'])
print("Math Grade:", student['grades']['math'])

# Modifying Values
student['age'] = 21
student['grades']['english'] = 88

# Adding New Key-Value Pairs
student['gender'] = 'Female'
student['grades']['physics'] = 87
```

# — Student Records

```
# Removing Key-Value Pairs
del student['grades']['history']

# Iterating Over Keys
print("\nKeys:")
for key in student:
    print(key)

# Iterating Over Values
print("\nValues:")
for value in student.values():
    print(value)

# Iterating Over Key-Value Pairs
print("\nKey-Value Pairs:")
for key, value in student.items():
    print(f"{key}: {value}")

# Check if Key Exists
print("\nCheck if 'gender' exists:", 'gender' in student)
```

# — Student Records

```
# Dictionary Comprehension
squared_numbers = {x: x**2 for x in range(5)}
print("\nDictionary Comprehension:")
print(squared_numbers)

# Nested Dictionaries
classroom = {
    'student1': {'name': 'Bob', 'grade': 'A'},
    'student2': {'name': 'Charlie', 'grade': 'B'},
    'student3': {'name': 'David', 'grade': 'C'}
}

print("\nNested Dictionaries:")
for student_id, details in classroom.items():
    print(f"Student {student_id}: {details['name']} - Grade: {details['grade']}")
```

# Fruits

```
# Creating Sets
fruits_set = {'apple', 'banana', 'orange', 'kiwi'}
vegetables_set = {'carrot', 'spinach', 'broccoli', 'banana'} # Note:
'banana' is included in both sets

# Accessing Elements
print("Fruits Set:", fruits_set)
print("Vegetables Set:", vegetables_set)

# Adding Elements
fruits_set.add('grape')
vegetables_set.add('cucumber')

# Removing Elements
fruits_set.remove('banana') # Raises an error if the element is not
present
vegetables_set.discard('banana') # Removes the element if present,
otherwise does nothing
```

- Create sets of fruits and vegetables, perform various set operations like union, intersection, and difference, use set comprehension, and demonstrate other set-related functionalities

# Fruits

```
# Set Operations
union_set = fruits_set.union(vegetables_set)
intersection_set = fruits_set.intersection(vegetables_set)
difference_set = fruits_set.difference(vegetables_set)
symmetric_difference_set = fruits_set.symmetric_difference(vegetables_set)

# Set Comprehension
squared_numbers_set = {x**2 for x in range(5)}

# Checking Membership
print("'kiwi' in fruits_set:", 'kiwi' in fruits_set)
print("'apple' not in vegetables_set:", 'apple' not in vegetables_set)

# Iterating Over Sets
print("\nIterating Over Fruits Set:")
for fruit in fruits_set:
    print(fruit)

# Clearing a Set
fruits_set.clear()

# Example: Finding Unique Elements in a List
numbers_list = [1, 2, 3, 4, 2, 3, 5, 6, 1, 7]
unique_numbers_set = set(numbers_list)

print("\nList with Duplicates:", numbers_list)
print("Set with Unique Elements:", unique_numbers_set)
```

# Sets

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# Sets

- Built-in data type used to store a collection of unique and unordered elements.
- Sets are defined using curly braces {} or the **set()** constructor.
- Uniqueness: Sets do not allow duplicate elements. If you try to add the same element multiple times, it will only be stored once in the set.

# Operations

## ■ Create a new set

```
my_set = {1, 2, 3, 4, 5}
```

## ■ Add a new element

```
my_set.add(6)  
# my_set: {1, 2, 3, 4, 5, 6}
```

## ■ Remove an element from set

```
my_set.remove(3)  
# my_set: {1, 2, 4, 5, 6}
```

## ■ Checking

```
print(2 in my_set) # Output: True  
print(3 in my_set) # Output: False
```

# — Union, Intersection, and Difference

```
# Set operations
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
union_set = set1 | set2 # Union of two sets
intersection_set = set1 & set2 # Intersection of two sets
difference_set = set1 - set2 # Difference between sets

# Checking for membership
if 5 in my_set:
    print("5 is in the set")

# Length of a set
set_length = len(my_set)

# Printing the sets
print(my_set) # Output: {1, 2, 4, 5, 6}
```

# Example

```
# Creating a set
my_set = {1, 2, 3, 4, 5}
# Adding an element
my_set.add(6)
# Removing an element
my_set.remove(3)
# Set operations
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
union_set = set1 | set2 # Union of two sets
intersection_set = set1 & set2 # Intersection of two sets
difference_set = set1 - set2 # Difference between sets

# Checking for membership
if 5 in my_set:
    print("5 is in the set")

# Length of a set
set_length = len(my_set)

# Printing the sets
print(my_set) # Output: {1, 2, 4, 5, 6}
```

# Sets of strings

```
# Define two sets of strings
fruits_set1 = {"apple", "banana", "cherry", "orange"}
fruits_set2 = {"banana", "kiwi", "mango", "cherry"}

# Union of sets
union_fruits = fruits_set1 | fruits_set2
print("Union of sets:", union_fruits)

# Intersection of sets
intersection_fruits = fruits_set1 & fruits_set2
print("Intersection of sets:", intersection_fruits)

# Difference between sets
difference_fruits = fruits_set1 - fruits_set2
print("Difference between sets (fruits_set1 - fruits_set2):", difference_fruits)

# Check if a specific fruit is in either set
if "apple" in fruits_set1:
    print("Apple is in fruits_set1")
if "mango" in fruits_set2:
    print("Mango is in fruits_set2")
```

# — Intersections of more than two sets

```
# Define three sets of integers
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4, 5, 6, 7}
set3 = {5, 6, 7, 8, 9}

# Find the intersection of the three sets
intersection_result = set1 & set2 & set3

# Alternatively, you can use the .intersection() method
# intersection_result = set1.intersection(set2, set3)

print("Intersection of the three sets:", intersection_result)
```

- It is possible to compute intersection for more than two sets.
- Results: elements that are common in all the considered sets.

# — Immutable and Hashable

- Immutable: Sets that cannot be changed (They are fixed).
- A hash is a function that takes in an input (or ‘message’) and returns a fixed-size string of characters, which is typically a hexadecimal number.
- The output, commonly referred to as the hash value or hash code, is unique to the specific input.
- “hashable” means that an object has a hash value associated with it, and it can be used as a key in a dictionary or an element in a set.

# Frozenset

- It is an immutable, hashable set, similar to a regular set, but with the key difference that it is not possible to change its elements once it is created
- Operations that modify sets, such as `add()`, `remove()`, and `pop()`, are not applicable to frozensets
- This immutability makes frozensets suitable for situations where you need a fixed set of elements that should not change

# Frozenset

- Similar to sets, frozensets contain only unique elements. If someone attempts to create a frozenset with duplicate elements, then the duplicates will be automatically removed
- Hashability: Unlike regular sets, frozensets are hashable, i.e., we can use frozensets as elements of other sets or as keys in dictionaries because they have a stable hash value
- Creation: We can create a frozenset by using the **frozenset()** constructor and passing an iterable (e.g., a list or another set) as an argument

# Frozenset

```
# Create a regular set
my_set = {1, 2, 3, 4, 5}

# Create a frozenset
my_frozenset = frozenset([1, 2, 3, 4, 5])

# Attempt to modify a frozenset (this will raise an error)
# my_frozenset.add(6) # Raises an error

# Using frozensets as keys in a dictionary
employee_info = {
    frozenset({"John", "Doe"}) : 30,
    frozenset({"Alice", "Smith"}) : 25,
}

# Printing the frozensets
print("Regular set:", my_set)
print("Frozenset:", my_frozenset)
```

# — Hashable

```
# Creating a frozenset
frozen_set = frozenset([1, 2, 3])

# Checking hashability
print(hash(frozen_set)) # You can hash a frozenset

frozen_set_2 = frozenset([1, 2, 3])

print(hash(frozen_set_2))

# Attempting to hash a regular set (which is mutable)
regular_set = {1, 2, 3}
# print(hash(regular_set)) # This would raise an error because sets
are not hashable
```

- Hashability: To be used as a key in a dictionary or an element in a set, an object needs to be hashable.
- Hashable objects have a fixed hash value that doesn't change during their lifetime.

# Examples

# — Student Records

- Create a dictionary representing a student, and then perform various operations like accessing values, modifying values, adding new key-value pairs, and iterating over the dictionary.
- Additionally, we demonstrate dictionary comprehension and working with nested dictionaries.

# — Student Records

```
# Creating a Dictionary
student = {
    'name': 'Alice',
    'age': 20,
    'major': 'Computer Science',
    'grades': {'math': 90, 'english': 85, 'history': 92}
}

# Accessing Values
print("Name:", student['name'])
print("Age:", student['age'])
print("Major:", student['major'])
print("Math Grade:", student['grades']['math'])

# Modifying Values
student['age'] = 21
student['grades']['english'] = 88

# Adding New Key-Value Pairs
student['gender'] = 'Female'
student['grades']['physics'] = 87
```

# — Student Records

```
# Removing Key-Value Pairs
del student['grades']['history']

# Iterating Over Keys
print("\nKeys:")
for key in student:
    print(key)

# Iterating Over Values
print("\nValues:")
for value in student.values():
    print(value)

# Iterating Over Key-Value Pairs
print("\nKey-Value Pairs:")
for key, value in student.items():
    print(f'{key}: {value}')

# Check if Key Exists
print("\nCheck if 'gender' exists:", 'gender' in student)
```

# — Student Records

```
# Dictionary Comprehension
squared_numbers = {x: x**2 for x in range(5)}
print("\nDictionary Comprehension:")
print(squared_numbers)

# Nested Dictionaries
classroom = {
    'student1': {'name': 'Bob', 'grade': 'A'},
    'student2': {'name': 'Charlie', 'grade': 'B'},
    'student3': {'name': 'David', 'grade': 'C'}
}

print("\nNested Dictionaries:")
for student_id, details in classroom.items():
    print(f"Student {student_id}: {details['name']} - Grade: {details['grade']}")
```

# Fruits

```
# Creating Sets
fruits_set = {'apple', 'banana', 'orange', 'kiwi'}
vegetables_set = {'carrot', 'spinach', 'broccoli', 'banana'} # Note:
'banana' is included in both sets

# Accessing Elements
print("Fruits Set:", fruits_set)
print("Vegetables Set:", vegetables_set)

# Adding Elements
fruits_set.add('grape')
vegetables_set.add('cucumber')

# Removing Elements
fruits_set.remove('banana') # Raises an error if the element is not
present
vegetables_set.discard('banana') # Removes the element if present,
otherwise does nothing
```

- Create sets of fruits and vegetables, perform various set operations like union, intersection, and difference, use set comprehension, and demonstrate other set-related functionalities

# Fruits

```
# Set Operations
union_set = fruits_set.union(vegetables_set)
intersection_set = fruits_set.intersection(vegetables_set)
difference_set = fruits_set.difference(vegetables_set)
symmetric_difference_set = fruits_set.symmetric_difference(vegetables_set)

# Set Comprehension
squared_numbers_set = {x**2 for x in range(5)}

# Checking Membership
print("'kiwi' in fruits_set:", 'kiwi' in fruits_set)
print("'apple' not in vegetables_set:", 'apple' not in vegetables_set)

# Iterating Over Sets
print("\nIterating Over Fruits Set:")
for fruit in fruits_set:
    print(fruit)

# Clearing a Set
fruits_set.clear()

# Example: Finding Unique Elements in a List
numbers_list = [1, 2, 3, 4, 2, 3, 5, 6, 1, 7]
unique_numbers_set = set(numbers_list)

print("\nList with Duplicates:", numbers_list)
print("Set with Unique Elements:", unique_numbers_set)
```

# The Pandas library

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila



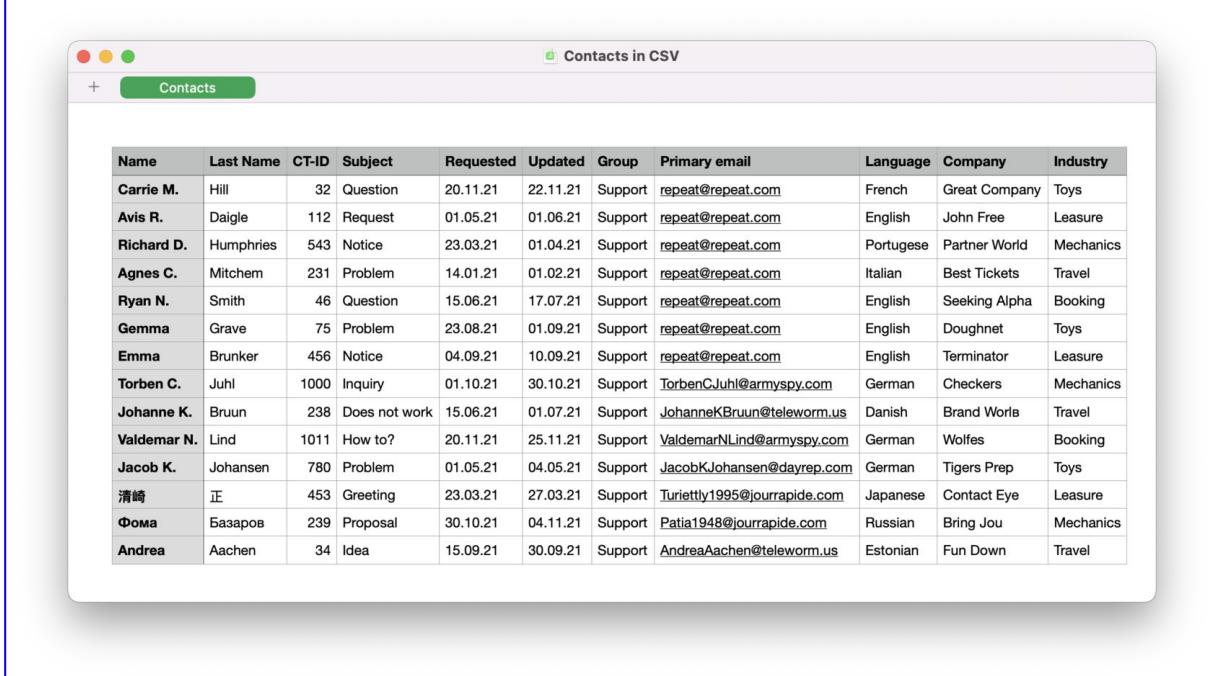
- **Python Data Analysis Library:** A popular and powerful Python library for data manipulation and analysis.
- It provides easy-to-use data structures and functions for working with structured data, such as spreadsheets or SQL tables.
- Pandas is widely used in data science, data analysis, and machine learning tasks.

# Functionalities



- **Data Indexing and Selection:** Pandas provides powerful methods for indexing and selecting data. We can use labels, conditions, or positions to extract specific portions of the data.
- **Data Cleaning:** Pandas simplifies the process of cleaning and preprocessing data. It offers functions for handling missing values, removing duplicates, and transforming data.
- **Data Aggregation and Grouping:** Pandas allows us to group data based on certain criteria and then perform calculations on each group. This is particularly useful for aggregating and summarizing data.

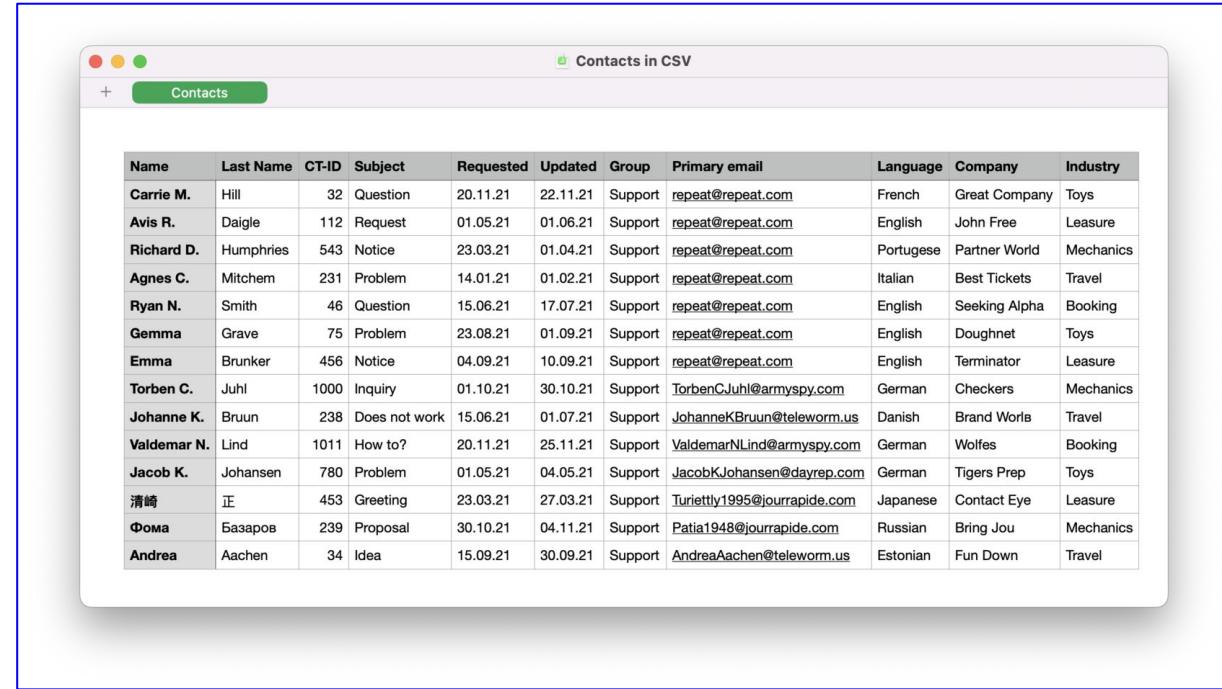
# CSV



Name	Last Name	CT-ID	Subject	Requested	Updated	Group	Primary email	Language	Company	Industry
Carrie M.	Hill	32	Question	20.11.21	22.11.21	Support	repeat@repeat.com	French	Great Company	Toys
Avis R.	Daigle	112	Request	01.05.21	01.06.21	Support	repeat@repeat.com	English	John Free	Leasure
Richard D.	Humphries	543	Notice	23.03.21	01.04.21	Support	repeat@repeat.com	Portugese	Partner World	Mechanics
Agnes C.	Mitchem	231	Problem	14.01.21	01.02.21	Support	repeat@repeat.com	Italian	Best Tickets	Travel
Ryan N.	Smith	46	Question	15.06.21	17.07.21	Support	repeat@repeat.com	English	Seeking Alpha	Booking
Gemma	Grave	75	Problem	23.08.21	01.09.21	Support	repeat@repeat.com	English	Doughnet	Toys
Emma	Brunker	456	Notice	04.09.21	10.09.21	Support	repeat@repeat.com	English	Terminator	Leasure
Torben C.	Juhl	1000	Inquiry	01.10.21	30.10.21	Support	TorbenCJuhl@armyspy.com	German	Checkers	Mechanics
Johanne K.	Bruun	238	Does not work	15.06.21	01.07.21	Support	JohanneKBruun@teleworm.us	Danish	Brand Worls	Travel
Valdemar N.	Lind	1011	How to?	20.11.21	25.11.21	Support	ValdemarNLind@armyspy.com	German	Wolfs	Booking
Jacob K.	Johansen	780	Problem	01.05.21	04.05.21	Support	JacobKJohansen@dayrep.com	German	Tigers Prep	Toys
清崎 正		453	Greeting	23.03.21	27.03.21	Support	Turietylly1995@journapide.com	Japanese	Contact Eye	Leasure
Фома	Базаров	239	Proposal	30.10.21	04.11.21	Support	Patia1948@journapide.com	Russian	Bring Jou	Mechanics
Andrea	Aachen	34	Idea	15.09.21	30.09.21	Support	AndreaAachen@teleworm.us	Estonian	Fun Down	Travel

- Comma-Separated Values: A plain text file format that stores tabular data (numbers and text) in plain text form, where each line of the file is a data record, and fields within each record are separated by commas
- Delimiter: While the default delimiter is a comma, other delimiters like semicolons (;) or tabs (\t) can also be used, depending on the application or region

# CSV



Name	Last Name	CT-ID	Subject	Requested	Updated	Group	Primary email	Language	Company	Industry
Carrie M.	Hill	32	Question	20.11.21	22.11.21	Support	repeat@repeat.com	French	Great Company	Toys
Avis R.	Daigle	112	Request	01.05.21	01.06.21	Support	repeat@repeat.com	English	John Free	Leasure
Richard D.	Humphries	543	Notice	23.03.21	01.04.21	Support	repeat@repeat.com	Portugese	Partner World	Mechanics
Agnes C.	Mitchem	231	Problem	14.01.21	01.02.21	Support	repeat@repeat.com	Italian	Best Tickets	Travel
Ryan N.	Smith	46	Question	15.06.21	17.07.21	Support	repeat@repeat.com	English	Seeking Alpha	Booking
Gemma	Grave	75	Problem	23.08.21	01.09.21	Support	repeat@repeat.com	English	Doughnet	Toys
Emma	Brunker	456	Notice	04.09.21	10.09.21	Support	repeat@repeat.com	English	Terminator	Leasure
Torben C.	Juhl	1000	Inquiry	01.10.21	30.10.21	Support	TorbenCJuhl@armyspy.com	German	Checkers	Mechanics
Johanne K.	Bruun	238	Does not work	15.06.21	01.07.21	Support	JohanneKBruun@teleworm.us	Danish	Brand Worls	Travel
Valdemar N.	Lind	1011	How to?	20.11.21	25.11.21	Support	ValdemarNLind@armyspy.com	German	Wolfs	Booking
Jacob K.	Johansen	780	Problem	01.05.21	04.05.21	Support	JacobKJohansen@dayrep.com	German	Tigers Prep	Toys
清崎 正		453	Greeting	23.03.21	27.03.21	Support	Turietylly1995@journapide.com	Japanese	Contact Eye	Leasure
Фома	Базаров	239	Proposal	30.10.21	04.11.21	Support	Patia1948@journapide.com	Russian	Bring Jou	Mechanics
Andrea	Aachen	34	Idea	15.09.21	30.09.21	Support	AndreaAachen@teleworm.us	Estonian	Fun Down	Travel

- Header Row: CSV files often have a header row at the beginning, which contains the names of the columns
- CSV files are widely supported and can be opened and edited using spreadsheet software like Microsoft Excel or Google Sheet

# DataFrame

- The most commonly used data structure in Pandas
- A two-dimensional, tabular data structure that resembles a spreadsheet or SQL table. Each column can have a different data type (e.g., numbers, text) and is labeled with column names
- **DataFrame** is a container for dataset, with rows and columns
- Series: A Series is a one-dimensional array-like data structure that can store data of a single data type, such as a list or a column in a DataFrame. Series objects are the building blocks of DataFrames

# — Read from CSV

```
import pandas as pd

# Read data from a CSV file into a DataFrame
df = pd.read_csv('data.csv')

# Display the first few rows of the DataFrame
print(df.head())

# Perform data selection and filtering
selected_data = df[df['Category'] == 'Science']

# Calculate statistics
mean_price = selected_data['Price'].mean()
print(f'Mean price of Science books: ${mean_price:.2f}')
```

## ■ Read and manipulate data from a CSV file

# DataFrame

- Pandas supports a wide range of data types, including objects, integers, floats, and datetime.
- This flexibility is especially useful for working with mixed data types in a DataFrame.

# DataFrame (2)

```
import pandas as pd

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

df = pd.DataFrame(data)

# Displaying the DataFrame
print(df)
```

- The central data structure in Pandas is the DataFrame. It is a two-dimensional table with labeled axes (rows and columns).
- Columns can be of different data types (integers, floats, strings, etc.).
- DataFrames can be created from various data sources like CSV files, Excel sheets, SQL queries, and more.

# DataFrame (3)

```
# Selecting a column
print(df['Name'])

# Selecting rows based on a condition
print(df[df['Age'] > 30])
```

```
# Performing operations on columns
df['DoubleAge'] = df['Age'] * 2
print(df)
```

- Rows and columns can be selected using labels or indices.
- Conditional indexing allows one to filter data based on specific conditions.
- DataFrames support various operations, such as arithmetic operations, element-wise operations, and statistical operations.
- Mathematical operations can be performed on entire columns.

# DataFrame

```
# Reading from a CSV file
df = pd.read_csv('data.csv')

# Writing to a CSV file
df.to_csv('output.csv', index=False)
```

- DataFrames can read data from various file formats, such as CSV, Excel, and SQL databases
- They can also write data to these formats

# — Series

```
import pandas as pd

# Creating a Series from a list
data = [10, 20, 30, 40]
series = pd.Series(data, name='MySeries')
print(series)
```

```
# Accessing elements by index
print(series[0]) # Output: 10
print(series[1:3]) # Output: 20, 30
```

- A Series is a one-dimensional labeled array capable of holding any data type
- It is similar to a column in a DataFrame but can be used independently
- A Series has both a data column and an index column
- The index provides a label for each element, and these labels can be used for indexing and slicing

# — Series (2)

```
# Performing operations on a Series
doubled_series = series * 2
print(doubled_series)
```

```
0    20
1    40
2    60
3    80
Name: MySeries, dtype: int64
```

- Series supports element-wise operations and vectorized operations
- Mathematical operations can be performed on entire Series
- Series provides various methods for statistical analysis, such as **mean**, **sum**, **min**, and **max**
- It has methods for handling missing data, like **dropna** and **fillna**

# — Series (3)

```
print(series.mean()) # Output: 25.0
print(series.dropna()) # Output: the Series without NaN values
```

- Series supports element-wise operations and vectorized operations
- Mathematical operations can be performed on entire Series
- Series provides various methods for statistical analysis, such as **mean**, **sum**, **min**, and **max**
- It has methods for handling missing data, like **dropna** and **fillna**

# List of Functions

- Handling Missing Data:
  - **dropna()**: Remove missing values
  - **fillna()**: Fill missing values with a specified value or using various methods
- Dropping Columns or Rows:
  - `drop()`: Remove specified labels from rows or columns
- Deduplicating Data:
  - `duplicated()`: Identify duplicate rows
  - `drop_duplicates()`: Remove duplicate rows
- Replacing Values:
  - `replace()`: Replace values with other values

# List of Functions (2)

- String Manipulation:
  - `str.lower()`, `str.upper()`: Convert strings to lowercase or uppercase
  - `str.strip()`: Remove leading and trailing whitespaces
  - `str.replace()`: Replace a substring with another substring
- Data Transformation:
  - `map()`: Apply a function to each element in a Series
  - `apply()`: Apply a function along the axis of a DataFrame
- Changing Data Types:
  - `astype()`: Convert the data type of a column
- Handling Categorical Data:
  - `astype('category')`: Convert a column to the categorical data type
  - `pd.get_dummies()`: Create dummy variables for categorical columns

# List of Functions (3)

- Handling DateTime Data:
  - `pd.to_datetime()`: Convert a column to datetime format
  - `dt accessor`: Access components of datetime data
- Sorting and Reordering Data:
  - `sort_values()`: Sort DataFrame or Series by specified columns
  - `sort_index()`: Sort DataFrame or Series by index
- Handling Outliers:
  - Various statistical methods and visualization techniques can be used to identify and handle outliers

# Data Manipulation

```
# Adding a new column
df['New_Column'] = df['Column_A'] + df['Column_B']

# Grouping and aggregation
grouped_data = df.groupby('Category')['Value'].sum()
```

- Operations on columns and rows
- Aggregation functions (sum, mean, count, etc.)
- Element-wise operations

# Data Cleaning

```
import pandas as pd
import numpy as np

# Creating a sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'Dave', 'Eve'],
    'Age': [25, 32, np.nan, 40, 22],
    'Salary': [50000, 60000, 75000, np.nan, 48000],
    'Country': ['USA', 'Canada', 'USA', 'Germany', ''],
    'Date_Joined': ['2022-01-15', '2022-02-20', '2022-01-15', '2021-12-10', '2022-03-01']
}

df = pd.DataFrame(data)
```

- A DataFrame named df with the following columns: 'Name', 'Age', 'Salary', 'Country', and 'Date\_Joined'

# Handle Missing Data

```
# Identifying missing values
missing_data = df.isnull().sum()

# Filling missing values (e.g., with mean)
df['Age'].fillna(df['Age'].mean(), inplace=True)

# Removing rows with missing 'Salary'
df.dropna(subset=['Salary'], inplace=True)
```

```
# Identifying and removing duplicates
df.drop_duplicates(inplace=True)
```

- Identify missing values and decide whether to remove or fill them
- The **inplace=True** parameter in pandas functions is used to perform the operation directly on the existing object without the need to assign the result back to a new variable
- When **inplace=True**, the changes are made directly to the original object, and it modifies the object in place

# fillna()

```
DataFrame.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None)
```

- **value:** The value to use for filling NaN entries. This can be a scalar, a dictionary, a Series, or a DataFrame.
- **method:** The method to use for filling NaN values. It can be 'pad' (forward fill), 'bfill' (backward fill), or None (default).
- **axis:** Specifies the axis along which to fill NaN values (0 for rows, 1 for columns).
- **inplace:** If True, modifies the DataFrame in place; otherwise, it returns a new DataFrame.
- **limit:** Limit the number of consecutive NaN values to fill.
- **downcast:** Downcast the resulting non-null values to a smaller datatype.

- Fills missing (NaN) values with specified values or using certain filling methods
- It provides flexibility in handling missing data by allowing one to fill NaN values with a constant, a calculated value, or methods like forward fill or backward fill

# Pandas



- Python Data Analysis Library: A popular and powerful Python library for data manipulation and analysis
- It provides easy-to-use data structures and functions for working with structured data, such as spreadsheets or SQL tables
- Pandas is widely used in data science, data analysis, and machine learning tasks

# — fillna()

```
import pandas as pd

data = {'Name': ['Alice', 'Bob', None, 'Charlie'],
        'Age': [25, None, 22, 30],
        'Salary': [50000, 60000, None, 75000]}

df = pd.DataFrame(data)

# Fill NaN values with a constant
df_filled = df.fillna(value={'Name': 'Unknown', 'Age': 0, 'Salary': 0})

print(df_filled)
```

- The `fillna()` function is used to fill `NaN` values in the `DataFrame` with specific values
- The 'Name' column is filled with 'Unknown', the 'Age' column with 0, and the 'Salary' column with 0

# — inplace

```
import pandas as pd
import numpy as np

# Creating a DataFrame with missing values
data = {'A': [1, 2, np.nan, 4], 'B': [5, np.nan, np.nan, 8]}
df = pd.DataFrame(data)

# Display the original DataFrame
print("Original DataFrame:")
print(df)

# Drop rows with missing values in place
df.dropna(inplace=True)

# Display the modified DataFrame
print("\nDataFrame after dropping rows with missing values in place:")
print(df)
```

```
# Drop rows with missing values and assign
the result back to df
df = df.dropna()
```

- Calling `df.dropna(inplace=True)` removes the rows containing missing values directly from the original DataFrame `df`
- If `inplace=False` or is not specified (as it is by default), a new DataFrame with the missing values removed would be returned, and it is necessary to assign it back to a variable

# — Correcting Inconsistent Data

```
# Standardizing 'Country' values
df['Country'] = df['Country'].str.upper().str.strip()
```

```
# Standardizing date format
df['Date_Joined'] = pd.to_datetime(df['Date_Joined'],
format='%Y-%m-%d')
```

```
# Replacing inconsistent 'Country' values
df['Country'].replace({'USA': 'United States', 'CANADA': 'Canada'}, inplace=True)
```

- Standardize text data with `upper()`
- Convert date and time to a unified format

# The `dropna()` function

- It is widely used for data manipulation and analysis in Python
- This function is used to remove missing or NaN (Not-a-Number) values from a DataFrame or Series, allowing you to clean and prepare your data for further analysis
- It gives you control over how you handle missing values, whether you choose to remove rows, columns, or a combination based on your specific data requirements

# The dropna() function

```
DataFrame.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)
```

- `axis`: Specifies whether to drop rows (`axis=0`) or columns (`axis=1`) containing missing values.
- `how`: Determines the conditions under which the row or column is dropped.
  - '`any`': Drops the row or column if any NaN values are present.
  - '`all`': Drops the row or column only if all values are NaN.
- `thresh`: Requires at least `thresh` non-null values for the row or column to be retained.
- `subset`: Specifies a subset of columns or rows to consider for missing values.
- `inplace`: If `True`, modifies the DataFrame in place; otherwise, it returns a new DataFrame.

- The `dropna()` function in pandas is used to remove missing values (NaN) from a DataFrame or Series
- It provides a flexible way to handle missing data by allowing you to specify how the removal should be performed

# Handle Outliers and Data Integrity

```
# Identifying outliers
salary_outliers = df[(df['Salary'] < 50000) | (df['Salary'] >
80000)]

# Removing outliers
df = df[(df['Salary'] >= 50000) & (df['Salary'] <= 80000)]
```

```
# Checking data integrity
invalid_data = df[df['Age'] > 130]

# Correcting data integrity issues (e.g., setting age to
NaN)
df.loc[invalid_data.index, 'Age'] = np.nan
```

- Identify and handle outliers in the 'Salary' column
- Check and correct data integrity issues

# — Drop with the `dropna()` function

## ■ Removing Rows with Missing Values

```
import pandas as pd

data = {'A': [1, 2, None, 4], 'B': [None, 2, 3, 4]}
df = pd.DataFrame(data)

# Remove rows with any NaN values
cleaned_df = df.dropna()
```

## ■ Removing Columns with Missing Values:

```
# Remove columns with all NaN values
cleaned_df = df.dropna(axis=1, how='all')
```

# — Drop with the `dropna()` function

## ■ Specifying a Threshold

```
# Keep rows with at least 2 non-NaN values
cleaned_df = df.dropna(thresh= 2)
```

## ■ Removing Rows Based on a Subset of Columns

```
# Remove rows where column 'A' has NaN values
cleaned_df = df.dropna(subset=[ 'A' ])
```

# The Pandas library (2)

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# — Reading data

- `pd.read_csv()`: Read CSV files.
- `pd.read_excel()`: Read Excel files.
- `pd.read_json()`: Read JSON files.
- `pd.read_sql()`: Query data from SQL databases.
- `pd.read_html()`: Read HTML tables.
- `pd.read_parquet()`: Read Parquet files for large datasets.

# Writing data

- `to_csv()`: Write DataFrame to a CSV file.
- `to_excel()`: Write DataFrame to an Excel file.
- `to_json()`: Write DataFrame to a JSON file.
- `to_sql()`: Write DataFrame to an SQL table.
- `to_parquet()`: Write DataFrame to a Parquet file.

# Data inspection

- `df.head(n)`: View the first n rows.
- `df.tail(n)`: View the last n rows.
- `df.info()`: Get an overview of the dataset (columns, types, non-null counts).
- `df.describe()`: Get summary statistics for numerical columns.
- `df.dtypes`: View data types of all columns.
- `df.astype()`: Change the data type of a column.

# Data selection

- `df[“column”]`: Access a single column.
- `df[[“col1”, “col2”]]`: Access multiple columns.
- `df.loc[row_label, col_label]`: Select rows and columns by labels.
- `df.iloc[row_index, col_index]`: Select rows and columns by index.
- Conditional filtering: `df[df[“column”] > value]`
- Multiple conditions: `df[(df[“col1”] > value1) & (df[“col2”] < value2)]`

# Data cleaning

- `df.isna()`: Detect missing values.
- `df.fillna(value)`: Fill missing values with a specified value.
- `df.dropna()`: Remove rows or columns with missing values.
- `df.interpolate()`: Fill missing values using interpolation.
- `df.duplicated()`: Check for duplicate rows.
- `df.drop_duplicates()`: Remove duplicate rows.

# Logical operators

	Name	Age	Salary	City
0	Alice	25.0	50000.0	New York
1	Bob	30.0	60000.0	Chicago
2	Charlie	NaN	55000.0	San Francisco
3	David	28.0	NaN	Los Angeles

```
import pandas as pd
from google.colab import drive
drive.mount('/content/drive/', force_remount=False)
DATA_PATH = '/content/drive/My Drive/Data Science/'
from os.path import join
data_file = join(DATA_PATH, 'example_data.csv')

df = pd.read_csv(data_file)
df.head()

# Use Logical Operators to Filter Data:
# Filter rows where Age is greater than 25 and Salary is less than 60000
filtered_df = df[(df['Age'] > 25) & (df['Salary'] < 60000)]
print("DataFrame after filtering rows where Age > 25 and Salary < 60000:")
print(filtered_df)
print("\n")
```

- Filter rows where Age is greater than 25, and Salary is less than 60000.

# Logical operators

	Name	Age	Salary	City
0	Alice	25.0	50000.0	New York
1	Bob	30.0	60000.0	Chicago
2	Charlie	NaN	55000.0	San Francisco
3	David	28.0	NaN	Los Angeles

```
# Filter rows where City is 'New York' or 'Chicago'  
filtered_df = df[(df['City'] == 'New York') | (df['City'] == 'Chicago')]  
print("DataFrame after filtering rows where City is 'New York' or 'Chicago':")  
print(filtered_df)  
print("\n")
```

- Selecting rows that satisfy some requirements

# Logical operators

	Name	Age	Salary	City
0	Alice	25.0	50000.0	New York
1	Bob	30.0	60000.0	Chicago
2	Charlie	NaN	55000.0	San Francisco
3	David	28.0	NaN	Los Angeles

```
# Filter rows where Age is not equal to 30
filtered_df = df[df['Age'] != 30]
print("DataFrame after filtering rows where Age is not equal to 30:")
print(filtered_df)
```

- Selecting rows whose Age is not equal to 30

# Logical operators

	Name	Age	Salary	City
0	Alice	25.0	50000.0	New York
1	Bob	30.0	60000.0	Chicago
2	Charlie	NaN	55000.0	San Francisco
3	David	28.0	NaN	Los Angeles

```
# Filter rows where Age is not equal to 30
filtered_df = df[df['Name'] > 'Alice']
print("DataFrame after filtering rows where Name is larger than Alice")
print(filtered_df)
```

- Logical operators work also with String data

# isin()

	Name	Age	Salary	City
0	Alice	25.0	50000.0	New York
1	Bob	30.0	60000.0	Chicago
2	Charlie	NaN	55000.0	San Francisco
3	David	28.0	NaN	Los Angeles

```
names = ['Charlie','David','Mike']
filtered_df = df[df.Name.isin(names)]
print("DataFrame after filtering rows where names are prespecified:")
print(filtered_df)
```

- Find names that belong to a predefined set

# Str accessor

	Name	Age	Salary	City
0	Alice	25.0	50000.0	New York
1	Bob	30.0	60000.0	Chicago
2	Charlie	NaN	55000.0	San Francisco
3	David	28.0	NaN	Los Angeles

```
filtered_df = df[df.Name.str.startswith('A')]
print("DataFrame after filtering rows where names start with A:")
print(filtered_df)
```

```
filtered_df = df[df.Name.str.contains('a')]
print("DataFrame after filtering rows where names contain a:")
print(filtered_df)
```

- Find names that start with a specific character

# Query

	Name	Age	Salary	City
0	Alice	25.0	50000.0	New York
1	Bob	30.0	60000.0	Chicago
2	Charlie	NaN	55000.0	San Francisco
3	David	28.0	NaN	Los Angeles

```
query_expression = 'Age > 25 and Salary < 60000'  
filtered_df = df.query(query_expression)  
print("DataFrame after querying rows where Age > 25 and Salary < 60000:");  
print(filtered_df)
```

```
# Query rows where City is 'New York' or 'Chicago'  
query_expression = "City in ['New York', 'Chicago']"  
filtered_df = df.query(query_expression)  
print("DataFrame after querying rows where City is 'New York' or 'Chicago':");  
print(filtered_df)  
print("\n")
```

- The query method provides a convenient way to filter a DataFrame based on a query expression.
- It allows us to express conditions using a more SQL-like syntax.

# Exercises

- Read the **gapminder\_full.csv** file (Files → Materials → Data).
- Drop rows containing cells with NaN values.
- Drop some columns, keep some columns.



# Average Life Expectancy Over Time

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

# Set style for the plots
sns.set_theme(style="whitegrid")

# Global Trend: Average Life Expectancy Over Time
avg_life_exp = df.groupby("year") ["life_exp"].mean()

plt.figure(figsize=(12, 6))
sns.lineplot(x=avg_life_exp.index, y=avg_life_exp.values, marker="o",
color="blue")
plt.title("Average Life Expectancy Over Time (1952-2007)", fontsize=16)
plt.xlabel("Year", fontsize=12)
plt.ylabel("Life Expectancy (Years)", fontsize=12)
plt.grid(axis="y", linestyle="--", alpha=0.7)
plt.show()
```



# The NumPy library

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# — Introduction

- A library for numerical and scientific computing.
- It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
- NumPy is a fundamental library for data science, machine learning, and scientific research in Python.

# — Introduction

- NumPy is known for its high performance and efficiency. It is written in C and optimized for speed, making it suitable for large-scale numerical computations.
- The focus is on **array operations**, **vectorized operations**, and **broadcasting**, allowing us to perform operations on entire arrays without explicit looping.
- NumPy allows us to perform fast and versatile operations on arrays, such as vectorization, indexing, and broadcasting.
- It offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more.

# — Introduction

- Pandas and NumPy complement each other in the Python data science ecosystem.
- Pandas is preferred for data manipulation, cleaning, and structured data analysis, while NumPy is the go-to choice for numerical computing and scientific applications.
- Data scientists often use both libraries together, utilizing Pandas for data preprocessing and exploration and NumPy for numerical computations and array-based operations.

# — Introduction

- A wide range of data types for numerical computing, such as int, float, complex, and more.
- It does not support mixed data types within a single array.
- It excels at numerical computation, linear algebra, and mathematical operations.
- It provides a vast array of functions for **element-wise** and array-level computations.

# Key Features

# — Arrays

- Arrays: The core feature of NumPy is the ndarray (n-dimensional array).
- NumPy arrays provide a uniform, efficient way to represent and manipulate data.
- These arrays can be one-dimensional (vectors), two-dimensional (matrices), or even higher-dimensional (**tensors**). NumPy arrays are more efficient than Python lists for numerical operations.

# Operations

```
import numpy as np

# Creation of Arrays
arr1 = np.array([1, 2, 3])
arr2 = np.zeros(2, 3)
arr3 = np.ones(3, 2)
arr4 = np.full(2, 2, 7)
arr5 = np.arange(0, 10, 2)
arr6 = np.linspace(0, 1, 5)

# Array Operations
addition_result = arr1 + 5
subtraction_result = arr1 - 1
multiplication_result = arr1 * 2
division_result = arr1 / 2
dot_product_result = np.dot(arr2, arr3.T)
matrix_multiply_result = np.matmul(arr3, arr4)
transpose_result = np.transpose(arr3)
sum_result = np.sum(arr1)
mean_result = np.mean(arr1)
median_result = np.median(arr1)
min_value = np.min(arr1)
max_value = np.max(arr1)
argmax_index = np.argmax(arr1)
```

# Operations

```
# Array Manipulation
reshaped_arr = np.reshape(arr1, (3, 1))
concatenated_arr = np.concatenate((arr2, arr3), axis=1)
split_arr = np.split(arr5, [2, 4])
appended_arr = np.append(arr1, [4, 5])
deleted_arr = np.delete(arr1, 1)
inserted_arr = np.insert(arr1, 1, 10)

# Linear Algebra
det_result = np.linalg.det(arr4)
inv_result = np.linalg.inv(arr4)
eig_values, eig_vectors = np.linalg.eig(arr4)
svd_result = np.linalg.svd(arr3)
```

```
print("Creation of Arrays:")
print(arr1)
print(arr2)
print(arr3)
print(arr4)
print(arr5)
print(arr6)

print("\nArray Operations:")
print(addition_result)
print(subtraction_result)
print(multiplication_result)
print(division_result)
print(dot_product_result)
print(matrix_multiply_result)
print(transpose_result)
print(sum_result)
print(mean_result)
print(median_result)
print(min_value)
print(max_value)
print(argmax_index)
```

# Arrays

```
import numpy as np

arr_1d = np.array([1, 2, 3, 4, 5])
print(arr_1d)

arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr_2d)

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Element-wise addition
result_addition = arr1 + arr2
print(result_addition)

# Element-wise multiplication
result_multiplication = arr1 * arr2
print(result_multiplication)
```

- We can easily perform element-wise operations.

# Operations

- NumPy provides a comprehensive set of linear algebra functions that work seamlessly with matrices, making it a powerful tool for numerical computation and scientific computing.
- The focus is on array operations, vectorized operations, and broadcasting, allowing us to perform operations on entire arrays without explicit looping.
- NumPy allows us to perform fast and versatile operations on arrays, such as vectorization, indexing, and broadcasting.

# — Array Indexing and Slicing

```
import numpy as np

# Creating a 1D array
arr_1d = np.array([1, 2, 3])

# Creating a 2D array (or matrix)
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Creating an array of zeros
zeros_arr = np.zeros((2, 3))

# Creating an array of ones
ones_arr = np.ones((3, 2))

# Creating an identity matrix
identity_mat = np.eye(3)
```

```
# Accessing elements
element = arr_1d[0]

# Slicing
sliced_arr = arr_1d[1:3]

# Boolean indexing
filtered_arr = arr_1d[arr_1d > 1]
```

- Accessing elements, getting a slice of the array.

# — Array Shape Manipulation

```
# Reshaping an array
reshaped_arr = arr_1d.reshape((3, 1))

# Flattening a multi-dimensional array
flattened_arr = arr_2d.flatten()
```

## ■ Accessing elements, getting a slice of the array

# Matrix Creation and Multiplication

```
import numpy as np

# Creating a matrix from a 2D array
matrix = np.array([[1, 2, 3], [4, 5, 6]])

# Creating a matrix of zeros
zeros_matrix = np.zeros((2, 3))

# Creating a matrix of ones
ones_matrix = np.ones((3, 2))

# Creating an identity matrix
identity_matrix = np.eye(3)
```

```
# Using the `@` operator for matrix multiplication
result = matrix @ identity_matrix

# Using the `dot` function
result_dot = np.dot(matrix, identity_matrix)
```

- Identity matrix: Matrix with 1 in the diagonal, and 0 in the other cells.

# Slicing

```
import numpy as np

# Creating a matrix
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Extracting a submatrix
submatrix = matrix[1:3, 0:2]

print(submatrix)

# Selecting entire rows or columns
row = matrix[1, :] # Second row
column = matrix[:, 2] # Third column

# Creating a boolean mask
mask = matrix > 5

# Using the boolean mask for slicing
result = matrix[mask]
```

## ■ Getting a part of the original matrix

# — Assigning Values

```
# Modifying values using slicing
matrix[1:3, 0:2] = 0

# Assigning specific values using boolean indexing
matrix[matrix > 5] = 10
```

## ■ Getting a part of the original matrix

# Matrix Multiplication

```
import numpy as np

matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])

result_matrix_multiplication = np.dot(matrix1, matrix2)
print(result_matrix_multiplication)
```

- We can easily perform element-wise operations

# Transpose a Matrix

```
import numpy as np
matrix = np.array([[1, 2, 3], [4, 5, 6]])
transposed_matrix = np.transpose(matrix)
print(transposed_matrix)
```

- Transpose swaps the rows and columns of a matrix

# Matrix Concatenation

```
import numpy as np

matrix1 = np.array([1, 2], [3, 4])
matrix2 = np.array([5, 6])

# Transpose matrix2 to have the same number of rows as matrix1
matrix2_transposed = np.transpose(matrix2)

# Concatenate horizontally
concatenated_horizontal = np.concatenate((matrix1, matrix2_transposed), axis=1)

# Concatenate vertically
concatenated_vertical = np.concatenate((matrix1, matrix2), axis=0)

print("Concatenated Horizontally:\n", concatenated_horizontal)
print("Concatenated Vertically:\n", concatenated_vertical)
```

- We can concatenate matrices horizontally or vertically

# Matrix Multiplication

```
import numpy as np
# Creating two matrices
matrix1 = np.array([[1, 2, 3], [4, 5, 6]])
matrix2 = np.array([[7, 8], [9, 10], [11, 12]])

# Using np.dot() for matrix multiplication
result_dot = np.dot(matrix1, matrix2)
# Using @ operator for matrix multiplication (available in Python 3.5
# and later)
result_at = matrix1 @ matrix2

print("Result using np.dot():")
print(result_dot)

print("\nResult using @ operator:")
print(result_at)
```

- The number of columns in the first matrix (matrix1) matches the number of rows in the second matrix (matrix2)
- The result will be a new matrix with dimensions equal to the number of rows in the first matrix and the number of columns in the second matrix

# Handle errors

```
import numpy as np

# Creating two matrices
matrix1 = np.array([[1, 2, 3], [4, 5, 6]])
matrix2 = np.array([[7, 8], [9, 10], [11, 12]])

try:
    # Attempting matrix multiplication
    result = np.dot(matrix1, matrix2)
except ValueError as e:
    # Handling the ValueError
    print(f"Error: {e}")
    print("Matrix multiplication is not possible with the given matrices.")
```

- If we try to perform matrix multiplication with incompatible shapes, we will encounter a `ValueError`
- This error indicates that the shapes of the matrices are not suitable for matrix multiplication

# Handle errors

```
import numpy as np

# Creating two matrices
matrix1 = np.array([[1, 2, 3], [4, 5, 6]])
matrix2 = np.array([[7, 8], [9, 10], [11, 12]])

try:
    # Attempting matrix multiplication
    result = np.dot(matrix1, matrix2)
except ValueError as e:
    # Handling the ValueError
    print(f"Error: {e}")
    print("Matrix multiplication is not possible with the given matrices.")
```

- If the matrices have incompatible shapes, the `np.dot()` function will raise a `ValueError`, and the code inside the `except` block will be executed
- Handling errors in this manner helps prevent your program from crashing and allows you to provide informative error messages to the user

# Element-wise Comparison

```
import numpy as np

matrix_a = np.array([[1, 2], [3, 4]])
matrix_b = np.array([[0, 2], [4, 4]])

# Element-wise greater than
greater_than_result = matrix_a > matrix_b
print("Greater Than Result:\n", greater_than_result)
```

- We can concatenate matrices horizontally or vertically.

# Element-wise Operations

```
import numpy as np

# Sample matrices
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])

# 1. Element-wise Operations
result_add = matrix1 + matrix2
result_sub = matrix1 - matrix2
result_mul = matrix1 * matrix2
result_div = matrix1 / matrix2

# Displaying Results
print("Matrix1:\n", matrix1)
print("Matrix2:\n", matrix2)
print("\n1. Element-wise Operations:")
print("Addition:\n", result_add)
print("Subtraction:\n", result_sub)
print("Multiplication:\n", result_mul)
print("Division:\n", result_div)
```

- We can concatenate matrices horizontally or vertically.

# Matrix Reshaping

```
import numpy as np

# Create a 1D array with 12 elements
arr_1d = np.arange(12)

# Reshape to a 2D array with 3 rows and 4 columns
arr_2d = arr_1d.reshape(3, 4)

print("Original 1D Array:")
print(arr_1d)
print("\nReshaped 2D Array:")
print(arr_2d)
```

- Reshaping is a common operation when working with matrices, and it allows us to change the dimensions of the array
- In NumPy, we can use the reshape function to achieve this

# Matrix Reshaping

```
import numpy as np

# Create a 2D array
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Reshape to a 1D array
arr_1d = arr_2d.reshape(-1)

print("Original 2D Array:")
print(arr_2d)
print("\nReshaped 1D Array:")
print(arr_1d)
```

- The -1 argument in the second example is a placeholder that means “whatever is needed” to make the reshape operation valid
- This is useful when we want to reshape an array to a 1D array without specifying the size

# — Reshaping to a different shape

```
import numpy as np

# Create a 1D array with 8 elements
arr_1d = np.arange(8)

# Reshape to a 2D array with 2 rows and 4 columns
arr_reshaped = arr_1d.reshape(2, 4)

print("Original 1D Array:")
print(arr_1d)
print("\nReshaped 2D Array:")
print(arr_reshaped)
```

- Reshaping is a common operation when working with matrices, and it allows us to change the dimensions of the array
- In NumPy, we can use the reshape function to achieve this

# — Functions & Broadcasting

- NumPy includes a wide range of mathematical functions for performing operations on arrays, including basic arithmetic, linear algebra, Fourier analysis, and more
- It also provides random number generation functions
- NumPy allows operations between arrays of different shapes and sizes through a mechanism called **broadcasting**
- Broadcasting makes it possible to perform element-wise operations even when the arrays have different dimensions

# — Functions & Broadcasting

- Indexing and Slicing: NumPy provides powerful indexing and slicing capabilities for arrays, allowing you to extract specific elements or sub-arrays efficiently.
- Vectorization: NumPy encourages vectorized operations, which means that operations are performed on entire arrays rather than looping through individual elements. This leads to more concise and efficient code.

# Broadcasting

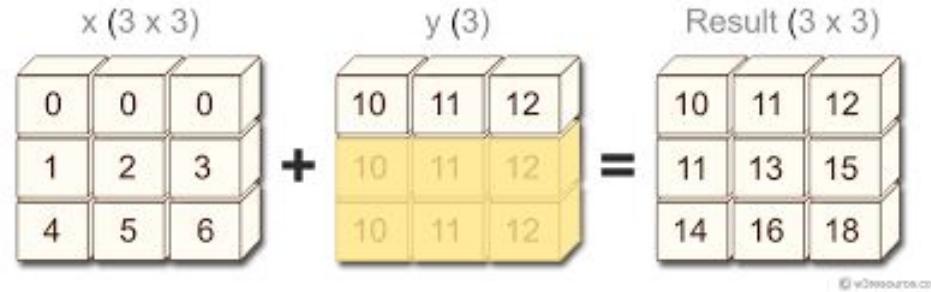


Image source:

<https://www.w3resource.com/python-exercises/numpy/python-numpy-exercise-124.php>

- Broadcasting is widely used in operations involving arrays of different shapes, making code more concise and readable.
- It simplifies operations like addition, subtraction, multiplication, etc., especially when working with multi-dimensional arrays.

# Broadcasting

- Broadcasting is a powerful feature in NumPy that allows arrays of different shapes and sizes to be combined in element-wise operations.
- This eliminates the need for explicit loops over array elements and makes the code more concise and readable.
- It implicitly expands the smaller array to match the shape of the larger one.

# Broadcasting: Rules

- Two dimensions are compatible when they are equal, or one of them is 1.
- The smaller array is “broadcast” across the larger array so that they have compatible **shapes**.

```
import numpy as np

matrix = np.array([[1, 2, 3], [4, 5, 6]])
scalar = 2

result = matrix * scalar
print(result)
```

- In this example, the scalar 2 is broadcasted to the shape of the matrix, and the multiplication is performed element-wise.

# Broadcasting with larger arrays

- Broadcasting can also happen when one array has more dimensions than the other.
- In this case, the smaller array is broadcasted along the dimensions of the larger array.

```
matrix = np.array([[1, 2, 3], [4, 5, 6]])
row_vector = np.array([10, 20, 30])

result = matrix + row_vector
```

- The row vector is broadcasted along the rows of the matrix.

# Linear Algebra Operations

- NumPy provides a rich set of functions for linear algebra operations, including matrix multiplication, inversion, eigenvalue decomposition, and more.
- These functionalities make NumPy suitable for a wide range of scientific and engineering applications.
- NumPy includes a random module (`numpy.random`) for generating random numbers. This is useful for tasks such as random sampling, permutation, and creating random arrays.

# Working with Files and Folders

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# — Introduction

- Files and folders are fundamental components of any computer's file system, serving as the primary means of organizing, storing, and accessing digital data.
- Files and folders play an essential role in managing digital information. They enable organization, access control, and retrieval of data in a structured way, supported by underlying file systems that facilitate storage and retrieval.

# Files

- Definition: A file is a digital container used to store data, information, or instructions on a computer. Files can contain various types of data, including text, images, videos, and executable code.
- Types of Files: Files can be broadly categorized into text files (e.g., .txt, **.csv**), binary files (e.g., .exe, .jpg), and system files (e.g., .dll, .sys). The type of file often determines how it can be opened or processed.

# Files

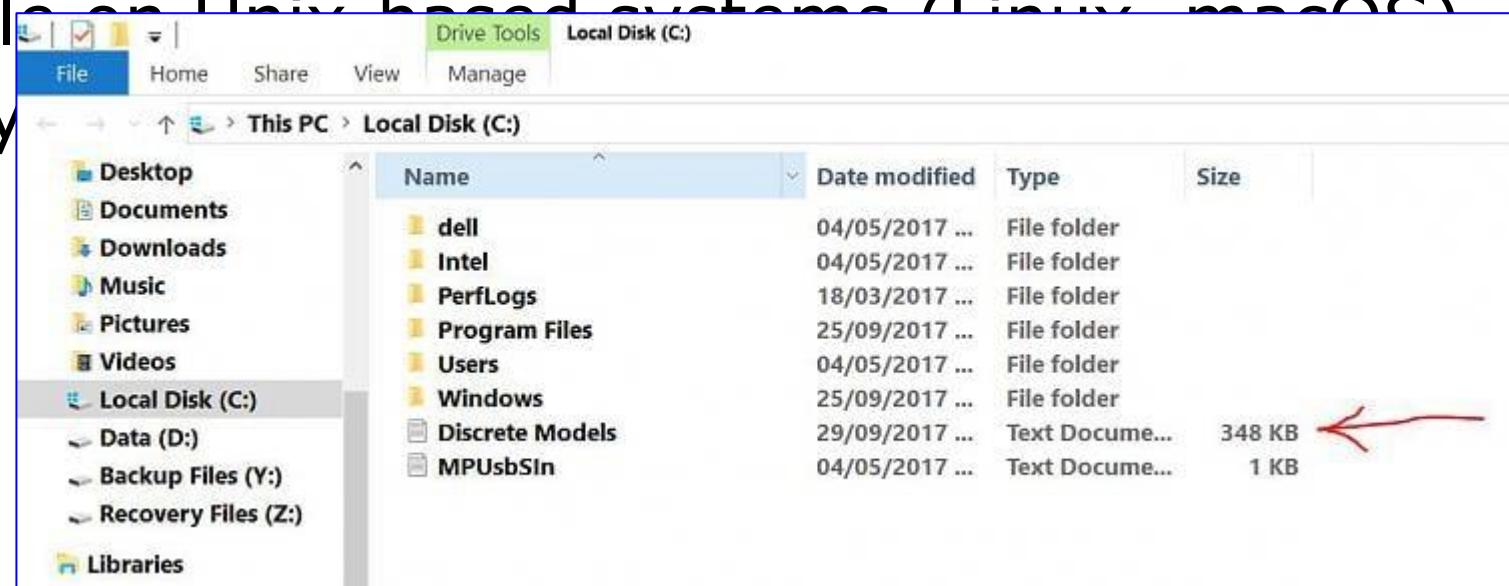
- **File Extensions:** Files are identified by extensions, such as .docx for Word documents, .mp3 for audio files, or .py for Python scripts. Extensions help the operating system determine which program to use to open the file.
- **Structure:** Files have a structure determined by the software that created them. For example, a .csv file organizes data in a tabular form, while a .json file organizes data in key-value pairs.

# Folders

- Definition: A folder, also known as a directory, is a virtual container used to organize files and other folders. Folders make it easier to group related files, creating a hierarchical structure in the file system.
- Hierarchy: Folders can contain both files and subfolders, allowing for a nested structure that mirrors a tree-like hierarchy. This organization helps to structure data, making it easier to locate and manage.

# Folders

- Root Directory: The topmost level of a file system is known as the root directory, which contains all other directories and files.
- In Windows, root directories are assigned drive letters like C:\, while on Unix-based systems (Linux, macOS) the root directory



# Paths

- Absolute Path: Specifies the complete path to a file or folder from the root directory (e.g., C:\Users\John\Documents\file.txt).
- Relative Path: Specifies the path relative to the current working directory (e.g., Documents/file.txt).
- Paths are crucial for accessing files and folders programmatically, allowing software to locate and manage files within the file system.

# File and Folder Operations

- Common Operations: Files and folders support operations such as creating, deleting, renaming, moving, and copying.
- File Permissions: File systems use permissions to control who can read, write, or execute a file. These permissions are essential for data security, especially on shared systems.
- Metadata: Files and folders store metadata, such as creation date, modification date, and size. Metadata helps in organizing, sorting, and managing files effectively.

# Creating Files and Folders

- **Files:** Creating a file can be done using applications (e.g., creating a document in Word) or programmatically through commands (e.g., using Python or shell commands).
- **Folders (Directories):** Folders are used to group related files. They can be created within an operating system interface (like Windows Explorer or Finder on macOS) or programmatically.

```
open("newfile.txt", "w").close() # Creates a new file  
import os  
os.makedirs("new_folder") # Creates a new folder
```

# Reading and Writing to Files

- Reading: This retrieves data from a file. Modes include reading the whole file, reading line-by-line, or reading binary data for non-text files.
- Writing: Writing adds new data to a file. Using “w” mode overwrites the file, while “a” mode appends to it.
- Binary Files: Reading or writing binary files (e.g., images) requires specifying “rb” or “wb” modes.

```
# Writing
with open("example.txt", "w") as file:
    file.write("Hello, world!")

# Reading
with open("example.txt", "r") as file:
    print(file.read())
```

# — Introduction

- File management in Python involves working with files, which can include tasks such as reading from files, writing to files, and performing various file-related operations.
- There are also operations related to the management of directories.
- It is important to handle paths, and perform operations such as copying, moving, and deleting files/directories.

# — Opening and Closing Files

```
# Example: Opening and closing a file
file = open("example.txt", "r")
# Perform operations on the file
file.close()
```

- The **open()** function is used to open a file. It takes two arguments: the file name and the mode (read, write, etc.).
- After performing file operations, it's essential to close the file using the `close()` method to free up system resources.

# — Reading from Files

```
# Example: Reading from a file
with open("example.txt", "r") as file:
    content = file.read()
    line = file.readline()
    lines = file.readlines()
```

- The **read()** method reads the entire content of a file.
- The **readline()** method reads a single line from the file.
- The **readlines()** method reads all lines of the file and returns them as a list.

# Writing to Files

```
# Example: Writing to a file
with open("example.txt", "w") as file:
    file.write("Hello, World!\n")
    file.writelines(["Line 1\n", "Line
2\n"])
```

- The **write()** method writes a string to a file.
- The **writelines()** method writes a list of strings to a file.

# File Modes

```
# Example: Opening a file in write mode
with open("example.txt", "w") as file:
    file.write("This will overwrite the existing content." )
```

- When opening a file, you specify a mode that defines the file's purpose (read, write, append, etc.).
- Common file modes include “r” (read), “w” (write), “a” (append), “b” (binary mode), and “x” (exclusive creation).

# — Read and Write

```
# Open a file for reading
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

```
# Open a file for writing
with open('example.txt', 'w') as file:
    file.write('Hello, World!\nThis is a sample file.' )
```

- Open an existing file, and read and display its contents.
- Open an existing file, and write new contents.

# Exception handling

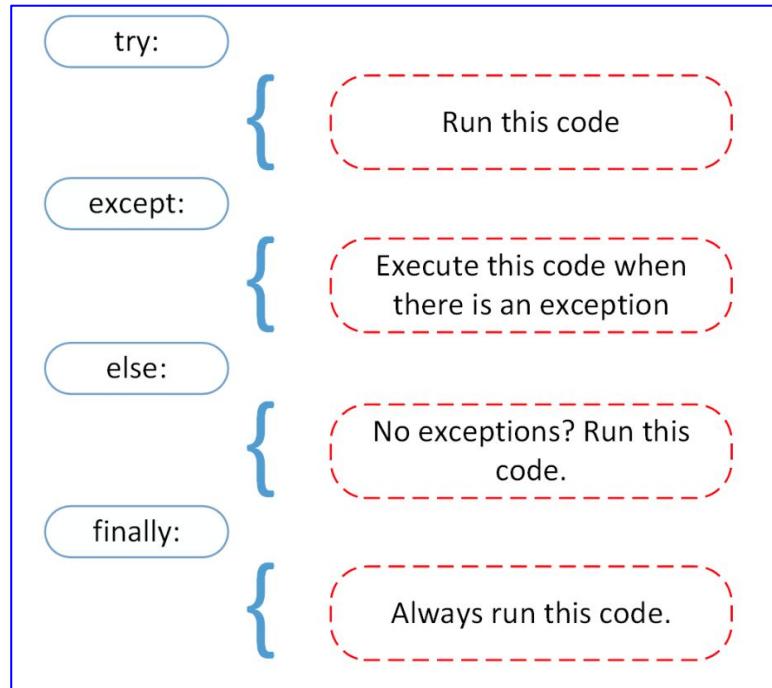


Image source: <https://realpython.com/python-exceptions/>

- When dealing with file operations, it's crucial to handle exceptions to account for potential issues such as file not found, permission errors, etc.

# Multiple exceptions

```
try:  
    # File operations  
except FileNotFoundError:  
    # Handle file not found error  
except PermissionError:  
    # Handle permission error  
except IOError:  
    # Handle general I/O error  
except Exception as e:  
    # Handle other unexpected errors
```

```
try:  
    # File operations  
except FileNotFoundError:  
    # Handle file not found error  
except Exception as e:  
    # Handle other errors  
finally:  
    # Cleanup code (e.g., closing files)
```

- Different file-related exceptions may occur, handle them separately to provide specific error messages
- The finally block ensures that cleanup code is executed, regardless of whether an exception occurred or not. Useful for closing files and releasing resources

# Exception handling

```
# Example: Handling file not found exception
try:
    with open("nonexistent_file.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("File not found.")
```

- It's good practice to handle exceptions when working with files to address potential errors, such as file not found or insufficient permissions

# Example (1)

```
import os

def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            content = file.read()
            print(content)
    except FileNotFoundError:
        print(f"Error: File not found at {file_path}")
    except PermissionError:
        print(f"Error: Permission denied for {file_path}")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

- When dealing with file operations, it's crucial to handle exceptions to account for potential issues such as file not found, permission errors, etc.
- Error when reading files: not found, permission, other exceptions

# Example (2)

```
import os

def write_to_file(file_path, content):
    try:
        with open(file_path, 'w') as file:
            file.write(content)
            print(f"Content written to {file_path}")
    except PermissionError:
        print(f"Error: Permission denied for {file_path}")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

- Error when writing files: not found, permission, other exceptions

# Example (3)

```
def main():
    current_dir = os.getcwd()
    file_path = os.path.join(current_dir, 'example.txt')

    # Read from a file
    read_file(file_path)

    # Write to a file
    new_content = "This is new content."
    write_to_file(file_path, new_content)

if __name__ == "__main__":
    main()
```

- Error when writing files: not found, permission, other exceptions

# FileNotFoundError

```
try:  
    with open("nonexistent_file.txt", "r") as file:  
        content = file.read()  
except FileNotFoundError:  
    print("File not found.")
```

- Raised when attempting to open or operate on a file that does not exist

# PermissionError

```
try:  
    with open("/root/sensitive_file.txt", "w") as file:  
        file.write("Top-secret data.")  
except PermissionError:  
    print("Permission denied.")
```

- Raised when there's a lack of permissions to perform the specified file operation

# — IOError

```
try:  
    with open("corrupted_file.txt", "r") as file:  
        content = file.read()  
except IOError as e:  
    print(f"An IOError occurred: {e}")
```

- A general input/output error that can occur during file operations

# ValueError

```
try:  
    with open("example.txt", "invalid_mode") as file:  
        content = file.read()  
except ValueError:  
    print("Invalid file mode.")
```

- Can occur when incorrect values are passed, such as an invalid file mode

# Exception

```
try:  
    with open("example.txt", "r") as file:  
        content = file.read()  
except Exception as e:  
    print(f"An unexpected error occurred: {e}")
```

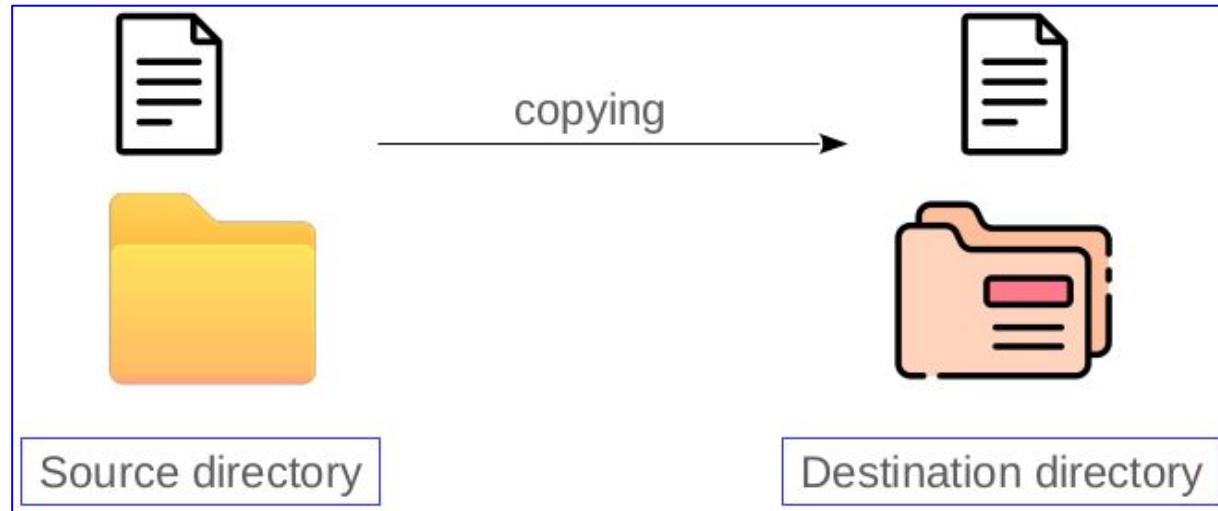
- Catching a general exception can be useful for handling any unexpected errors

# finally

```
try:  
    with open("example.txt", "r") as file:  
        content = file.read()  
except FileNotFoundError:  
    print("File not found.")  
finally:  
    print("Cleanup or finalization code here.")
```

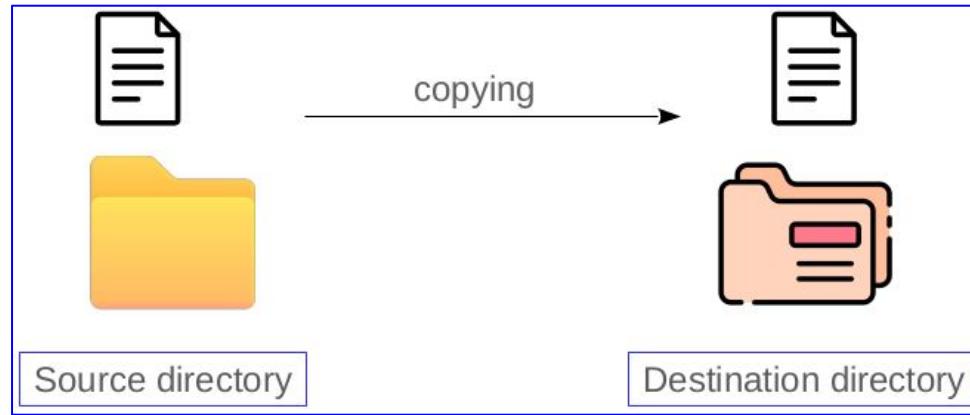
- The finally block is executed regardless of whether an exception is raised or not. It is useful for cleanup operations

# Copying Files



- Copy a file, and paste it into a new directory.
- Keep both files.

# Moving Files



- Copy a file, and paste it into a new directory.
- Delete the file in the source directory.

# Copying a File

```
import shutil

# Copy a file
shutil.copy('example.txt', 'example_copy.txt')

# Move a file
shutil.move('example_copy.txt', 'new_directory/example_copy.txt')
```

- Copy a file, and paste it into a new directory.
- Delete an existing file, directory.

# Copying a File

```
import shutil

try:
    shutil.copy("source_file.txt", "destination_folder/")
except FileNotFoundError:
    print("Source file not found.")
except PermissionError:
    print("Permission denied.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

- Copy a file to a new destination.
- Handle possible errors/exceptions.

# Moving a File

```
import shutil

# Copy a file
shutil.copy('example.txt', 'example_copy.txt')

# Move a file
shutil.move('example_copy.txt', 'new_directory/example_copy.txt')
```

```
# Delete a file
os.remove('example_copy.txt')

# Delete an empty directory
os.rmdir('new_directory')
```

- Copy a file, and paste it into a new directory.
- Delete the original file.

# Moving a File

```
import shutil

try:
    shutil.move( "source_file.txt" ,
    "new_location/source_file.txt" )
except FileNotFoundError:
    print("Source file not found." )
except PermissionError:
    print("Permission denied." )
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

- Move file to a new location.
- Handle the possible errors.

# Renaming a File

```
import os

try:
    os.rename("old_name.txt", "new_name.txt")
except FileNotFoundError:
    print("File not found. ")
except PermissionError:
    print("Permission denied. ")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

- Replace the filename with a new one.
- Handle the possible errors.

# Deleting a File

```
import os

try:
    os.remove("file_to_delete.txt")
except FileNotFoundError:
    print("File not found.")
except PermissionError:
    print("Permission denied.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

- Change the filename.
- Handle the possible errors.

# Directories

# Creating a Directory

```
import os

directory_path = '/path/to/new_directory'
os.makedirs(directory_path)
```

```
import os

directory_path = '/path/to/existing_directory'
if os.path.exists(directory_path):
    print('Directory exists!')
```

- The **os** and **shutil** modules provide a powerful set of tools for working with files and directories in a platform-independent way
- Use `os.makedirs()` to create a directory and its parent directories if they don't exist
- Use `os.path.exists()` to check if a directory exists

# — Listing all Files

```
import os

directory_path = '/path/to/directory'
files = os.listdir(directory_path)
print(files)
```

```
import os

new_directory = '/path/to/new_directory'
os.chdir(new_directory)
```

- Use `os.listdir()` to get a list of files in a directory
- Use `os.chdir()` to change the current working directory

# Removing a Directory

```
import os
import shutil

directory_path = '/path/to/directory_to_remove'

# Remove an empty directory
os.rmdir(directory_path)

# Remove a directory and its contents
shutil.rmtree(directory_path)
```

Be careful with this!

- Use `os.rmdir()` to remove an empty directory
- For removing a directory and its contents, use `shutil.rmtree()`

# Renaming a Directory

```
import os

old_name = '/path/to/old_directory'
new_name = '/path/to/new_directory'

os.rename(old_name, new_name)
```

- Use `os.rename()` to rename a directory
- Try to handle possible errors

# — Directory: Example

```
import os
# Get the current working directory
current_dir = os.getcwd()
print(f'Current Directory: {current_dir}')
# Join paths
file_path = os.path.join(current_dir, 'example.txt')
# Check if a file or directory exists
if os.path.exists(file_path):
    print('File exists!')
# Create a directory
new_dir = os.path.join(current_dir, 'new_directory')
os.mkdir(new_dir)
# List files in a directory
files_in_dir = os.listdir(current_dir)
print(f'Files in Directory: {files_in_dir}')
```

- Get current working directory `getcwd()`
- Check if the file or directory has been available

# Search for Files

```
import os

def search_files(directory, extension):
    found_files = []
    for root, dirs, files in os.walk(directory):
        for file in files:
            if file.endswith(extension):
                file_path = os.path.join(root, file)
                found_files.append(file_path)
    return found_files

# Example: Search for all ".txt" files in the "my_directory" and its subdirectories
directory_to_search = 'my_directory'
target_extension = '.pdf'
result = search_files(directory_to_search, target_extension)
if result:
    print(f"Found {len(result)} files:")
    for file_path in result:
        print(file_path)
else:
    print(f"No {target_extension} files found.")
```

- Traverse through all the subdirectories, and look for the corresponding files

# — Connect with Google Drive

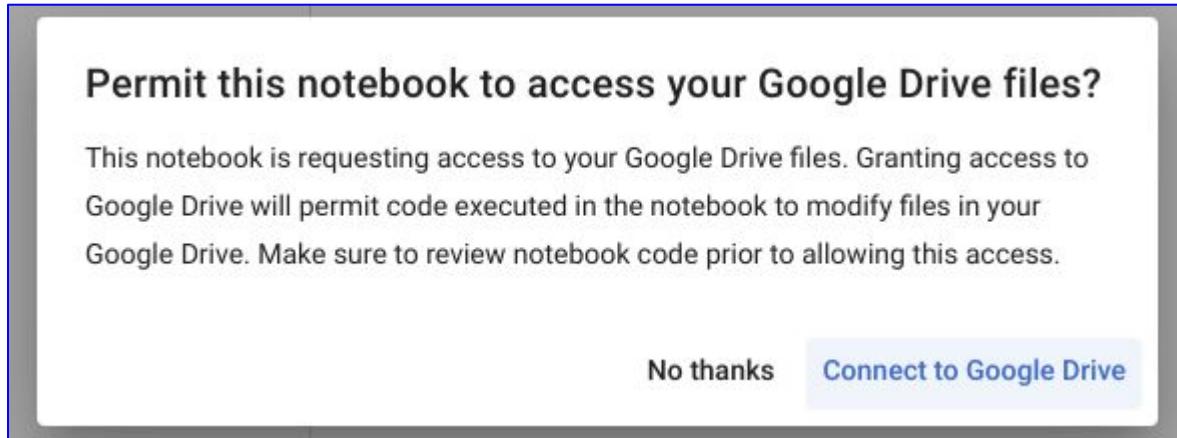
```
from google.colab import drive
drive.mount('/content/drive/', force_remount=False)
DATA_PATH = '/content/drive/My Drive/Colab Notebooks/Data Science/'
from os.path import join
data_file = join(DATA_PATH, 'data.csv')
input_data = pd.read_csv(data_file, header = None)
```



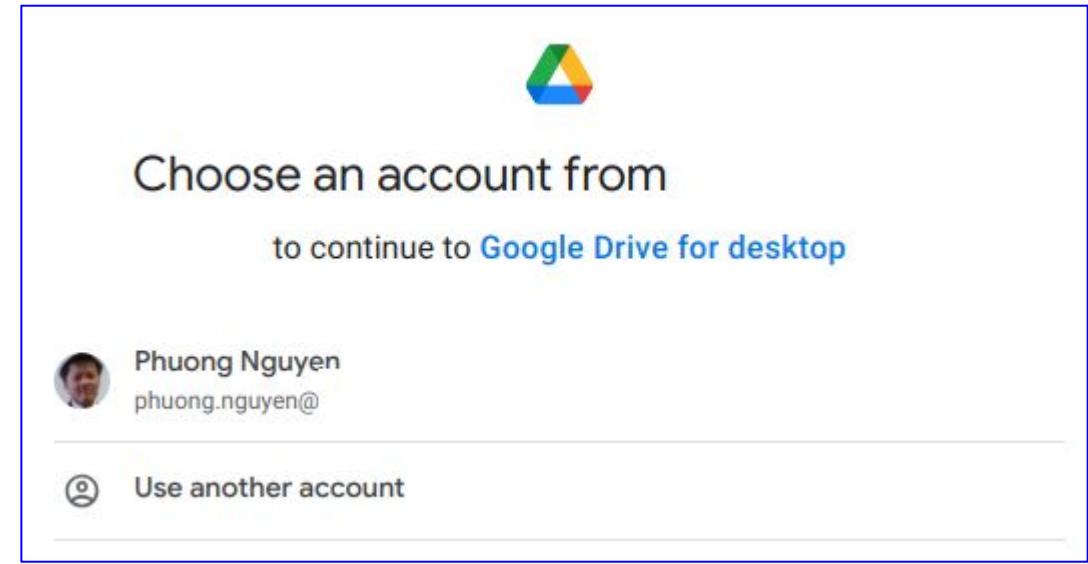
Google Drive

- Use `google.colab` to access data stored in Google Drive

# — Connect with Google Drive (2)



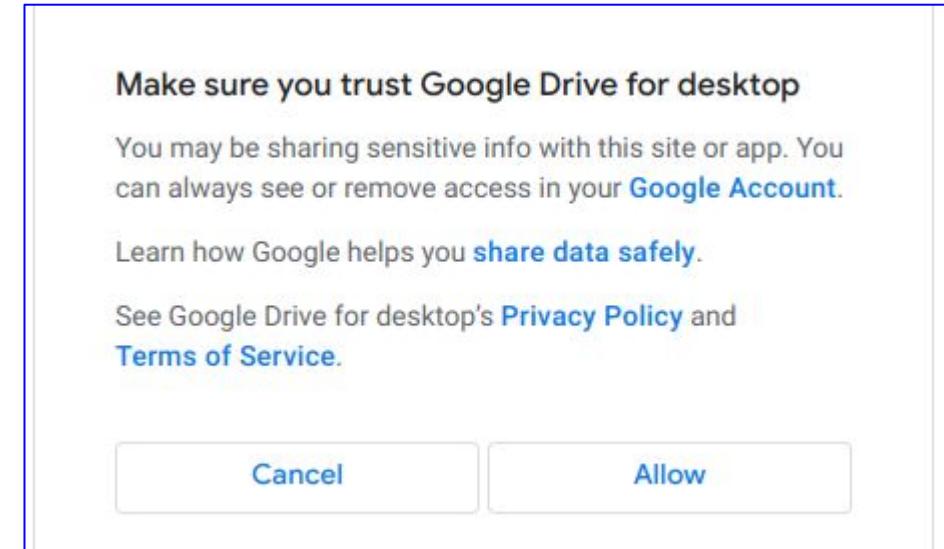
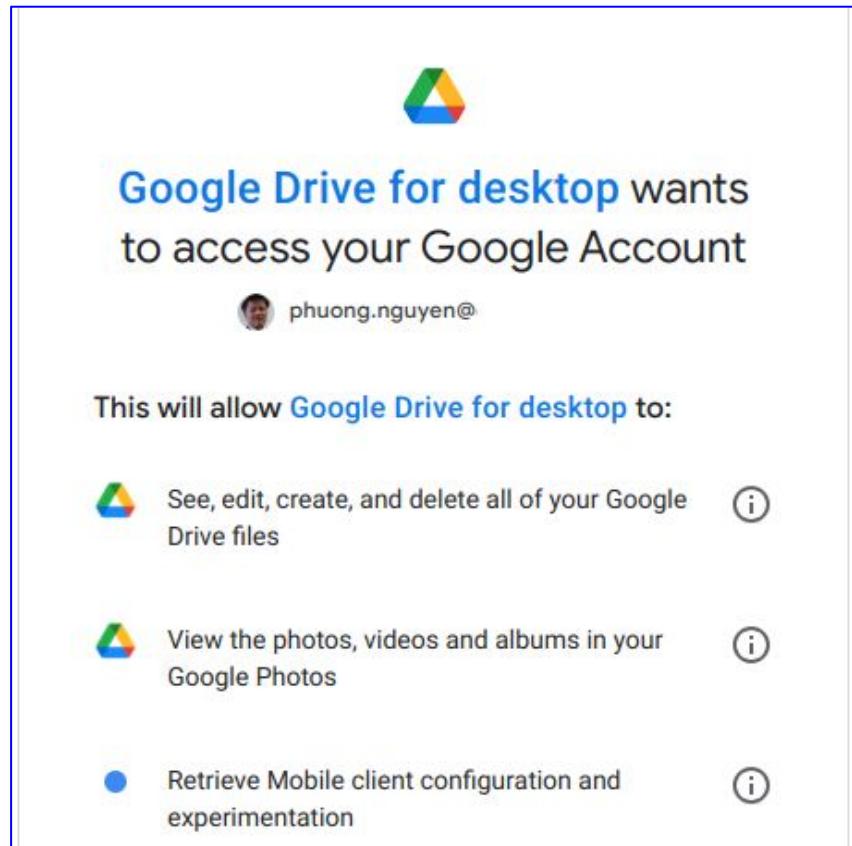
1



2

- Use google.colab to access data stored in Google Drive

# — Connect with Google Drive (3)



## ■ Use google.colab to access data stored in Google Drive

# Exercises

# File Operations

- Exercise 1: File Copy. Write a Python program that copies the content of one text file to another. Allow the user to input the names of the source and destination files.
- Exercise 2: File Move. Create a program that moves a file from one directory to another. Prompt the user to enter the file name, source directory, and destination directory
- Exercise 3: File Rename. Write a script that renames a file. Allow the user to input the current file name and the new desired name

# File Operations

- Exercise 4: File Deletion. Develop a Python program that deletes a specified file. Ask the user to input the name of the file to be deleted
- Exercise 5: File Information. Create a script that displays information about a file. Allow the user to input the file name, and then output details such as file size, creation time, and modification time
- Exercise 6: Directory Listing. Write a program that lists all files in a specified directory. Allow the user to input the directory path

# Solutions

# Exercise 1

```
def copy_file(source_file, destination_file):
    with open(source_file, 'r') as source:
        content = source.read()
        with open(destination_file, 'w') as destination:
            destination.write(content)

# Example Usage
copy_file('source.txt', 'destination.txt')
```

- Open the source file, read its content
- Open the destination file, write the content

# Exercise 2: Move File

```
import os

def move_file(source_file, destination_directory):
    if not os.path.exists(destination_directory):
        os.makedirs(destination_directory)

    destination_path = os.path.join(destination_directory, os.path.basename(source_file))

    # Copy content from source to destination
    with open(source_file, 'r') as source:
        content = source.read()
        with open(destination_path, 'w') as destination:
            destination.write(content)

    # Delete the source file
    os.remove(source_file)

# Example Usage
move_file('source.txt', 'new_directory')
```

- Open the source file, read its content
- Open the destination file, write the content

# Data Analysis Examples with Pandas, Matplotlib, and Seaborn

Phuong T. Nguyen



Dipartimento di Ingegneria e Scienze  
dell'Informazione e Matematica

Università degli Studi dell'Aquila

# Assignment



- Reading: Loaded a large CSV file with multiple rows and columns.
- Filtering: Selected rows and columns based on conditions.
- Cleaning: Handled missing values, removed duplicates, and renamed columns.
- Transforming: Added new columns like Experience.
- Analyzing: Performed statistical analysis and counted employees by department.
- Exporting: Saved the cleaned and transformed data to new CSV and Excel files.

A	B	C	D	E	F	G
EmployeeID	Name	Age	Department	Salary	JoiningDate	PerformanceScore
1	Employee_1	58	Marketing	96199	2015-01-01	3
2	Employee_2	48	Finance	64766	2015-02-01	3
3	Employee_3	34	Finance	91087	2015-03-01	3
4	Employee_4	27	IT	98840	2015-04-01	2
5	Employee_5	40	Marketing	84384	2015-05-01	
6	Employee_6		HR	81005	2015-06-01	1
7	Employee_7	38	Finance	76576	2015-07-01	4
8	Employee_8	42	Marketing	69353	2015-08-01	1
9	Employee_9	30	Marketing	92003	2015-09-01	
10	Employee_10	30	IT	82733	2015-10-01	4
11	Employee_11	43	Finance	95318	2015-11-01	
12	Employee_12	55	Finance	53664	2015-12-01	3
13	Employee_13	59	HR	97172	2016-01-01	4
14	Employee_14	43	Finance	56736	2016-02-01	3
15	Employee_15		HR	30854	2016-03-01	1
16	Employee_16	41	Finance	68623	2016-04-01	1

- Comma-Separated Values: A plain text file format that stores tabular data (numbers and text) in plain text form, where each line of the file is a data record, and fields within each record are separated by commas
- Delimiter: While the default delimiter is a comma, other delimiters like semicolons (;) or tabs (\t) can also be used, depending on the application or region

# Create a CSV file

```
# Create a sample DataFrame
np.random.seed(42)

data = {
    "EmployeeID": range(1, 101),
    "Name": [f"Employee_{i}" for i in range(1, 101)],
    "Age": np.random.randint(20, 60, 100),
    "Department": np.random.choice(["HR", "IT", "Finance", "Marketing"], 100),
    "Salary": np.random.randint(30000, 100000, 100),
    "JoiningDate": pd.date_range(start="2015-01-01", periods=100, freq="MS").to_list(),
    "PerformanceScore": np.random.choice([1, 2, 3, 4, np.nan], 100),
}

df = pd.DataFrame(data)
```

- Using random functions to create 100 data points (records).

# Data Selection

```
# List the employees who have a high salary
high_salary = df[df["Salary"] > 70000]
print("Employees with Salary > 70,000:")
print(high_salary)
```

```
# Select the employees who are in between 30 and 40 years old,
# and work for the IT Department.
age_and_department = df[ (df["Age"] >= 30) & (df["Age"] <= 40) &
(df["Department"] == "IT")]
print("Employees aged 30-40 in IT Department:")
print(age_and_department)
```

- Using Pandas to select data by specifying the criteria.

# Cleaning data

```
# Fill rows with missing age
df["Age"] = df["Age"].fillna(df["Age"].mean())
print("After filling missing values in 'Age':")
print(df.head())
```

```
# Drop all rows without any numbers for salary
df_cleaned_salary = df.dropna(subset=["Salary"])
print("After dropping rows with missing Salary:")
print(df_cleaned_salary.head())
```

```
# Remove duplicates
df_no_duplicates = df.drop_duplicates()
print("After removing duplicates:")
print(df_no_duplicates.shape)
```

- Using Pandas to select data by specifying the criteria.

# — Change columns

```
# Add a new column
from datetime import datetime
df["Experience"] = datetime.now().year -
pd.to_datetime(df["JoiningDate"]).dt.year
print("After adding 'Experience' column:")
print(df.head())
```

```
# Rename a column
df_renamed = df.rename(columns={"PerformanceScore": "Performance_Score"})
print("After renaming columns:")
print(df_renamed.head())
```

- Adding a new column.
- Renaming an existing column.

# Save data

```
# Save the data frame to an external file
output_file = join(DATA_PATH, 'cleaned_employee_data.csv')
df_cleaned_salary.to_csv(output_file, index=False)
print("Cleaned data saved to 'cleaned_employee_data.csv'.")
```

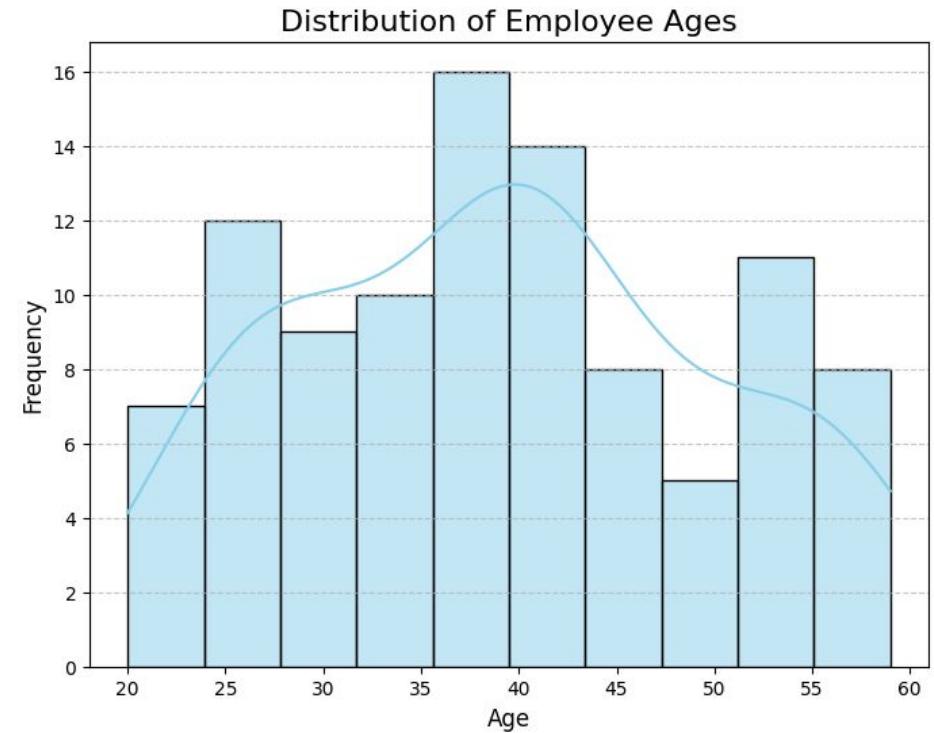
```
excel_file = join(DATA_PATH, 'transformed_employee_data.xlsx')
df.to_excel(excel_file, index=False)
print("Transformed data saved to
'transformed_employee_data.xlsx'.")
```

- A whole data frame can be exported to a file.
- File format could either be CSV or XLS.

# Visualizing the data

```
# Histogram: Distribution of Ages
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

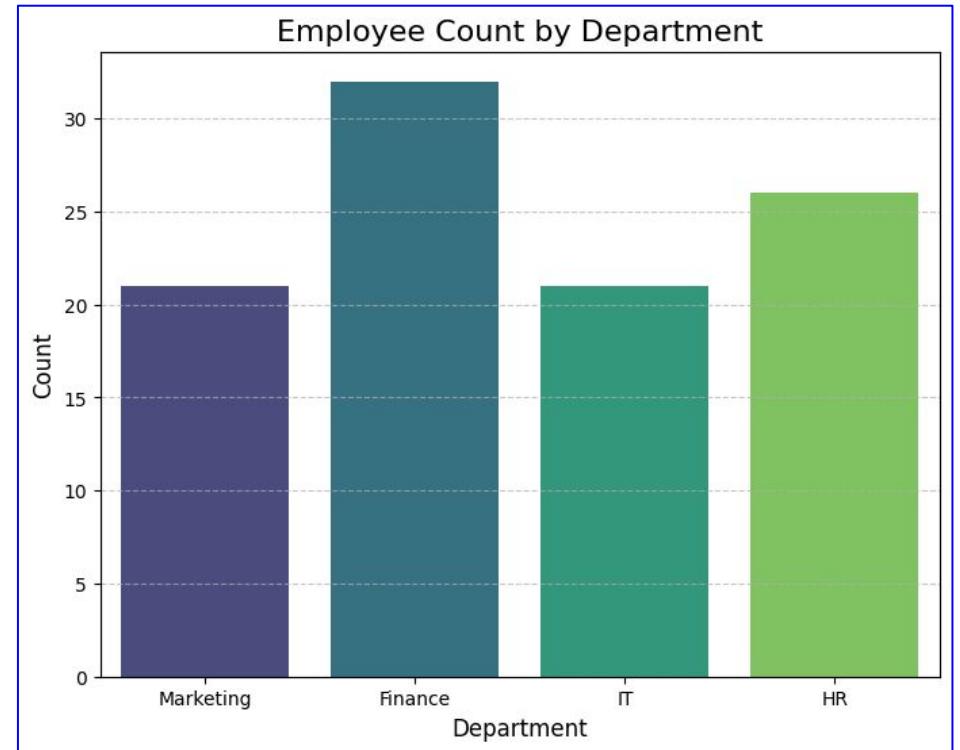
plt.figure(figsize=(8, 6))
sns.histplot(df["Age"], kde=True, bins=10, color="skyblue")
plt.title("Distribution of Employee Ages", fontsize=16)
plt.xlabel("Age", fontsize=12)
plt.ylabel("Frequency", fontsize=12)
plt.grid(axis="y", linestyle="--", alpha=0.7)
plt.show()
```



- Displaying the histogram of ages of the employees.

# Visualizing the data

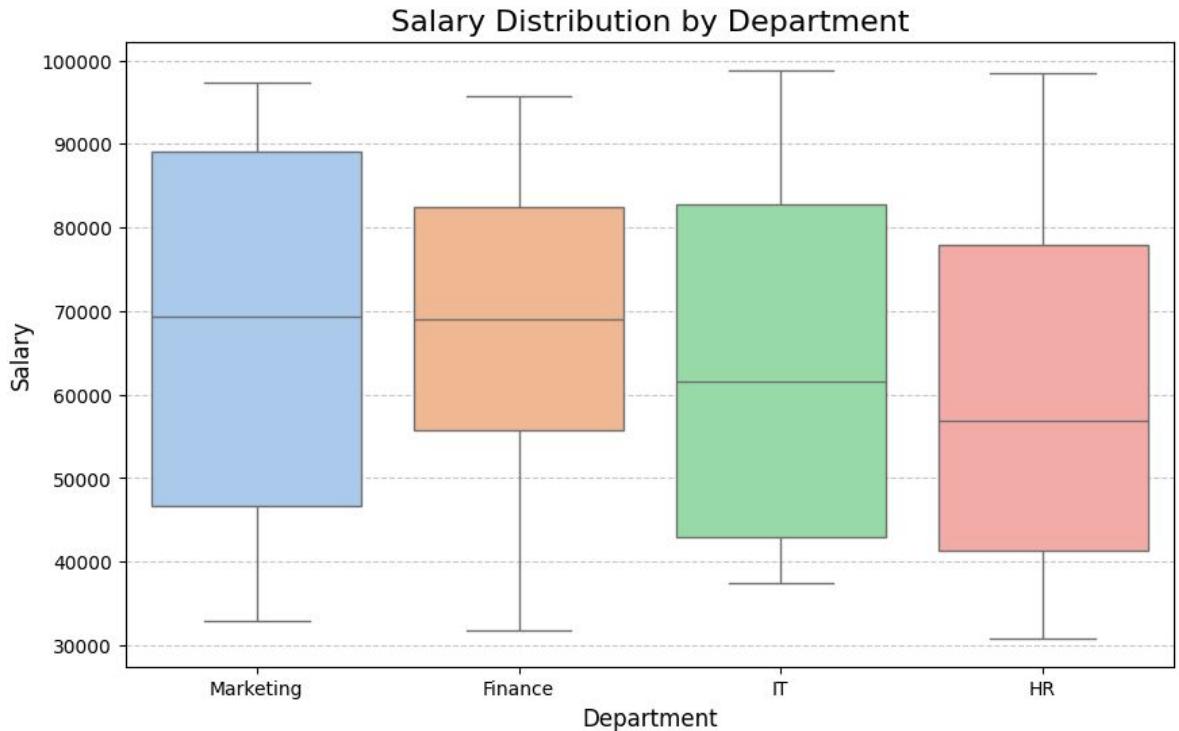
```
#Bar Chart: Count of Employees by Department
plt.figure(figsize=(8, 6))
sns.countplot(data=df, x="Department",
palette="viridis")
plt.title("Employee Count by Department", fontsize=16)
plt.xlabel("Department", fontsize=12)
plt.ylabel("Count", fontsize=12)
plt.grid(axis="y", linestyle="--", alpha=0.7)
plt.show()
```



- Displaying the number of employees by each department.

# Visualizing the data

```
#Box Plot: Salary Distribution by Department
plt.figure(figsize=(10, 6))
sns.boxplot(data=df, x="Department", y="Salary",
palette="pastel")
plt.title("Salary Distribution by Department",
fontsize=16)
plt.xlabel("Department", fontsize=12)
plt.ylabel("Salary", fontsize=12)
plt.grid(axis="y", linestyle="--", alpha=0.7)
plt.show()
```



- Displaying the distribution of salary following the departments.

# Plotting salary by each department

```
departments = salary_trends["Department"].unique()
fig, axes = plt.subplots(len(departments), 1, figsize=(10, 12), sharex=True)

for i, dept in enumerate(departments):
    dept_data = salary_trends[salary_trends["Department"] == dept]
    sns.lineplot(data=dept_data, x="JoiningDate", y="Salary", ax=axes[i],
    marker="o", color="b")
    axes[i].set_title(f"Salary Trend for {dept}", fontsize=14)
    axes[i].set_xlabel("")
    axes[i].set_ylabel("Salary")
    axes[i].grid(axis="both", linestyle="--", alpha=0.7)

plt.tight_layout()
plt.show()
```

- Displaying the distribution of salary following the departments.