PYTHON

Variables
- Dictionary
- Tuple
- list
- Numbers
  - int → $x=1$
  - float → $x=2.8$
  - Complex → $x=3+2j$
- strings → a = "Hello World"

String Input →
```
print("Enter your name:")
x = input()
print("Hello, " + x)
```

Important Packages:

NumPy – The fundamental package for scientific computing with python.

SciPy – a free and open-source Python library used for scientific computing and technical computing. SciPy contains modules for optimization, linear algebra, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering.

Matplotlib – a Python 2D plotting library

⋮

```
import numpy as np
x = 3
y = np.sin(x)
print(y)
```

```
import math as mt
import numpy as np
x = 3
y = mt.sin(x)
print(y)
y = np.sin(x)
print(y)
```

Plotting in Python

import matplotlib.pyplot as plt ⇒ importing the library (whole)

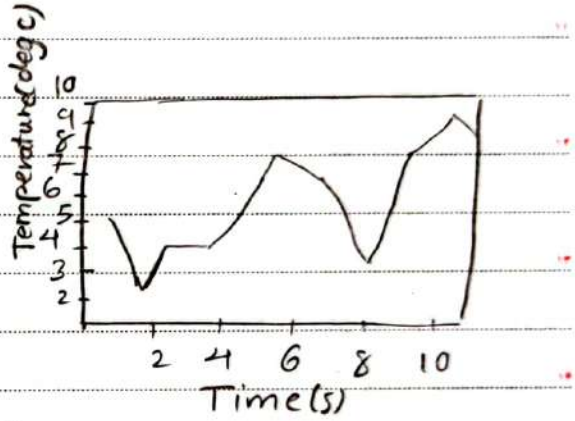
functions: plot(), title(), xlabel(), ylabel()

axis(), grid(), subplot(), legend(), show()

Ex: from matplotlib.pyplot import *

x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

y = [5, 2, 4, 4, 8, 7, 4, 8, 10, 9]



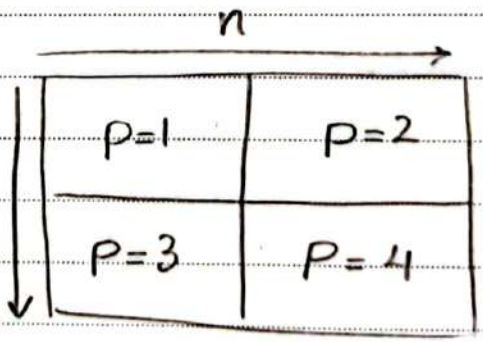

```
plot(x, y)
xlabel('Time (s)')
ylabel('Temperature (deg C)')
show()
```

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

xstart = 0
xstop = 2 * np.pi
increment = 0.1

x = np.arange(xstart, xstop, increment)

y = np.sin(x)
```

```
plt.plot(x,y)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Subplot(m,n,p) → partitions the figure window into an m-by-n matrix of small subplots

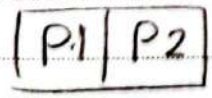
Ex:
```
import matplotlib.pyplot as plt
import numpy as np

xstart=0
xstop=2*np.pi
increment=0.1

x=np.arange(xstart,xstop,increment)

y=np.sin(x)

z=np.cos(x)

plt.subplot(2,1,1)   # row, column, plot1
plt.plot(x,y,'g')
plt.show()

plt.subplot(2,1,2)   # row, column, plot2
plt.plot(x,z,'r')
plt.show()
```



m = number of rows in the grid

n = number of columns in the grid

P = index of the subplot in the grid (starting from 1, left to right, top to bottom)



If... Else

Ex: a=5
b=8

```
if a>b:
    print ("a is greater than b")

if b>a:
    print ("b is greater than a")

if a==b:
    print ("a is equal to b")
```

```
if expression:
    statement(s)
else:
    statement(s)
```

```
if expression1:
    statement(s)
elif expression2:
    statement(s)
elif expression3:
    statement(s)
else:
    statement(s)
```

elif not elseif !

Comparision Operators

==	equal
!=	not equal
<>	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

---

assert → use to check conditions during development and testing. where failing the condition should halt the program with an error

Ex: x=5
assert x>3, "x should be greater than 3"
This will pass

~~Arrays~~ Python Arithmetic Operators

+ Addition → a+b - subtraction → a-b

* Multiplication → a*b / Division → b/a

% Modulus → b%a ** Exponent → a**b توان

// Floor Division → 9//2 = 4 and 9.0//2.0 = 4.0

Python Assignment Operators

+= Add And (C+=a) = (C=C+a)

-= Subtrad And %= Modulus And

*= Multiply And **= Exponent And

/= Divide And //= Floor Division And

Python Bitwise Operators

& Binary AND

| Binary OR

^ Binary XOR

~ Binary ones Complement

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

Python Logical Operators

and Logical AND: (a and b) is True

or Logical OR: (a or b) is True

not Logical NOT: Not (a and b) is false

Python Membership Operators

in x in y not in x not in y

Strings

Ex: str = 'Hello World!'

Hello_World!
0 1 2 3 4 5 6 7 8 9 10 11
-12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

print str[0] → H
print str[2:5] → llo
print str[2:] → llo World!
print str*2 → Hello World! Hello World!
print str + "Test" → Hello World! Test

[a : b]
excluded
included

str[::-1] # Reverse the string

[Lists] → ordered collection that can hold a variety of data type, including numbers, strings, other lists, and even custom objects. List are mutable, meaning that you can change their contents (add, remove, or modify elements) after creation.

numbers = [1, 2, 3, 4, 5]
0 1 2 3 4

mixed = [1, "hello", 3.14]
0 1 2
-3 -2 -1

fruits = ["apple", "banana", "cherry"]
0 1 2
-3 -2 -1

۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵ ۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹ ۳۰ ۳۱

changing → fruit[1] = "blueberry"

Adding → append(x): Adds element to the end of list

insert(i,x): Adds at a specific position

Ex: fruits.insert(1, "orange")

fruits.append("pear")

Removing → remove(element) → Removes the first occurrence of the value.

pop(number) → Removes at a given index

clear() → Removes all the elements

Operations

len() → length (number of elements)

in → checks if an element exists in the list

slicing → numbers[1:5] → 2, 3, 4, 5

numbers[-3:] → 3, 4, 5

Methods: append(x), insert(i,x), remove(x), pop(i)

Sort() → Sorts the list in ascending order

reverse() → Reverses the elements of the list

* Sorted can be used for strings too.

```
s = "hello"
sorted_s = sorted(s)
print(sorted_s)
# output: ['e', 'h', 'l', 'l', 'o']
```

```
# sorted_s = ''.join(sorted(s))   # output : 'ehllo'
```

۱۴۰۰/۲/۱۰

Ex: numbers = [4, 2, 7, 1, 9]

```
numbers.sort()
print(numbers)      # [1, 2, 4, 7, 9]

numbers.reverse()
print(numbers)      # [9, 7, 4, 2, 1]
```

Sort() and Sorted()

- Sort(): sorts the list in place
- Sorted(): returns a new, sorted list without modifying the original

Ex:
```
sorted_numbers = sorted(numbers)
print(sorted_numbers)    # [1, 2, 4, 7, 9]
print(numbers)           # [4, 2, 7, 1, 9]
```

Sorting in reverse ⇒ use (reverse = True)

Ex:
```
numbers.sort(reverse = True)
```

```
sorted_numbers = sorted(numbers, reverse = True)
```

Sorting Based on a Key

Ex:
```
fruits = ["apple", "banana", "cherry", "blueberry"]

sorted_fruits = sorted(fruits, key = len)

print(sorted_fruits)    # ['apple', 'banana', 'cherry', "blueberry"]
```

(Tuples) → ordered, immutable collection in python ۱۴۰۰/۲/۱۱
faster than lists, Cannot change elements
used for grouping related pieces of data together

Ex: numbers = (1, 2, 3, 4, 5) # numbers [2] = 3

person = ("John", 30, True) # person [-1] = True

Single_element = (42,) ← لازم

Tuple Operations

- Combined = tuple1 + tuple2
- repeated = tuple * 2
- slicing → numbers [1:2]

Tuple Methods

- count(x): Returns the number of times x appears in the tuple.
- index(x): Returns the index of the first occurrence of x

Packing: When you assign values to a tuple

person = ("John", 30, "New York")

Unpacking: unpack a tuple into individual variables

name, age, city = person

۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵ ۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹ ۳۰ ۳۱

۱۴۰۰/۲/۱۲

Ex: numbers = (1, 2, 3, 4, 5)

a, b, *rest = numbers

a = 1 , b = 2

rest = [3, 4, 5]

↳ It's a list

{Dictionaries} → unordered, mutable Collection of key-value pairs indexed by Keys, each key must be unique and immutable

Ex: person = {

"name": "John",

"age": 30,

"city": "New York",

"is_student": False,

}

empty_dict = { }

* if access a key that doesn't exist ⇒ Key Error ⇒ use get()

```
print(person["name"])      # John
print(person["age"])       # 30
```

```
print(person.get("name"))      # John
print(Person.get("salary"))    # None (Key doesn't exist)
print(Person.get("salary", 0)) # 0 (default value)
```

Adding or Updating Key_Value Pairs

Ex: person ["salary"] = 5000

person ["age"] = 31

Removing Key_value Pairs

1. pop(key) : Removes the Key and returns its value

2. del : Deletes the Key-value pair.

3. popitem() : Removes and returns the last inserted Key_value pair

4. clear(): Removes all key_value pairs.

Ex: age=person.pop("age")
print(age) # 31

del person["city"]

last_item = person.popitem()
print(last_item) # ('Salary', 5000)

person.clear()
print(person) # {}

Dictionary Operations

1. Checking if a key exists:

```
person = {"name": "John", "age": 30, "city": "New York"}

Ex: print("name" in person)      # True
    print("salary" in person)    # False
```

2. Getting the length of the dictionary:

```
print(len(person))      # 3
```

Looping Through a Dictionary

1. Keys: Loop through Keys using keys()
2. Values: Loop through values using values()
3. key_value Pairs: Loop through key_value pairs using items()

```
Ex: for key in person.keys():
        print(key)

    for value in person.values():
        print(value)

    for key_value in person.items():
        print(key, value)
```

Dictionary Methods:

1. get(key, default) : Returns the value for the key

2. update(other_dict) : Updates the dictionary with key_value pairs from another dictionary or Herable.

3. pop(key)

4. popitem()

9. zip() : pairs elements directly from 2 lists
did (zip (keys, values))

5. clear()

6. keys() : Returns a view object of the keys.

7. values() : Returns a view object of the values.

8. items() : Returns a view object of key_value pairs.

Nested Dictionaries

```
Ex: students = {
    "student1": {"name": "John", "age": 22},
    "student2": {"name": "Jane", "age": 21},
    }
print(students["student1"]["name"])        # John
```

Try - Except ⇒ for handling exceptions.

```
try:
    #Code that might raise an exception
    risky_code()
except SomeExceptionType:
    #Code to run if an exception occurs
    handle_exception()
```

Ex:
```
try:
    result = 10/0        ⇒ ZeroDivisionError
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")

                #output: Error: Division by zero ...
```

Ex:
```
try:
    num = int(input("Enter a number:"))
    result = 10/num
except ValueError
    print("Invalid input! Please enter a valid integer.")
except ZeroDivisionError:
    print("Error : Division by zero is not allowed.")
else:
    print(f"Result is: {result}")
```

else: executed if the try block does not raise any exceptions.
(for running some code only when no exception occurs.)

finally: always executed. (useful for releasing resources or performing cleanup actions, such as closing a file)

Full try-except-else-finally Example :

```
try:
    num = int(input("Enter a number: "))
    result = 10/num
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter a valid integer.")
else:
    print(f"Result: {result}")
finally:
    print("This will always run, whether there's an error or not.")
```

Set ⇒ unordered, mutable collection of unique elements.

Ex: fruits = {"apple", "banana", "cherry"} ⇒ using { }

numbers = Set([1,2,3,4]) ⇒ using set() function

* Elements do not have a fixed position.
* Duplicate elements are automatically removed.

Common Set Operations :

add(), remove() or discard(), union() or |
intersection() or &
difference() or -

Ex:
```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1.union(set2)
print(union_set)        # {1, 2, 3, 4}
```

Feature	Set	List
Mutability	Mutable	Mutable
Ordering	Unordered	Ordered
Duplicates	No duplicates allowed	Duplicates allowed
Access by Index	Not allowed	Allowed
Use Cases	Unique elements, set operations	Ordered Collections, duplicate
Performance	Fast for membership checks	Slower for membership check
Adding/Removing	add(), remove(), unordered	append(), remove(), ordered

Conversion ⇒ list() and set()

Frozen Sets: the immutable version of a set.
useful for a dictionary key

```
frozen_fruits = frozenset(["apple", "banana", "cherry"])
```

* supports set operations like union, intersection, ... but returns a new frozenset.

Loops → For → iterate over a sequence
Loops → While → repeats as long as a given condition is True.

١٤٠٠/٢/١٩

For: for variable in iterable:
    block of code

Ex: fruits = ["apple", "banana", "cherry"]

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

range(): a function used in for loops to generate a sequence of numbers.

Ex:
```
for i in range(5):
    print(i)
```
(5 excluded → 0 to 4)
output: 0, 1, 2, 3, 4

```
for i in range(2, 11, 2):
    print(i)
```
(start, stop, step)
output: 2, 4, 6, 8, 10

```
for letter in "hello":
    print(letter)
```
h, e, l, l, o

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

for key, value in my_dict.items():
    print(key, ":", value)
```

* underscore (_): Indicates that the loop variable is not going to be used.

Ex:
```
for _ in range(5)
    Print("Hello")
```

While :

```
while Condition:
    Block of code
```

Ex:

```
Count=0
while Count<5:
    print(count)
    Count+=1
```

Output :
0
1
2
3
4

Infinite Loop:

```
while True:
    print("This is an infinite loop")
```

(press ctrl + C to stop the code)

Break:

Ex:

```
count=0
while count<10:
    if Count == 5:
        break
    print(Count)
    Count += 1
```

0
1
2
3
4

Break and Continue:

Ex:

```
for i in range(5):
    if i == 2:
        continue  -> skip the iteration
    elif i == 3
        break  -> Exit the loop
    Print(i)
```

0
1

else with loops: executed when the loop finishes normally (no break)

Ex: for i in range(5):
    print(i)
else:
    print("Loop completed!")

(also used in while loops)

Pass: does nothing and acts as a placeholder.
used during development when you need to outline code that will be written later.

```
def my_function():
    pass        # This function does nothing (yet)

for i in range(10):
    pass        # Loop does nothing (yet)
```

enumerate(): a built-in function that adds a counter to an iterable (like a list, tuple, or string) and returns it as an enumerate object

enumerate(iterable, start=0) (start=1) → starting from 1

Ex: fruits = ['apple', 'banana', 'cherry"]

```
for index, fruit in enumerate(fruits):
    print(f"Index: {index}, Fruit: {fruit}")
```

Functions

Defining ⇒ def function_name(Parameters):

Code block (indentation required)
Optional return statement

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

Calling ⇒ greet_person("Alice") # Hello, Alice!

Ex:
```
def multiply(a,b,c):
    return a*b*c
```

Default Parameters:

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

greet("Alice")    # Hello, Alice!
greet()           # Hello, Guest!
```

Keyword Arguments:

```
def describe_pet(animal_type, pet_name):
    print(f"I have a {animal_type} named {pet_name}.")

describe_pet(animal_type = "dog", pet_name = "Buddy")

#output: I have a dog named Buddy.
```

*args and **kwargs

*args: Allows you to pass a variable number of positional arguments.

*kwargs: Allows you to pass a variable number of keyword arguments.

Ex:
```
def add_numbers(*args):
    total=0
    for number in args:
        total += number
    return total

print(add_numbers(1,2,3))      # 6
print(add_numbers(4,5,6,7))    # 22
```

* *args allows the function to take any number of arguments.

Ex:
```
def print_details(**kwargs):
    for key,value in kwargs.items():
        print(f"{key}: {value}")

print_details(name="Alice",age=25, city="New York")

# output =  name: Alice
            age: 25
            city: New York
```

**kwargs allows the function to accept any number of keyword arguments and print them.

۱۴۰۰/۲/۲۴

Scope of Variables

Local: Created inside a function, Only accessable within it.

Global: Created outside of functions, Can be accessed from anywhere

```
Ex: x = 10      # Global variable

    def my_funtion():
        x = 5      # Local variable
        print("Inside function:", x)

    my_function()                  # output: Inside function: 5
    print("Outside function:", x)  # output: outside function: 10
```

Anonymous Function (lambda) ⇒ for small and simple functions

```
lambda parameters : expression

Ex: add = lambda a, b: a + b          # output: 8
    print(add(3, 5))
```

Recursive Functions ⇒ Calls itself to solve smaller instances of the same problem

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵ ۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹ ۳۰ ۳۱

تعطیل به مناسبت عید سعید فطر - لغو امتیاز تنباکو به فتوای آیت الله میرزا حسن شیرازی (۱۲۷۰ ه.ش)

```
print(factorial(5))          # output: 120
```

Working Directory (CWD)

1) Checking the current Working Directory ⇒ getcwd()

```
import os

cwd = os.getcwd()
print("current working Directory:", cwd)

#output: Current Working Directory: /Users/username/project
```

2) Changing the Working Directory ⇒ chdir()

```
import os

os.chdir("/path/to/new/directory")
```

3) why CWD is important

File operations: When opening, reading, or saving files, Python looks for the file in the cwd unless a full path is provided.

Relative paths: If you use relative paths (paths that don't start with / or C:), Python interprets them relative to the current working directory.

Ex (relative path): with open ("data.txt", "r") as file:
    Content = file.read()

* Python looks for the file data.txt in the cwd.

4) Getting the List of files in the W.D ⇒ os.listdir()

Ex: import os

files = os.listdir()
print ("Files and directories:", files)

5) Absolute vs. Relative Paths

Absolute: The full path from the root directory (e.g., /home/user/documents/file.txt or C:\Users\User\Documents\file.txt).

Relative: A path relative to the current working directory (e.g. file.txt or .../other_directory/file.txt).

6) Best Practices for Managing Working Directory

- use absolute paths for file operations if the files are stored in different locations or if you're unsure of the cwd.
- use relative paths when working within the same project folder, and you're sure the working directory will remain constant.
- Check the current working directory using os.getcwd() if you're encountering file_not_found errors, as Python might be looking for files in the wrong directory.

7) Using Pathlib for Working Directory

pathlib module offers a more modern and object_oriented way of working with paths and directories. It makes handling paths more intuitive compared to os

Ex: from pathlib import Path

```
cwd = Path.cwd()
# checking cwd   print("Current working Directory:", cwd)
# changing      Path("/path/to/new/directory").mkdir(parents=True, exist_ok=True)
```

* Module: a file containing Python code that you can import into your script to use its functions, classes, or variables. (A Single Python file (.py))

Built_in modules: math, os, sys, pathlib, ...

Importing Modules:

import module_name: Import the whole module

from module_name import function_name: Import specific func

import module_name as alis: Import a module and give it a short alias

Ex:

a file called math_utils.py

```
def add(a,b):
    return a+b
def subtract(a,b)
    return a-b
```

```
import math_utils

result = math_utils.add(5,3)
print(result)
```

* Package: is a collection of related modules organized in a directory, and it usually includes an _init_.py file

۱۴۰۰/۲/۲۸

File Operations :

1) Opening a file ⇒ open() ⇒ read, write, or append

```
file = open ('filename', 'mode')
```

Common Modess

'r': Open a file for reading (default)
'w': Open a file for writing. Overwrites the file if it already exists or creates a new file.
'a': Open a file for appending. New data is written at the end of the file.
'x': Open a file for exclusive creation. Fails if the file already exists.
'b': Binary mode (used for non-text files like images)

't': Text mode (default)

'+': Open for both reading and writing.
(doesn't ensure the file is created if it doesn't exist.)

2) Reading a File ⇒ file is opened in reading mode 'r'

Ex:
```
file = open ('example.txt', 'r')
Content = file.read()
print (Content)
file.close()        # Always close after using a file
```

* file.read(): reads the entire file as a single string.

'w+', 'a+'
↳ append and read
↓
write and read

١٤٠٠/٢/٢٩

Ex (reading line by line)

```
file=open('example.txt','r')
for line in file:
    print(line.strip())    # use strip() to remove newline characters
file.close()
```

* file.split() ⇒ splits the content by whitespace

other reading methods:

file.readline(): Reads one line at a time.
file.readlines(): Reads all lines and returns them as a list of strings.

3) Writing to a file ⇒ open in 'w' (write) or 'a' (append) mode

Ex:
```
file = open('example.txt', 'w')
file.write("This is a new line.\n")    #overwrites the file
file.write("Another line.")
file.close()
```

* file.write() writes the string to the file. You need to manually include '\n' for new lines.
* if the file doesn't exist, Python will create it.

Ex:
```
file= open('example.txt', 'a')
file.write("\nAppending a new line.")
file.close()
```

* If you try to read after writing without repositioning the pointer, you will read nothing because the pointer is at the end.

* file.read().split()
# splits the content into words.

۱۴۰۰/۲/۳۰

4) Automatically closing a File with (with)
The best practice for file handling in Python

```
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

5) Working with Binary Files
for binary files, such as images or audio files, you can use 'rb' or 'wb' modes.

Ex:
```
with open('image.jpg', 'rb') as file:
    binary_data = file.read()

with open('output.jpg', 'wb') as file:
    file.write(binary_data)
```

6) File Pointer (Cursor)
When reading or writing, Python maintains a "file pointer" or "cursor" to know where to read or write next. You can manually manipulate the file pointer using file.seek() and get the current position using file.tell().

Ex:
```
file = open('example.txt', 'r')
file.seek(5)    # Move cursor to the 6th character
print(file.read())    # Read from the 6th character onwards
file.close()
```

7) Checking if a File Exists ⇒ using os or pathlib

Ex os:

```
import os

if os.path.exists('example.txt'):
    print("File exists")
else:
    print("File does not exist")
```

Ex pathlib:

```
from pathlib import Path

file_path = Path('example.txt')
if file_path.is_file():
    print("File exists")
else:
    print("File does not exist")
```

8) File Operations (Rename, Delete, etc.)

```
import os
os.rename('old_name.txt', 'new_name.txt')

import os
os.remove('example.txt')
```

۱۴۰۰/۳/۱

Important Packages

1) Core utilities and general purpose packages
   1.1) os ⇒ interact with the operating system
   1.2) sys ⇒ access to system-specific parameters and funcs
   1.3) shutil ⇒ High-level operations on files and collections of files
   1.4) pathlib ⇒ Object-oriented filesystem paths
   1.5) logging ⇒ Provides a flexible framework for logging messages from Py

2) Data Science and Machine learning packages
   2.1) numpy ⇒ Core library for numerical computing and handling arrays
   2.2) pandas ⇒ Data manipulation and analysis tool
   2.3) matplotlib ⇒ Plotting and visualization library.
   2.4) seaborn ⇒ Statistical data visualization built on top of matplotlib
   2.5) scikit-learn ⇒ Machine learning library ⇒ simple & efficient tools
   2.6) tensorflow and keras ⇒ Libraries for building and training deep learning models
   2.7) statsmodels ⇒ Statistical modeling and hypothesis testing

3) Web developement packages
   3.1) flask ⇒ Lightweight web framework for building web applications.
   3.2) django ⇒ Full-featured web framework for building complex, large-scale web apps
   3.3) requests ⇒ Simplifies making HTTP requests.
   3.4) beautifulsoup4 ⇒ Web scraping for parsing HTML and XML documents.
   3.5) selenium ⇒ Automates browser interactions.

4) Automation and task management packages
   4.1) schedule 4.2) celery 4.3) pyautogui

۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵ ۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹ ۳۰ ۳۱

5) Networking and APIs

5.1) socket 5.2) http.client 5.3) websocket

6) Testing and Debugging

6.1) unittest ⇒ Python's built-in unit testing
6.2) pytest ⇒ Popular testing framework
6.3) pdb ⇒ Python's built-in debugger
6.4) Coverage ⇒ Tool to measure code coverage during testing

7) Data formats and File handling

7.1) json ⇒ Built in library for working with JSON data
7.2) csv ⇒ Built-in library for reading and writing CSV files.
7.3) openpyxl ⇒ Read and write Excel files
7.4) pytz ⇒ World timezone definitions and Conversion

8) Cryptography and Security

8.1) hashlib 8.2) cryptography

9) Miscellaneous and fun packages

9.1) random ⇒ Generates random numbers and select random elements.
9.2) pillow ⇒ Python Imaging Library (PIL) fork for image processing
9.3) pygame ⇒ Python library for making games. (2D)

۱۴۰۰/۳/۳

Virtual Environments ⇒ create isolated environments for projects.

1) Dependency Isolation: Each project has its own set of dependencies. For example, one project may require Django 2.2, while another project may require Django 3.0. with ve, each project can use its own version without conflict.

2) Clean Global Environment: Without ve, installing packages globally can lead to version conflicts and clutter.

3) Easy Package Management: ve allows you to manage packages for specific projects easily, without affecting others.

Create a VE ⇒ venv module

`python -m venv myenv`

(`-m venv` creates VE; `myenv` name of the VE)

Activate the VE

On Windows ⇒ `myenv\Scripts\activate`

On macOS/Linux ⇒ `source myenv/bin/activate`

Install Packages in the VE ⇒ using pip

`pip install numpy pandas matplotlib`

* installs numpy, pandas and matplotlib in the VE

۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵ ۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹ ۳۰ ۳۱

* pip list ⇒ lists installed packages

Deactivating the VE

deactivate

Virtual Environment Folder Structure

Once you create a virtual environment, the folder structure looks something like this:

```
myenv/
|
|— bin/ (or Scripts/ on Windows)
|  |— activate    # Activates script
|  |— python      # python interpreter specific to this env
|
|— include/
|  |— ...         # C headers for compiled packages
|
|— lib/ (or Lib/ on Windows)
|  |— site-packages/
|     |— ...      # Installed Python packages go here
```

Virtual Enviroment with virtualenv (Alternative to venv)

virtualenv is an older, widely-used tool for creating virtual envs, and it offers more features than venv.

to use virtualenv, you need to install it:

⇒ pip install virtualenv

virtualenv myenv

Managing VEs with pipenv ⇒ Combines pip and virtualenv

- Install pipenv:

⇒ pip install pipenv

- Create and Activate a VE:

⇒ pipenv shell

- Install a Package:

⇒ pipenv install numpy

Virtual Environment Best Practices

1) Use VE for every project

2) Store dependencies in a requirements.txt file: You can export the current environment's dependencies to a requirements.txt file:

⇒ pip freeze > requirements.txt

3) Recreate environments using requirements.txt: if you share your project or switch to another system, you can recreate the environment with:

⇒ pip install -r requirements.txt

4) Use .gitignore for virtual environments: if you're using version control like Git, you should exclude your VE folder from being tracked by adding it to your .gitignore file:

⇒ myenv/

String Module:

1. Constants: pre_defined character sets

- string.ascii_letters: A combination of both lowercase and uppercase letters. ('abcde... ABCDE... WXYZ)
- string.ascii_lowercase: Lower case letters
- string.ascii_uppercase: Uppercase letters
- string.digits: The digits '0123456789'
- string.punctuation: A string containing all punctuation characters. (!"#$%&'()*+,-./:;<=>?@[\]^_{|}~`)
- string.whitespace: A string of all whitespace characters. (' \t \n \r \x0b \x0c')
- string.printable: A combination of digits, letters, punctuations, and whitespace

* ↑ useful for string validation, formatting, or parsing.

Ex: import string

```
print(string.ascii_letters)  # 'abcd... WXYZ'
print(string.digits)         # '0123...'
```

2. Template Strings: Provide simpler string substitutions compared to Python's build_in str.format() or f_strings. You can define a template with placeholders (indicated by $), and then substitude them with actual values.

Ex: from string import Template

```
template = Template("Hello $name! Welcome to $place.")
result = template.substitute(name="Alice", place="Wonderland")
print(result)
```

(#output: "Hello Alice! Welcome to Wonderland")

* safe_substitute() works similarly, but it won't raise an error if any placeholders are missing.

3. string.capwords(): Capitalizes the first letter of each word in a string (similar to the str.title() method), but it also ensures that the rest of the letters in each word are lowercase.

```
import string

s = "hello world"
print(string.capwords(s))   # "Hello World"
```

4. String Formatter (in older Python versions):

The Formatter class in the string module can be used to create custom string formatting options, but in most cases, Python's f-strings or str.format() method are preferred for formatting tasks.

Practical Use Cases:

1. Validating Input: Check if a string contains only certain types of characters.

Ex:
```
import string

s = "abc123"

if all(char in string.ascii_letters + string.digits for char in s):
    print("String contains only letters and digits.")
else:
    print("Invalid characters found!")
```

* all() is used to check if all characters in the string s belongs to a set of valid characters (string.ascii_letters + string.digits).

* in check ensures that each character in the string s is either a letter or a digit.

۱۴۰۰/۳/۹

2. Generating Random Strings:

Ex: Password generator

```
import string
import random

def generate_password(length):
    characters = string.ascii_letters + string.digits + string.punctu
    password = ''.join(random.choice(characters) for _ in range(length))
    return password
```

* random.choice(): randomly select characters from the pool of valid chars.

* join(): a string method to combine elements of an iterable (like a list or tuple) into a single string, with a specified separator placed between the elements.

```
separator.join(iterable)
```

Ex1:
```
words = ['Hello', 'world', 'from', 'Python']
result = ' '.join(words)
print(result)                    # Hello world from Python
```

Ex2:
```
numbers = [1, 2, 3, 4, 5]
result = ','.join(map(str, numbers))
print(result)
                                 # 1, 2, 3, 4, 5
```

* map(): Convert each number to a string.

١٤٠٠/٣/١٠

3. Removing Punctuation:

Ex:

```
import string

s = "Hello, World! How's everything?"

s_no_punctuation = ''.join(char for char in s if char not in string.punctuation)
print(s_no_punctuation)
```

List Comprehension:

```
new_list = [expression for item in iterable if condition]
```

expression: The value that you want to include in the new list. This can be a function call, a calculation, or even just the item itself.

item: The variable that takes the value of each element in the iterable as the loop iterates.

iterable: The collection you are iterating over (e.g., a list, tuple, or string)

Condition: An optional filter that allows you to include only items that meet a certain criterion.

Ex:

```
squares = [x**2 for x in range(10)]
print(squares)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

۱۴۰۰/۳/۱۱

Ex 2: evens = [x for x in range(10) if x % 2 == 0]
print(evens)

[0, 2, 4, 6, 8]

Ex 3: words = ['hello', 'world', 'python']
uppercase_words = [word.upper() for word in words]
print(uppercase_words)

['HELLO', 'WORLD', 'PYTHON']

Object_Oriented Programming (OOP)

a programming pardigm that is based on the concept of objects. Objects are instances of classes, and they bundle data (attributes) and methods (functions) that operate on the data. OOP helps in organizing and structuring your code by creating reusable and modular pieces.

Key Concepts of OOP:

1. Class: A blueprint for creating objects. It defines attributes and methods that the objects created from the class will have.

2. Object: An instance of a class. Each object has its own attributes and can use the methods defined in the class.

3. Encapsulation: The practice of bundling the data (attributes) and methods that operate on the data into a single unit, the object. It restricts direct access to some attributes to protect the object's state.

4. Inheritance: A mechanism where one class can inherit attributes and methods from another class. This allows for code reusability and the creation of a hierarchical relationship between classes.

5. Polymorphism: The ability of different objects to respond to the same method call in different ways. For example, different classes may have their own implementations of a method with the same name.

6. Abstraction: Hiding the complex implementation details of methods and exposing only the necessary parts.

Basic Structure of a Class and Object:

1) Defining a Class and Creating an object:

```
class Car:                                    # Define a class 'Car'
    # Constructor method to initialize object attributes.
    def __init__(self, brand, model, year):
        self.brand = brand          # Att for the car's brand
        self.model = model          # Att for the car's model
        self.year = year            # Att for the car's manufacturing year

    # Method to display car details
    def display_info(self):
        print(f"Car: {self.brand} {self.model}, Year: {self.year}")

# Create an object (instance) of the class Car
my_car = Car("Toyota", "Corrolla", 2020)

# Call the method to display car information
my_car.display_info()  # output: Car: Toyota Corolla, Year: 2020
```

2) Encapsulation: You can make certain attributes private by prefixing them with two underscore (__). This hides them from external access.

```
Class Car:
    def __init__(self, brand, model, year):
        self.__brand = brand    # Private attribute
        self.model = model
        self.year = year

    def display_info(self):
        print(f"Car: {self.__brand} {self.model}, Year: {self.year}")

    # Method to get the private attribute
    def get_brand(self):
        return self.__brand

# Create an object
my_car = Car("Toyota", "Corolla", 2020)

# Accessing private attribute using a method
print(my_car.get_brand())          # output: Toyota
```

3) Inheritance: Allows you to create a new class (child class) based on an existing class (parent class). The child class inherits attributes and methods from the parent class.

```
# Parent class
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
```

```
    def display_info(self):
        print(f"Vehicle: {self.brand} {self.model}")


# Child class (inherits from Vehicle)
class Car(Vehicle):
    def __init__(self, brand, model, year):
        super().__init__(brand, model) # Call the parent class constructor
        self.year = year

    def display_info(self):
        # Override the parent class method
        print(f"Car: {self.brand} {self.model}, Year: {self.year}")

# Create an object of the child class
my_car = Car("Toyota", "Corolla", 2020)
my_car.display_info()    #output: Car: Toyota Corolla, Year: 2020
```

* The Car class inherits from Vehicle. The method display_info is overridden in the child class to provide a specific implementation for cars.

4) Polymorphism: Allows different classes to have methods with the same name but potentially different implementations. This makes it easier to work with different types of objects.

```
Class Animal:
    def sound(self):
        pass
```

```
class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"
# Polymorphism in action
animals = [Dog(), Cat()]

for animal in animals:
    print(animal.sound())    # output:
                             # Bark
                             # Meow
```

5) Abstraction: Achieved by hiding the implementation details of a method and exposing only the necessary information. In Python, this can be done using abstract classes with the abc module:

```
from abc import ABC, abstractmethod

class Animal(ABC):          # Abstract class
    @abstractmethod
    def sound(self):
        pass        # Abstract method, must be implemented by subclasses

class Dog(Animal):
    def sound(self):
        return "Bark"
# Trying to create an instance of Animal would raise an error
#animal = Animal()  # Cannot instantiate abstract class
```

۱۴۰۰/۴/۱۷

```
# Correct usage:
dog = Dog()
print(dog.sound())  # output: Bark
```

OOP Benefits:

- Modularity: You can divide the program into separate objects.
- Code Reusability: Inheritance allows for reusing existing code.
- Flexibility: Polymorphism allows objects of different types to be treated similarly.
- Maintainability: Encapsulation helps in isolating changes to specific parts of code.

~~OOP Practice Problems:~~

~~Create a Class~~

Random: A module that lets you work with random numbers, choices, and other probability_based functionalities.

1. Importing random: import random

2. Generating Random Numbers:

- random.random()
  - Generates a random floating_point number between 0.0 and 1.0.

Ex: print(random.random())

- random.randint(a,b)
  - Generates a random integer between a and b (both inclusive).

Ex: print(random.randit(1,10))

- random.uniform(a,b)
  - Generates a random floating_point number between a and b.

Ex: print(random.uniform(1.5, 10.5))

3. Choosing Random Elements: - random.choice(sequence)
   - Selects a random item from a sequence like a list, tuple, or string.

Ex:
```
Colors = ['red', 'blue', 'green', 'yellow']
print(random.choice(colors))
```

۱۴۰۰/۳/۱۹

- random.choices (sequence, k=n)
  - Chooses n random elements from a sequence, allowing repeats.

- random.sample (sequence, k=n)
  - Chooses n unique random elements from a sequence (no repeats)

4. Shuffling : random.shuffle (sequence)
   - Shuffles the elements of a list in place.

5. Seeding : You can set a "seed" for reproducibility using random.seed (value). This ensures the random numbers generated will be the same every time you run the code with the same seed.

Input Function

In Python, you can use the input() function to get input from the user. This input is always captured as a string, so if you need a different data type, you'll need to convert it.

Ex:
```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

```
age = int(input("Enter your age: "))
print("You are", age, "years old.")
```

```
height = float(input("Enter your height in meters: "))
print("Your height is", height, "meters.")
```

Taking multiple inputs at once:
You can take multiple inputs in a single line and split them using .split().

```
x, y = input("Enter two numbers seperated by a space: ").split()
x = int(x)
y = int(y)
print("Sum:", x+y)
```

Prompting the user with a default message:
You can add a prompt message to input() to give the user instructions directly in the function call:

```
favorite_color = input("what's your favorite color?")
```

String Methods:

1. string.capitalize(): converts the first character of a string to an uppercase letter and all other alphabets to lowercase.

2. string.casefold(): Converts all characters of the string into lowercase letters and return a new string.

3. string.center(): returns a new centered string after padding it with the specified character.
   Syntax ⇒ string.center(width, [fillchar])
   width = length of the string with padded characters.
   fillchar = (optional) _padding character (if not provided, whitespace is taken)

4. string.count(): returns the number of occurrences of a substring in the given string.
   Syntax ⇒ string.count(substring, start=..., end=...)
   substring = string whose count is to be found.
   start (optional) = starting index within the string where search starts.
   end (optional) = ending index within the string where search ends.

5. string.encode(): returns an encoded version of the given string.
   Syntax ⇒ string.encode(encoding='UTF-8', errors='strict')

6. string.endswith(): returns True if a string ends with the specified suffix. If not, it returns False.
   Syntax ⇒ string.endswith(suffix[, start[, end]])
   suffix = string or tuple of suffixes to be checked.
   start (optional) = Beginning position where suffix is to be checked
   end (optional) = Ending position where suffix is to be checked.

7. String.expandtabs(): returns a string where all '\t' characters are replaced with whitespace characters until the next multiple of tabsize parameter.

Syntax ⇒ string.expandtabs(tabsize)

The default tabsize is 8.

8. String.find(): returns the index of first occurrence of the substring (if found). If not found, it returns -1.

Syntax ⇒ string.find(sub[, start[, end]])

sub = the substring to be searched in the string.

start and end (optional) = The range.

9. string.rfind(): returns an integer value.

- if substring exists inside the string, it returns the highest index where substring is found.
- if substring doesn't exist inside the string, it returns -1.

10. string.format():

returns the formatted string.

print("Hello {}, your balance is {}.".format("Adam", 230.2346))

11. string.format_map(): Formats the string using dictionary.

12. string.index(): returns the index of a substring inside the string (if found). If the substring is not found, it raises an exception.

Syntax ⇒ string.index(sub[,start[, end]])

13. String.isalnum(): returns True if all characters in the string are alphanumeric (either alphabets or numbers). If not, it returns False.

14. string.isalpha(): Checks if all characters are alphabets.

15. string.isdecimal(): Checks if all characters are decimal numbers.

16. string.isdigit(): Checks if all characters are digits.
(only 0123456789)

17. string.isidentifier(): checks if the string is a valid identifier.

18. string.islower(): Checks if all alphabets in a string are lowercase.

19. string.isnumeric(): Checks if all characters in the string are numeric (½ counts as number)

20. string.isprintable(): Checks if all characters are printable

21. string.isspace(): Checks if all characters are whitespace.
('\t', '\n')

22. string.istitle(): Checks if the string is titlecased
'Python Is Good' ⇒ True
'Python is good' ⇒ False

23. string.isupper(): Checks if all characters are uppercase.

24. string.join(): returns a string by joining all the elements of an iterable (list, string, tuple), separated by the given separator.
separator.join(iterable)

25. string.ljust(): returns left-justified string of given width

26. string.lower(): returns lowercased string.

27. string.lstrip(): remores characters from the left based on the argument

Syntax ⇒ string.lstrip([chars])

chars = a string specifying the set of characters to be removed. If chars argument is not provided, all leading whitespaces are removed from the string.

28. string.maketrans(): creates a one to one mapping of a character to its translation/replacement.

29. string.partition(): takes a string parameter separator that separates the string at the first occurrence of it.
the method returns a 3_tuple.

30. string.replace(): replaces each matching occurrence of a substring with another string.

Syntax ⇒ string.replace(old, new [, count])

31. string.rindex(): if substring exists inside the string, it returns the highest index in the string where the substring is found.
if substring doesn't exist, it raises a ValueError exception.

Syntax ⇒ string.rindex(sub [, start [, end]])

32. string.rjust(): right aligns the string up to a given width using a specifed character.

33. string.rpartition(): like string.partition() but separates the string at the last occurrence of the sub.

34. string.rsplit(): breaks the string at the separator starting from the right and returns a list of strings.
Syntax ⇒ string.rsplit([separator [, maxsplit]])
- separator (optional) - if not specified, any whitespace string is a separator.
- maxsplit (optional): defines the max number of splits. the default value is -1, meaning, no limit on the number of splits.

35. string.rstrip(): returns a copy of the string with trailling characters removed (based on the string argument passed).
Syntax ⇒ string.rstrip([chars])
- chars (optional) - the set of trailings to be removed.
if not provided, all whitespaces on the right are removed.

36. string.split(): breaks down a string into a list of substrings using a chosen separator.

37. string.splitlines(): splits the string at line breaks and returns a list.

38. string.startswith(): returns True if a string starts with the specified prefix (string). If not, it returns False.

39. string.strip(): removes any leading and trailling whitespaces.

40. string.swapcase(): Converts all the characters to their opposite letter case.

41. string.title(): returns a title cased version of the string.

42. string.translate(): returns a string where each character is mapped to its corresponding character as per the translation table.

43. string.upper(): Converts all lowercase characters in a string into uppercase characters and returns it.

44. string.zfill(): returns a copy of the string with 0 filled to the left. The length of the returned string depends on the width provided.
Syntax ⇒ string.zfill(width)

۱۴۰۰/۳/۲۷

CSV (Comma-Separated Values)

CSV format is commonly used for storing tabular data, like data in a spreadsheet. Python's csv library allows you to work with CSV files easily.

1. Reading a CSV File: use the csv.reader() function.

Ex: csv file named data.csv:

```
name,age,city
Alice,30,New York
Bob,25,Los Angeles
Charlie,35,chicago
```

reading in Python:

```
import csv

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

```
#output:
['name','age','city']
['Alice','30','New York']
['Bob','25','Los Angeles']
['charlie','35','chicago']
```

2. Reading a CSV File as a Dictionary: use csv.DictReader()

```
with open("data.csv","r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(row["name"],row["age"],row["city"])
```

```
#output:
Alice 30 New York
Bob 25 Los Angeles
Charlie 35 Chicago
```

۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵ ۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹ ۳۰ ۳۱

روز بزرگداشت حضرت احمد بن موسی شاهچراغ (ع) - روز جهاد کشاورزی (تشکیل جهاد سازندگی به فرمان حضرت امام خمینی (ره) - ۱۳۵۸ ه.ش)

* csv.DictReader() reads each row in the CSV file as a dictionary where the keys are the column headers, and the values are the data in each row.

row ⇒ {'name': 'Alice', 'age': '30', 'city': 'New York'}
(stored as strings) {'name': 'Bob', 'age': '25', 'city': 'Los Angeles'}
{'name': 'Charlie', 'age': '35', 'city': 'Chicago'}

2.1. Customizing Column Names

If CSV file does not have headers, or you want to use custom headers, you can pass a list of fieldnames to csv.DictReader(). In this case, the first row will not be treated as headers.

Ex: data_no_header.csv :

```
Alice, 30, New York
Bob, 25, Los Angeles
Charlie, 35, Chicago
```

```
with open("data_no_header.csv", "r") as file:
    reader = csv.DictReader(file, fieldnames=["name", "age", "city"])
    for row in reader:
        print(row)
```

2.2. Skipping the Header Row

use next(reader) when the header in your file is not desired or if you're setting your own headers

2.3. Working with Different Delimiters

specify the delimiter (e.g., a semicolon) with the delimiter parameter

```
reader = csv.DictReader(file, delimiter=";")
```

۱۴۰۰/۳/۲۹

2.4. Handling Missing Values

If some rows in your CSV file are missing data, csv.DictReader() will assign None for missing values.

3. Writing to a CSV File: use csv.writer(). This will create a new CSV file or overwrite on existing one.

Ex:

```
import csv

                                    # Creates output.csv
data = [
    ["name", "age", "city"],
    ["Alice", 30, "New York"],
    ["Bob", 25, "Los Angeles"],
    ["Charlie", 35, "Chicago"]
]

with open("output.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(data)
```

4. Writing a CSV File from a Dictionary: use csv.DictWriter()

```
import csv

data = [
    {"name": "Alice", "age": 30, "city": "New York"},
    {"name": "Bob", "age": 25, "city": "Los Angeles"},
    {"name": "Charlie", "age": 35, "city": "Chicago"}
]
```

۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵ ۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹ ۳۰ ۳۱

```
with open("output.csv", "w", newline="") as file:
    fieldnames = ["name", "age", "city"]
    writer = csv.DictWriter(file, fieldnames=fieldnames)
    writer.writeheader()        # writes the header row
    writer.writerows(data)      # writes the data rows
```

5. Appending to a CSV File: open the file in append mode ("a")

```
new_data = ["David", 28, "Miami"]

with open("output.csv", "a", newline="") as file:
    writier = csv.writer(file)
    writier.writerow(new_data)
```

Additional Tips:

1. Delimiter: If your file uses a different delimiter (like ;) you can specify it.

```
reader = csv.reader(file, delimiter=";")
```

2. Skipping Headers: If your file has headers and you want to skip them, you can use next(reader) to skip the first row.

3. Handling Quotes and Special Characters: Sometimes CSV data includes commas, newlines, or quotes within fields. You can handle these by setting the quotechar and quoting parameters.

Ex:
```
data = [
    ["name", "age", "city"],
    ["Alice", 30, "New York, NY"],
    ["Bob", 25, "Los Angeles, CA"]
]
```

```
with open ("output_quoted.csv", "w", newline = "") as file:
    writer = csv.writer (file, quoting = csv.QUOTE_MINIMAL)
    writer.writerows (data)
```

QUOTE_MINIMAL : Quotes only fields that contains special chars. (default)
QUOTE_ALL : Quotes all fields.
QUOTE_NONNUMERIC : Quotes all non_numeric fields.
QUOTE_NONE : Disables quoting entirely.

4. Error Handling in CSV Operations : When reading or writing CSV files, you might encounter errors, especially if the file format is inconsistent. To handle errors gracefully, use a try ... except block.

Ex:

```
import csv

try:
    with open ("data.csv", "r") as file:
        reader = csv.reader (file)
        for row in reader:
            print(row)
except. FileNotFoundError:
    print ("The file was not found.")
except csv.Error as e:
    print (f"CSV error: {e}")
```

5. Using pandas for CSV Operations: The pandas library is widely used in data analysis and provides powerful tools for handling CSV files, making it great alternative to Python's csv module.

5.1. Reading a CSV with Pandas: You can read a CSV file into a DataFrame (a table-like structure) with just one line:

Ex: import pandas as pd

```
df = pd.read_csv("data.csv")
print(df)
```

This will automatically handle headers, delimiters, and missing values better than csv.reader().

5.2. Writing to a CSV with Pandas: You can save a DataFrame to a CSV file just as easily:

```
df.to_csv("output.csv", index=False)
```

6. Advanced Operations with Pandas: With pandas, you can perform powerful data manipulation operations on CSV data.

6.1. Filtering Data:

```
#Seled rows where age>30
filtered_df = df[df["age"]>30]
print(filtered_df)
```

6.2. Adding or Modifying Columns:

```
# Add a new column
df["new_column"] = df["age"] + 5
print(df)
```

6.3. Handling Missing Data: You can easily handle missing values (NaN) with pandas:

```
# Fill missing values
df.fillna("Unknown", inplace=True)
```

7. Large CSV Files: For very large CSV files, reading the entire file at once may not be efficient. You can read it in chunks with pandas, which lets you process the file in smaller parts.

```
chunk_size = 1000
for chunk in pd.read_csv("large_data.csv", chunksize=chunk_size):
    print(chunk.head())  # process each chunk
```

8. Reading and Writing CSV Files with Headers: In both csv.reader() and pandas.read_csv(), you can specify whether the CSV file has headers and how to handle them.

8.1. Skipping Headers with csv.reader(): If a CSV file has headers and you don't want to read them, you can use next():

```
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    next(reader)  # skip the header row
    for row in reader:
        print(row)
```

8.2. Specifying Headers with pandas: If the CSV file lacks headers, you can manually assign them with names:

```
df = pd.read_csv("data_no_headers.csv", names=["name", "age", "city"])
```

9. Writing to Specific Columns with csv.DictWriter():
With csv.DictWriter(), you can write specific columns to a CSV file, which can be useful when working with dictionaries:

```
import csv

data = [
    {"name": "Alice", "age": 30, "city": "New York"},
    {"name": "Bob", "age": 25, "city": "Los Angeles"}
]

with open("output.csv", "w", newline="") as file:
    fieldnames = ["name", "city"]  # Only write specific columns
    writer = csv.DictWriter(file, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerows(data)
```

This will output only the name and city columns, ignoring the age.

Rounding and Controlling Decimal Places:

1. Using round(): The round() function rounds a number to the specified number of decimal places.

Ex: x = 3.14159
print(round(x,2)) # 3.14 (rounded to 2 decimal places)

2. Using String Formatting: You can format a float when converting it to a string using f_string or the format() method.

Ex using f_string: pi = 3.14159
print(f"{pi:.2f}") # 3.14

Ex using format(): pi = 3.14159
print("{:.2f}".format(pi) # 3.14

3. Using Decimal Module: The decimal module provides more precision control and better handling of floating-point arithmetic. (for financial calculations)

Ex: from decimal import Decimal, ROUND_HALF_UP

x = Decimal("3.14159")
rounded_x = x.quantize(Decimal("0.01"), rounding=ROUND_HALF_UP)
print(rounded_x) # 3.14

١٤٠٠/٤/٦

Truncating a Float

1. Using String Slicing: Convert the number to a string and slice up to the required decimal places.

```
Ex: x = 3.14159
    truncated_x = float(str(x)[:str(x).find('.')+3])
    print(truncated_x)    # 3.14
```

2. Using math.trunc: The math.trunc function removes the decimal portion entirely, returning only the integer part.

```
Ex:  import math

     x = 3.14159
     truncated_x = math.trunc(x)
     print(truncated_x)    # 3
```

3. Using Integer Arithmetic: Multiply the number by 10^n (where n is the number of decimal places), truncate it using int(), and then divide back.

```
Ex: x = 3.14159
    n = 2
    truncated_x = int(x*(10**n))/(10**n)
    print(truncated_x)    # 3.14
```

4. Using Decimal Module: The Decimal module allows for truncation using the quantize() method with no rounding.

```
from decimal import Decimal, ROUND_DOWN

x = Decimal("3.14159")
truncated_x = x.quantize(Decimal("0.01"), rounding=ROUND_DOWN)
print(truncated_x)    # 3.14
```

lambda:

1. map:

```
mylist = [1, 3, 4, 2, 0.5]
list(map(lambda x: x*2, mylist))
# [2, 6, 8, 4, 1]

number = [10, 11, 8, 6, 100, 7, 9, 21]
list(map(lambda x: 'big' if x>10 else 'small', numbers))
# ['small', 'big', 'small', 'small', ... ]
```

★ professional way to use if: 'big' if x>10 else 'small'

2. filter:

```
mylist = [1, 5, 6, 8, 10, 11]
list(filter(lambda x: x%2 ==0, mylist))
# [6, 8, 10]
```

★ filter(func, list)

۱۴۰۰/۱۲/۸

Generator

def → problems with Memory and Time in big projects.
also they are limited

Ex:
```
def firstn():
    yield 1        first time the function is called, returns 1
    yield 2        second time, returns 2
    yield 3        Third time, returns 3

for i in firstn():      #output: (1, 2, 3)
    print(i)
```

Ex:
```
def firstn(n):
    num=0
    while (num<n):
        yield num
        num +=1          #output: (0,1,2,3,4,5,6,7,8,9,10)

for i in firstn(10):
    print(i)
```

Generators in Python are a way to create iterators in a more memory-efficient and readable manner. Instead of computing all the values at once and storing them in memory (like with a list), a generator produces one value at a time as needed.

Defining a Generator:

A generator can be created using:

1. Generator Functions: These use the yield keyword.
2. Generator Expressions: These resemble list comprehensions but use parentheses instead of square brackets.

1. Generator Functions:

A generator function is defined like a normal function but uses yield instead of return. When the function is called, it doesn't execute immediately; instead, it returns a generator object.

Ex:
```
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()
# Accessing values one by one
print(next(gen))    # output: 1
print(next(gen))    # output: 2
print(next(gen))    # output: 3
# print(next(gen))  # Raises StopIteration
```

* yield pauses the function and saves its state. When next() is called on the generator, it resumes from where it left off.

benefits: Memory efficiency: No need to store all values in memory.
Lazy evaluation: Values are generated only when needed.

۱۴۰۰/۴/۱۰

2. Generator Expressions:

A generator expression is similar to a list comprehension but uses parentheses.

Ex:
```
gen_exp = (x**2 for x in range(5))
print(next(gen_exp))    # output: 0
print(next(gen_exp))    # output: 1
print(list(gen_exp))    # output: [4, 9, 16]
```
(Remaining values)

Key Differences from List Comprehension:

- Uses less memory, especially for large ranges.
- Does not store all elements at once; Calculates them on the fly.

Practical Examples:

Infinite Generators: Generators can create infinite sequences, which are impossible with lists.

```
def infinite_numbers():
    num = 0
    while True:
        yield num
        num += 1

gen = infinite_numbers()
print(next(gen))    # output: 0
print(next(gen))    # output: 1
```

Fibonacci Sequence Generator:

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b

fib_gen = fibonacci()
for _ in range(5):
    print(next(fib_gen))   #output: 0, 1, 1, 2, 3
```

Reading Large Files Efficiently: Generators are great for processing large files line by line without loading the entire file into memory.

```
def read_large_file(file_name):
    with open(file_name, 'r') as file:
        for line in file:
            yield line.strip()

for line in read_large_file('large_file.txt'):
    print(line)    # Processes one line at a time.
```

Key Generator Methods:

1: next(generator); Moves the generator to the next value.

2: generator.send(value): Sends a value to the generator.
The yield statement will return the value sent.

3: generator.close(): Stops the generator.

Ex of send:

```
def my_generator():
    value = yield "Start"
    yield f"Received: {value}"

gen = my_generator()
print(next(gen))          #output: "Start"
print(gen.send(42))       #output: "Received: 42"
```

Numpy

Array Creation

Arrays are the core data structure in NumPy.

- 1D Array:

```
import numpy as np

arr_1d = np.array([10,20,30,40,50])
print("1D Array:", arr_1d)
```

- 2D Array:

```
arr_2d = np.array([[1,2,3],[4,5,6],[7,8,9]])
print("2D Array:\n", arr_2d)
```

- Random Array:

```
# Random array of shape (3,4)
random_arr = np.random.rand(3,4)
print("Random Array:\n", random_arr)
```

Special Arrays

These are useful for initializing data.

- Array of Zeros:

```
zeros = np.zeros((2,3))    # 2x3 array of zeros
print("Zeros Array:\n", zeros)
```

np.random.randint(start, stop, size=(x,y))
uniform => floating-point ~~numbers~~ like 0.673

np.random.rand(x, y) * (stop-start) + start
[0,1) scaling to 5 to 15

۱۴۰۰/۱۴/۱۴

- Array of Ones:

```
ones = np.ones((3,2))   # 3x2 array of ones
```

- Range of Numbers:

```
range_arr = np.arange(0, 20, 5)
# Start at 0, go up to 20, step by 5
# [0, 5, 10, 15, 20]
```

np.linspace(start, stop, num)
endpoint=False → excluding stop value
arr, step = np.---(retstep =True)

Array Properties:

Learn to inspect arrays.

```
arr = np.array([[1,2,3],[4,5,6]])
print("Array:\n", arr)
print("Shape:", arr.shape)      # Rows and columns
print("Dimensions:", arr.ndim)  # Number of dimensions
print("Size:", arr.size)        # Total number of elements
print("Data Type:", arr.dtype)  # Type of elements
```

Indexing and Slicing

- Accessing Elements:

```
# Access elements in a 2D array
print("Element at [0,1]:", arr[0,1])  # Row 0, Column 1
```

- Slicing:

```
# Get specific rows/columns
print("First row:", arr[0,:])  # All columns in the first row
print("Second column:", arr[:,1])  # All rows in the second column
```

۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵ ۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹ ۳۰ ۳۱

روز قلم

identity Matrix = np.eye(n) or np.identity(n)

١٤٠٠/٤/١٥

- Bodean Indexing:

```
# Filter elements greater than 3
filtered = arr[arr > 3]
print("Filtered Elements:", filtered)
```

Basic Array Operations

- Element-wise Operations:

```
a = np.array([1, 2, 3])
b = np.array([4,5,6])
print("Addition:", a+b)
print("Multiplication:", a*b)
```

- Aggregation Functions:

```
print("Sum:", arr.sum())
print("Mean:", arr.mean())
print("Max:", arr.max())
print("Min:", arr.min())
```

Reshaping and Transposing

- Reshaping Arrays:

```
reshaped = np.arange(12).reshape(3, 4) # Reshape to 3×4
print("Reshaped Array: \n", reshaped)
```

- Transposing Arrays:

```
transposed = reshaped.T  # Swap rows and columns
print("Transposed Array: \n", transposed)
```

١ ٢ ٣ ٤ ٥ ٦ ٧ ٨ ٩ ١٠ ١١ ١٢ ١٣ ١٤ ١٥ ١٦ ١٧ ١٨ ١٩ ٢٠ ٢١ ٢٢ ٢٣ ٢٤ ٢٥ ٢٦ ٢٧ ٢٨ ٢٩ ٣٠ ٣١

Element-wise Operations

NumPy allows operations directly on arrays without needing loops.

- Add, Multiply, Power

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

print("Addition:", a + b)          # [5, 7, 9]
print("Multiplication:", a * b)    # [4, 10, 18]
print("Power:", a ** 2)            # [1, 4, 9]
```

- Logical Operations

```
print("Comparision:", a > 2)       # [False, False, True]
```

Broadcasting

Broadcasting makes it possible to perform operations on arrays of different shapes.

- Adding Scalar to Array

```
arr = np.array([1, 2, 3], [4, 5, 6])
print("Add 10:\n", arr + 10)  # Each element gets 10 added
```

- Broadcasting Shapes

```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([10, 20, 30])
print("Broadcast Add:\n", a + b)

# [[11, 22, 33], [14, 25, 36]]
```

Broadcasting Rules:

1. If the dimensions differ, pad the smaller shape with 1 on the left.
2. If shapes are incompatible, NumPy raises an error.
3. Size 1 dimensions can stretch to match.

Rule 1: `[1, 2, 3]` + `[[10], [20], [30]]` ⇒
`[1, 2, 3]` → (3,) → becomes (1, 3)
`[[10], [20], [30]]` → (3, 1)

```
[[11 12 13]
 [21 22 23]
 [31 32 33]]
```
(3, 3)

Rule 2: `[1, 2, 3]` + `[[10], [20]]` ⇒ ValueError
`[1, 2, 3]` → (3,) → (1, 3)
`[[10], [20]]` → (2, 1)
dimensions do not match

Rule 3: `[[1], [2], [3]]` * `[10, 20, 30]` ⇒
`[[1], [2], [3]]` → (3, 1)
`[10, 20, 30]` → (3,) → (1, 3)

```
[[10 20 30]
 [20 40 60]
 [30 60 90]]
```

* np.broadcast_to (arr, (2,3)) : for visualizing

Horizontal Stacking: np.hstack ((arr1, arr2))

Vertical Stacking: np.vstack ((arr1, arr2))

Column Stacking: np.column_stack ((arr1, arr2))

Row Stacking: np.row_stack ((arr1, arr2))

۱۴۰۰/۴/۱۸

Mathematical Functions

NumPy offers many mathematical functions useful for machine learning preprocessing.

- Trigonometric Functions:

```
angles = np.array([0, np.pi/2, np.pi])
print("Sine:", np.sin(angles))      # [0, 1, 0]
print("Cosine:", np.cos(angles))    # [1, 0, -1]
```

- Exponentials and Logarithms

```
arr = np.array([1, 2, 3])
print("Exponential:", np.exp(arr)) #[e^1, e^2, e^3]
print("Logarithm:", np.log(arr))   # Natural log
```

Advanced Indexing

- Boolean Indexing

```
# Extract elements satisfying a condition
arr = np.array([1, 2, 3, 4, 5])
filtered = arr[arr>3]
print("Filtered:", filtered)      # [4, 5]
```

- Fancy Indexing

```
# Index with a list of indices
indices = [0, 2, 4]
print("Fancy Indexed:", arr[indices]) # [1, 3, 5]
```

۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵ ۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹ ۳۰ ۳۱

For 2D matrices ⇒ np.dot = @operator = np.matmul

* matmul : supports broadcasting for higher-dimensional arrays. Follows strict matrix multiplication rules.
* np.dot : Can also perform vector dot products and scalar multiplication.

١٤٠٠/٤/١٩

Linear Algebra

Linear algebra is essential for machine learning models like linear regression.

- Dot Product

$$C[i][j] = \sum_{k=1}^{n} A[i][k] \times B[k][j]$$

```
a = np.array([1, 2])
b = np.array([3, 4])
print("Dot Product:", np.dot(a, b)) # 1*3 + 2*4 = 11
```

or print(a @b) ⇒ shorter, more Pythonic syntax in newer Python versions.

- Matrix Multiplication

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print("Matrix Product: \n", np.matmul(a, b))
```

- Inverse and Determinant

```
# Inverse of a matrix
matrix = np.array([[1, 2], [3, 4]])
inverse = np.linalg.inv(matrix)
print("Inverse: \n", inverse)

# Determinant
det = np.linalg.det(matrix)
print("Determinant:", det)
```

$matrix = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ $det = 4 \times 1 - 3 \times 2 = -2$

$inverse = \frac{1}{-2} \begin{bmatrix} 4 & -2 \\ -3 & 1 \end{bmatrix} = \begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix}$

١ ٢ ٣ ٤ ٥ ٦ ٧ ٨ ٩ ١٠ ١١ ١٢ ١٣ ١٤ ١٥ ١٦ ١٧ ١٨ ١٩ ٢٠ ٢١ ٢٢ ٢٣ ٢٤ ٢٥ ٢٦ ٢٧ ٢٨ ٢٩ ٣٠ ٣١

۱۴۰۰/۴/۲۰

Statistical Operations

Understanding data distribution is vital for machine learning.

- Mean, Median, and Standard Deviation

```
data = np.array([1, 2, 3, 4, 5])
print("Mean:", np.mean(data))
print("Median:", np.median(data))
print("Standard Deviation:", np.std(data))
```

Standard Deviation ⇒ $\sigma = \sqrt{\frac{\Sigma(x_i - \mu)^2}{N}}$

- Percentiles

```
print("25th Percentile:", np.percentile(data, 25))  # 2
```

* The 25th percentile is the value below which 25% of the data falls.

```
np.percentile(data, [25, 50, 75])  # Multiple Percentiles
```

Optimizing Array Operations

- Avoiding Loops: Loops can slow down code. Instead, use vectorized operations.

```
#Slow
result = []
for x in arr:
    result.append(x**2)

# Fast
result = arr**2
```

- Memory Views: When slicing, NumPy provides a "view" instead of copying data.

```
a = np.array([1,2,3,4])
b = a[1:3]
b[0] = 99
print("Original Array:", a)   # [1 99 3 4]
```

Understanding Dimensions:

For a 2D array:

Axis 0: Operates along the rows (column_wise operation)

Axis 1: Operates along the columns (row-wise operation)

For a 3D array:

Axis 0: Operates along the depth (across matrices)

Axis 1: Operates along the rows (within each matrix)

Axis 2: Operates along the columns (within each matrix)

```
column_means = np.mean(arr, axis=0)
row_means = np.mean(arr, axis=1)
```

---

np.linalg.solve $\vec{A}.\vec{x} = \vec{B}$

$\vec{A}$: a coefficient matrix (2D array) where each row corresponds to the coefficients of one equation. (Must be squar matrix and invertible)

$\vec{x}$: a column vector to be solved.

$\vec{B}$: a column vector (or array) of constants on the right hand side of the equations.

$$2x + y = 5$$
$$x - y = 1$$

$$\begin{bmatrix} 2 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$$

Ex: import numpy as np

```
# Coefficient matrix A
A = np.array([[2, 1], [1, -1]])

#Constants vector B
B = np.array([5, 1])
# Solve for x
x = np.linalg.solve(A, B)
print("Solution (x):", x)
```

```
# output:
Solution (x): [2 1]
```

Pandas

Core Data Structures in Pandas

Pandas has two primary data structures:

1. Series: A one-dimensional labeled array.
2. DataFrame: A two-dimensional labeled table.

- Creating a Series:

```
import pandas as pd

# Create a Series from a list
s = pd.Series([10, 20, 30, 40])
print(s)
```

```
#  0    10
   1    20
   2    30
   3    40
dtype: int64
```

- Creating a DataFrame:

```
# Create a DataFrame from a dictionary
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 70000]
}
df = pd.DataFrame(data)
print(df)
```

```
#       Name  Age  Salary
    0   Alice   25   50000
    1     Bob   30   60000
    2  Charlie  35   70000
```

١٤٠٠/٤/٢٤

Reading and Writing Data

Pandas makes it easy to load and save data.

Reading CSV Files:

```
df = pd.read_csv('data.csv')
print(df.head()) # Displays the first 5 rows.
```

```
>> df = pd.read_csv('data.csv', sep='\t', index_col='ID')
```

sep: Defines the delimiter used (default is a comma).
header: Specifies which row to use as column names (default is 0)
index_col: Defines the column to use as the index
dtype: Allows specifying data types for columns.

Writing Data with to_csv():

Basic: `df.to_csv('output.csv', index=False)`

index = False: Prevents saving the DataFrame index as a column. If you want to save the index, set it to True.

Advanced: `df.to_csv('output.csv', sep=';', columns=['Name', 'Age'], index=False)`

columns = ['Name', 'Age']: Saves only the specified columns.

١ ٢ ٣ ٤ ٥ ٦ ٧ ٨ ٩ ١٠ ١١ ١٢ ١٣ ١٤ ١٥ ١٦ ١٧ ١٨ ١٩ ٢٠ ٢١ ٢٢ ٢٣ ٢٤ ٢٥ ٢٦ ٢٧ ٢٨ ٢٩ ٣٠ ٣١

Reading Excel Files with read_excel():

Pandas also allows reading Excel files. You can use the pd.read_excel() function to read data from .xls and .xlsx files.

Basic: df.read_excel('data.xlsx')
print(df.head())

Reading Multiple Sheets:

df.= pd.read_excel('data.xlsx', sheet_name = 'Sheet1')
print(df.head()) loads the sheet named 'Sheet1'

» use sheet_name = None to read all sheets into a dictionary of DataFrames.

Writing Excel Files with to_excel():

Basic: df.to_excel('output.xlsx', index = False)

Writing Multiple Sheets:

```
with pd.ExcelWriter('output.xlsx') as writer:
    df.to_excel(writer, sheet_name = 'Sheet 1', index = False)
    df.to_excel(writer, sheet_name = 'Sheet 2', index = False)
```

ExcelWriter: allows you to write multiple DataFrames to different sheets in the same Excel file.

۱۴۰۰/۱۲/۲۶

Reading Data from a SQL Database:

If your data is stored in a SQL database, Pandas can also handle that. You'll need to install the required SQL database connector (e.g., sqlite3, sqlalchemy, or others.)

Basic:

```
import sqlite3

#Connect to a SQLite database
conn = sqlite3.connect('database.db')

#Read a table from the database into a DataFrame.
df = pd.read_sql_query('SELECT * FROM table_name', conn)
print(df.head())
```

pd.read_sql_query(): Executes an SQL query and returns the result as a DataFrame.

Conn: The database connection object.

Writing Data to a SQL Database:

Pandas can also write DataFrames back into a SQL database using to_sql().

```
df.to_sql('table_name', conn, if_exists='replace', index=False)
```

if_exists='replace': If the table exists, it will be replaced with the new data. You can also use 'append' to add to the existing table.

Handling Large Files:

For large datasets, reading and writing files in chunks can help manage memory usage.

Reading Data in Chunks:

```
chunk_size = 1000  #Number of rows per chunk
chunks = pd.read_csv('large_file.csv', chunksize=chunk_size)

    #Process each chunk
for chunk in chunks:
        #Perform operations on each chunk
        print(chunk.head())
```

Inspecting Data

1. Understanding the Dataset:

- A dataset is a collection of data organized in rows and columns (like a table)
- **Rows:** Represent individual data points or samples (e.g., one row = one person)
- **Columns:** Represent features or variables (e.g., age, income, height).

---

2. Common Formats of Datasets:

- **CSV:** Text files where each row is a data record, and columns are separated by commas.
- **Excel Files:** .xls or .xlsx files often used for tabular data.

- Databases: Data stored in SQL databases.
- JSON/XML: For hierarchial or nested data.

---

3. Loading Data into Python:

Libraries like pandas help you load and manipulate datasets easily.

```
import pandas as pd

# Load a CSV file into a DataFrame.
df = pd.read_csv('dataset.csv')
```

---

4. Inspecting the Structure of the Dataset:

Use these commands in pandas to get an overview of the data:

- df.head(): View the first few rows.
- df.info(): Get information about columns, data types, and non_null values.
- df.describe(): Summarize numerical columns (e.g., mean, median, standard deviation).

```
print(df.head())      # Preview the top 5 rows
print(df.info())      # Overview of data types and missing values
print(df.describe())  # Summary statistics for numerical data.
```

---

5. Checking for Missing Values:

Missing values can impact your model's performance.

- df.isnull().sum(): Counts missing values per column.
- Handle them using:
  - Dropping rows/columns (df.dropna()).
  - Imputation (filling missing values with the mean/median/mode).

6. Checking for Duplicates:

Duplicates can skew your results:

- df.duplicated(): Find duplicates.
- df.drop_duplicates(): Remove duplicate rows.

---

7. Exploring Individual Columns:

- Check unique values: df['column_name'].unique()
- Value counts: df['column_name'].value_counts()

---

8. Visualizing Data:

Visualizations help you understand the data distribution and patterns.

- Histogram for numerical data:

```
df['column_name'].hist()
```

- Boxplot for outliers:

```
import seaborn as sns
sns.boxplot(data = df['column_name'])
```

---

9. Understanding Data Distributions:

- Mean, Median, Mode: Central tendencies of data.
- Variance/Standard Deviation: Spread of data.
- Outliers: Extreme values that may need handling.

---

10. Exploring Relationships Between Variables:

- Correlations: Use .corr() to find relationships between numerical columns.
- Scatter plots: Visualize relationships between two variables.

```
import matplotlib.pyplot as plt
plt.scatter(df['feature1'], df['feature2'])
plt.show()
```

Selecting and Filtering Data

1. Selecting Columns:

- Use column names to extract specific features (columns) for analysis or modeling.

Single Column:

```
df['column_name']
```

Multiple Columns:

```
df[['column1', 'column2']]
```

Ex:

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'Salary': [50000, 60000, 70000]}
df = pd.DataFrame(data)

# Select the 'Age' column
print(df['Age'])

# Select 'Name' and 'Salary' columns
print(df[['Name', 'Salary']])
```

2. Selecting Rows by Index:

- Use the index to select specific rows.

Select First Few Rows:

```
df.head(n)  # First n rows
```

Select Specific Rows by Position:

```
df.iloc[0]  # First row (Zero-based index)
df.iloc[1:3]  # Row 1 and 2
```

Select Specific Rows by Label:

```
df.loc[0]  # Row with index label 0
```

---

3. Filtering Rows by Conditions:

- Filter rows based on one or more conditions.

Single Condition:

```
df[df['column_name'] > value]
```

Multiple Conditions:

```
df[(df['column1'] > value1) & (df['column2'] < value2)]
```

---

4. Selecting Specific Cells:

- Access specific values using row and column indices.

By Position:

```
df.iloc[row_index, column_index]
```

By Label:

```
df.loc[row_label, column_label]
```

Ex:

```
#Get value at row1, column 'Salary'
print(df.loc[1, 'Salary'])
```

١٤٠٠/٥/١

5. Applying Functions to Columns:

- Apply operations to modify or extract information from columns.

Ex:

```
# Add a new column
df['Bonus'] = df['Salary'] * 0.1
print(df)

# Apply a custom function
df['Salary'] = df['Salary'].apply(lambda x: x + 1000)
```

6. Advanced Selection Using query:

- Use SQL-like syntax for filtering rows.

Ex:

```
# Filter rows where Age > 25
print(df.query('Age > 25'))
```

١٤٠٠/٥/٢

Modifying Data

1. Adding New Columns:

- Why: To create new features from existing data.
- How: Diredly assign values or use existing columns to calculate new values.

Ex:

```
import pandas as pd
data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Age': [25, 30, 35],
'Salary': [50000, 60000, 70000]}
df. pd. DataFrame (data)

# Add a 'Bonus' column (10% of salary)
df['Bonus'] = df['Salary'] * 0.1
print(df)
```

---

2. Renaming Columns:

- Why: To make column names more meaningful or consistent.
- How: Use the rename function.

Ex:

```
# Rename 'Age' to 'Years' and 'Salary' to 'Income'
df.rename (columns = {'Age': 'Years', 'Salary': 'Income'}, inplace = True)
print(df)
```

١ ٢ ٣ ٤ ٥ ٦ ٧ ٨ ٩ ١٠ ١١ ١٢ ١٣ ١٤ ١٥ ١٦ ١٧ ١٨ ١٩ ٢٠ ٢١ ٢٢ ٢٣ ٢٤ ٢٥ ٢٦ ٢٧ ٢٨ ٢٩ ٣٠ ٣١

١٤٠٠/٥/٣

3. Modifying Column Values:

- Why: To correct errors, normalize values, or apply transformations
- How: Use the apply method or vectorized operations.

Ex:

```
# Increase Salary by 10%
df['Salary'] = df['Salary'] * 1.1

# Apply a custom function to round Salary
df['Salary'] = df['Salary'].apply(lambda x : round(x))
print(df)
```

---

4. Deleting Columns or Rows:

- Why: To remove irrelevant data.
- How: Use the drop function.

Ex:

```
# Drop the 'Bonus' column
df.drop(columns = ['Bonus'], inplace = True)

# Drop the first row
df.drop(index = 0, inplace = True)
print(df)
```

١ ٢ ٣ ٤ ٥ ٦ ٧ ٨ ٩ ١٠ ١١ ١٢ ١٣ ١٤ ١٥ ١٦ ١٧ ١٨ ١٩ ٢٠ ٢١ ٢٢ ٢٣ ٢٤ ٢٥ ٢٦ ٢٧ ٢٨ ٢٩ ٣٠ ٣١

5. Changing Data Types:

- Why: Ensures data is in the correct format (e.g., Converting strings to integers)
- How: Use the astype method.

Ex:

```
data = {'Name': ['1', '2', '3'], 'Age': [25.0, 30.5, 35.1]}
df= pd.DataFrame(data)

# Convert 'Name' to integers and 'Age' to integers.
df['Name'] = df['Name'].astype(int)
df['Age'] = df['Age'].astype(int)
print(df)
```

---

6. Handling Missing Data:

- Why: Missing values can cause errors or bias in machine learning models.
- How:
  - Fill with a specific value (fillna)
  - Drop rows or columns (dropna)

Ex:

```
data = {'Name': ['Alice', 'Bob', None], 'Age': [25, None, 35]}
df = pd.DataFrame(data)

# Fill missing values with a placeholder
df['Name'].fillna('Unknown', inplace = True)

# Drop rows with missing values
df.dropna(inplace=True)
print(df)
```

۱۴۰۰/۱۰/۵

7. Replacing Values:

- Why: Replace specific values to correct errors or standardize.
- How: Use the replace method.

Ex:

```
data = {'Name': ['Alice', 'Bob', 'Charlie'], 'Department': ['HR', 'IT', 'HR']}
df = pd.DataFrame(data)

#Replace 'HR' with 'Human Resources'
df['Department'] = df['Department'].replace('HR', 'Human Resources')
print(df)
```

---

8. Sorting Data:

- Why: Helps in organizing data for analysis or visualization.
- How: Use the sort_values method.

Ex:

```
# Sort by 'Age' in ascending order
df.sort_values(by='Age', ascending=True, inplace=True)
print(df)
```

---

9. Merging and Joining Datasets:

- Why: Combine multiple datasets into one.
- How:
  - Merging: Combines datasets based on a key.
  - Joining: Similar to merging but aligns datasets by their index.

۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵ ۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹ ۳۰ ۳۱

سالروز عملیات افتخار آفرین مرصاد (۱۳۶۷ ه.ش)

Ex:

```
data1 = {'ID': [1,2], 'Name': ['Alice', 'Bob']}
data2 = {'ID': [1,2], 'Salary': [50000, 60000]}
df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

# Merge on 'ID'
merged_df = pd.merge(df1, df2, on='ID')
print(merged_df)
```

---

10. Pivoting and Reshaping Data:

- **Why:** Rearrange data into a format that's easier to analyze.
- **How:** Use pivot, melt, or pivot_table.

Ex:

```
data = ['Name': ['Alice', 'Bob'], 'Year': [2020, 2020], 'Salary':
 [50000, 60000]}
df = pd.DataFrame(data)

# Pivot data
pivot = df.pivot(index='Name', Columns='Year', values='Salary')
print(pivot)
```

Handling Missing Data

Missing data is common in machine learning datasets.

- Detect Missing Values:

```
print(df.isnull())  # Shows True where data is missing
print(df.isnull().sum())  # Count of missing values per column.
```

- Fill Missing Values:

```
df['Age'].fillna(df['Age'].mean(), inplace=True)
                                    # Replace NaNs with mean
```

- Drop Rows/Columns with Missing Data:

```
df.dropna(inplace=True)  # Drop rows with NaNs
```

Combining Data

1. Merging Data:

- **Why:** Merging datasets is useful when you have separate datasets with complementary information (e.g., customer details in one dataset and purchase history in another).
- **How:** Use the merge() function to combine datasets based on a common column or index, similar to SQL joins.

Types of Merges (Joins):

- Inner Join: Only includes rows that have a match in both datasets.
- Left Join: Includes all rows from the left dataset, and the matching rows from the right dataset.
- Right Join: Includes all rows from the right dataset, and the matching rows from the left dataset.
- Outer Join: Includes all rows from both datasets, with NaNs where there are no matches.

Ex:

```
import pandas as pd

# Dataset 1: Customer details
df1 = pd.DataFrame({
    'CustomerID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
})

# Dataset 2: Purchase history
df2 = pd.DataFrame({
    'CustomerID': [1, 2, 4],
    'Purchase': ['Laptop', 'Phone', 'Tablet']
})

# Merge using 'CustomerID' column (Inner Join by default)
merged_df = pd.merge(df1, df2, on='CustomerID', how='inner')
print(merged_df)
```

output:

	CustomerID	Name	Purchase
0	1	Alice	Laptop
1	2	Bob	Phone

---

2. Joining Data (based on Index):

- Why: If two datasets have the same index (e.g., time series data), you may want to join them based on their index.
- How: Use the join() function, which joins data based on the index by default.

۱۴۰۰/۱۰/۹

Ex:

```
# Dataset 1 : Customer details
df1 = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Charlie']
}, index=[1, 2, 3])

# Dataset 2 : Purchase history
df2 = pd.DataFrame({
    'Purchase': ['Laptop', 'Phone', 'Tablet']
}, index=[1, 2, 4])

# Join by index
joined_df = df1.join(df2, how='inner')
print(joined_df)
```

#output:

	Name	Purchase
1	Alice	Laptop
2	Bob	Phone

In this example, only the rows with index 1 and 2 are kept because the row with index 4 does not have a match.

---

3. Concatenating Data:

- **Why:** Concatenating is useful when you have datasets with the same structure) and you want to stack them vertically (rows) or horizontally (columns).
- **How:** Use the concat() function.

١٤٠/٥/١٠

Vertical Concatenation (Stacking Rows):

- Combines rows from multiple datasets that share the same columns.

Ex:

```
# Dataset 1
df1 = pd.DataFrame({
    'Name': ['Alice', 'Bob'],
    'Age': [25, 30]
})

# Dataset 2
df2 = pd.DataFrame({
    'Name': ['Charlie', 'David'],
    'Age': [35, 40]
})

# Concatenate vertically (add rows)
Concatenated_df = pd.concat([df1, df2], axis=0, ignore_index=True)
print(Concatenated_df)
```

Output:

	Name	Age
0	Alice	25
1	Bob	30
2	Charlie	35
3	David	40

Horizontal Concatenation (Stacking Columns):
Combine datasets that share the same index, adding new columns.

Ex:

```
# Dataset 1
df1 = pd.DataFrame({
    'Name': ['Alice', 'Bob']
})

# Dataset 2
df2 = pd.DataFrame({
    'Age': [25, 30]
})

# Concatenate horizontally (add columns)
concatenated_df = pd.concat([df1, df2], axis=1)
print(concatenated_df)

#output:      Name  Age
          0  Alice   25
          1    Bob   30
```

4. Appending Data:

- Why: Appending is useful when you want to add rows from one dataset to another, similar to vertical concatenation.
- How: Use the append() method.

Ex:

```
# Dataset 1
df1 = pd.DataFrame({
    'Name': ['Alice', 'Bob'],
    'Age': [25, 30]
})

# Dataset 2
df2 = pd.DataFrame({
    'Name': ['Charlie'],
    'Age': [35]
})

# Append df2 to df1
appended_df = df1.append(df2, ignore_index=True)
print(appended_df)
```

#Output:

	Name	Age
0	Alice	25
1	Bob	30
2	Charlie	35

Sorting Data

1. Sorting by Columns:

- sorting by a single Column: Use the sort_values() func.
  default is ascending order. for descending use: ascending=False

Ex:

```
import pandas as pd

# Sample data
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 20]}
df = pd.DataFrame(data)

# Sort by 'Age' in ascending order
df_sorted = df.sort_values(by='Age')
print(df_sorted)
```

```
# output:      Name  Age
             2  Charlie  20
             0  Alice    25
             1  Bob      30
```

- Sorting by Multiple Columns:

Ex:

```
data = {'Name': ['Alice', 'Bob', 'Charlie', 'Alice'],
        'Age': [25, 30, 20, 25],
        'Salary': [50000, 60000, 70000, 55000]}
df = pd.DataFrame(data)
```

```
# Sort first by 'Age', then by 'Salary'
df_sorted = df.sort_values(by=['Age', 'Salary'], ascending=[True, False])
print(df_sorted)
```

#output:

	Name	Age	Salary
2	Charlie	20	70000
0	Alice	25	55000
3	Alice	25	50000
1	Bob	30	60000

First, the data is sorted by Age in ascending order. Then, for rows with the same Age, it is sorted by salary in descending order.

---

2. Sorting by Index:

- **Why:** Sometimes, you might need to sort by the DataFrame's index rather than by column values.
- **How:** Use sort_index() to sort the DataFrame by its index.

Ex:

```
# Sort by index (in ascending order by default)
df_sorted_index = df.sort_index()
print(df_sorted_index)
```

#output:

	Name	Age	Salary
0	Alice	25	50000
1	Bob	30	60000
2	Charlie	20	70000
3	Alice	25	55000

١٤٠٠/٥/١٥

3. Sorting for Time Series Data:

- **Why:** Time series data often requires sorting based on the time column (e.g., year, month, date)
- **How:** Convert the time column to datetime and the sort.

Ex:

```
data = {'Date': ['2020-03-01', '2020-01-01', '2020-02-01'],
        'Temperature': [22, 25, 20]}
df = pd.DataFrame(data)

# Convert 'Date' column to datetime.
df['Date'] = pd.to_datetime(df['Date'])

# Sort by 'Date'
df_sorted = df.sort_values(by='Date')
print(df_sorted)
```

#output:

	Date	Temperature
1	2020-01-01	25
2	2020-02-01	20
0	2020-03-01	22

Pandas for Machine Learning

In machine learning, Pandas is used for:

1. Data Cleaning: Handling missing or inconsistent data.
2. Feature Engineering: Creating new features or selecting subsets of data.
3. Exploratory Data Analysis (EDA): Understanding the data through visualization and statistics.
4. Input to Machine Learning Models: Converting DataFrames to NumPy arrays or tensors.

Converting Pandas DataFrame to NumPy:

```
x = df[['Age', 'Salary']].values    # Input features
y = df['Bonus'].values              # Target variable
```

۱۴۰۰/۰۵/۱۷

Pandas Indexing Techniques

Indexing in Pandas is essetial for selecting, filtering, and manipulating data efficiently in Series or DataFrames. Below are the key indexing techniques in Pandas, explained in detail with examples:

1. Selecting Rows and Columns by Labels (loc[])

The loc[] method is label_based. It selects rows and columns using labels or boolean conditions.

Ex: import pandas as pd

```
# Create a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 35, 40],
        'Salary': [50000, 60000, 70000, 80000]}
df = pd.DataFrame(data, index=['A', 'B', 'C', 'D'])

# Select a single row by label
print(df.loc['A'])

# Select multiple rows by Labels
print(df.loc[['A', 'C']])

#Select rows and specific columns
print(df.loc['A', ['Name', 'Salary']])

# Apply a condition to select rows
print(df.loc[df['Age'] > 30])
```

2. Selecting Rows and Columns by Position (iloc[])

The iloc[] method is position-based. It uses integer indices for rows and columns.

Ex:

```
# Select a single row by position
print (df.iloc[0])

# Select multiple rows by positions
print (df.iloc[0:2])

# Select specific rows and columns by position:
print (df.iloc[1,2]    # Second row, third column
print (df.iloc[0:2,1:3]) # First two rows, second and third columns.
```

---

3. Bolean Indexing

You can use boolean conditions to filter rows or columns based on specific criteria.

Ex:

```
# Filter rows where Age>30
print (df[df['Age']>30])

# Filter rows where Salary is less than 70000
print (df[df['Salary'] < 70000])
```

۱۴۰۰/۵/۱۹

4. Indexing with at[] and iat[]

- at[]: Access a single value by row and column label (faster for single-value access).
- iat[]: Access a single value by row and column position.

Ex:

```
# Access a single value using at[]
print(df.at['A', 'Age'])  # output: 25

# Access a single value using iat[]
print(df.iat[0, 1])  # output: 25 (first row, second column)
```

---

5. Index Slicing with Rows and Columns

You can slice rows and columns using labels or positions.

Ex:

```
# Slice rows by labels
print(df.loc['A' : 'C'])

# Slice rows by positions
print(df.iloc[1:3])

# Slice both rows and columns
Print(df.loc['A': 'C', 'Name': 'Age'])
```

6. Using .set_index() and .reset_index()

- set_index(): Change the DataFrame index to one of the columns.
- reset_index(): Reset the index to the default integer-based index.

Ex:

```
# Set the 'Name' column as the index
df_indexed = df.set_index('Name')
print(df_indexed)

# Reset the index
df_reset = df_indexed.reset_index()
print(df_reset)
```

---

7. MultiIndex (Hierarchical Indexing)

Pandas supports MultiIndex for more complex data organization.

Ex:

```
# Create a MultiIndex DataFrame
arrays = [['Group1', 'Group1', 'Group2', 'Group2'], ['A', 'B', 'C', 'D']]
index = pd.MultiIndex.from_arrays(arrays, names=('Group', 'Label'))
df_multi = pd.DataFrame({'Values': [10, 20, 30, 40]}, index=index)

print(df_multi)

# Access data from a specific group
print(df_multi.loc['Group1'])
```

Group	Label	values
Group1	A	10
	B	20
Group2	C	30
	D	40

Label	Values
A	10
B	20

۱۴۰۰/۵/۲۱

8. Accessing Columns Directly

You can access columns using dot notation (df.column) or bracket notation (df['column']).

Ex:

```
# Access the 'Age' column
print(df['Age'])

# Add a new column
df['Bonus'] = df['Salary'] * 0.1
print(df)
```

---

9. Conditional Indexing with .query()

The query() method provides a cleaner way to filter data using conditions.

Ex:

```
# Filter rows where Age > 30
print(df.query('Age > 30'))
```

---

10. Selecting Data with .xs()

Use .xs() to select data at a particular level in MultiIndex DataFrames.

Ex:

```
# Seled rows from a specific group in MultiIndex
print(df_multi.xs('Group1'))
```

query() Method

1. Readability: Conditions in query() are written in a natural, SQL-like syntax.
2. Efficiency: Faster than boolean indexing for large DataFrames due to internal optimizations.
3. Dynamic Filtering: Supports variables for dynamic condition building.

DataFrame.query(expr, inplace=False, **kwargs)
expr: A string containing the condition to filter the DataFrame.
inplace: If True, modifies the DataFrame in place.
**kwarg: Additional arguments, like local_dict or global_dict, for defining variables.

- Filter with Multiple Conditions: df.query('Age > 30 and Salary > 65000')

- Select Columns Based on Filtered Rows:
df.query('Age > 30')[['Name', 'Salary']]

- Using Variables in query(): @
threshold_age = 30
df.query('Age > @threshold_age')

- Using String Operations in query(): == or != or .str.contains()
df.query('Name == "Alice"')

- Query with Mathematical Expressions:
df.query('Salary >= 2*Age * 1000')

۱۴۰۰/۵/۲۲

- Using isnull() and notnull(): for Dataframes with missing values.

```
df_missing.query('Name.notnull()')
```

۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵ ۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹ ۳۰ ۳۱

روز مقاومت اسلامی

Matplotlib

A versatile library used for data visualization in Python. It allows you to create various types of plots like line charts, bar charts, scatter plots, and more.

1. Installation:

```
pip install matplotlib
```

Importing ⇒ `import matplotlib.pyplot as plt`

2. Creating Basic Plots:

- Line Plot:

```
x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]
plt.plot(x, y)
plt.show()
```

- Adding Labels and Title:

```
plt.xlabel("X-axis Label")
plt.ylabel("Y-axis Label")
plt.title("Basic Line Plot with Labels")
```

- Adding a Legend: Use the label argument in plt.plot()

```
x = [1, 2, 3, 4, 5]
y1 = [10, 20, 25, 30, 40]
y2 = [5, 15, 20, 25, 35]
plt.plot(x, y1, label = "Dataset 1", color = 'blue')
plt.plot(x, y2, label = "Dataset 2", color = 'green')
```

```
plt.xlabel ("X-axis")
plt.ylabel ("Y-axis")
plt.title ("Line Plot with Legend")
plt.legend()
plt.show()
```

3. Customizing Line Plots:

- Changing Line Color: Color parameter

```
plt.plot(x, y, color = 'red')
```

'-' '--' '-.' ':'

- Changing Line Style: solid, dashed, dash-dot, dotted

```
plt.plot(x, y, linestyle = '--', color = 'blue')
```

- Adding Markers: highlight data points 'o' circle, 's' square, '^' triangle, etc

```
plt.plot(x, y, marker = 'o', color = 'green')
```

- Combining Styles:

```
plt.plot(x, y, color = 'purple', linestyle = '-.', marker = 's')
```

- Setting Axis Limits: plt.xlim() and plt.ylim()

```
plt.plot(x, y, color = 'orange')
plt.xlim(0, 6)
plt.ylim(0, 50)
```

⋮

4. Other Plot Types:

- **Scatter Plot:** for visualizing relationships between two variables.

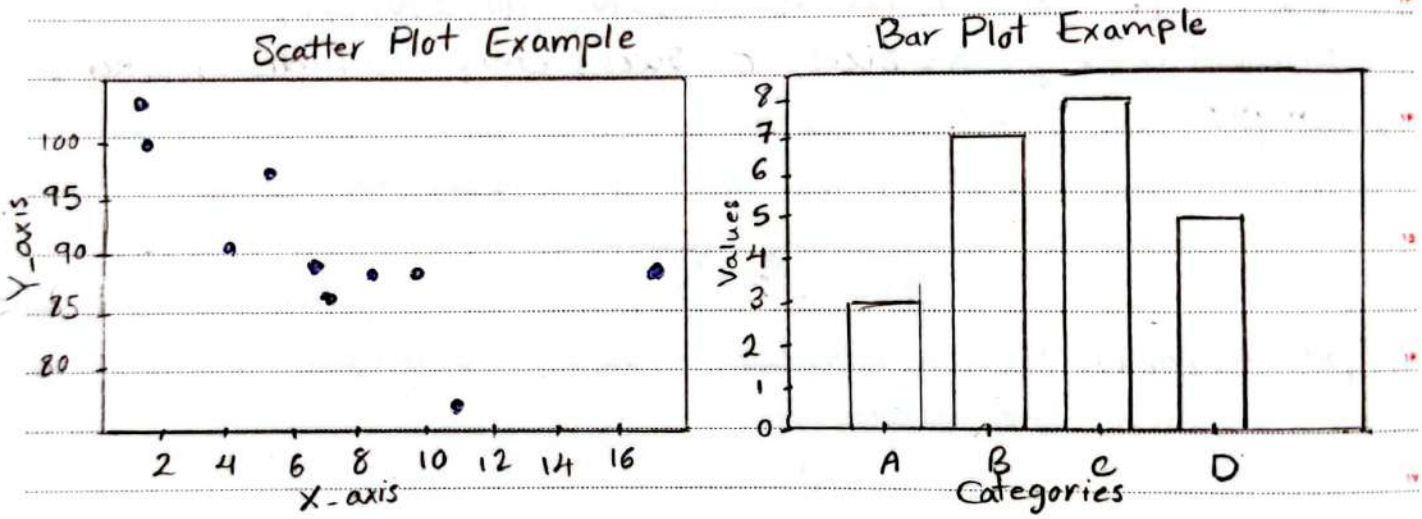
```
plt.scatter(x, y, color='blue', marker='o')
```

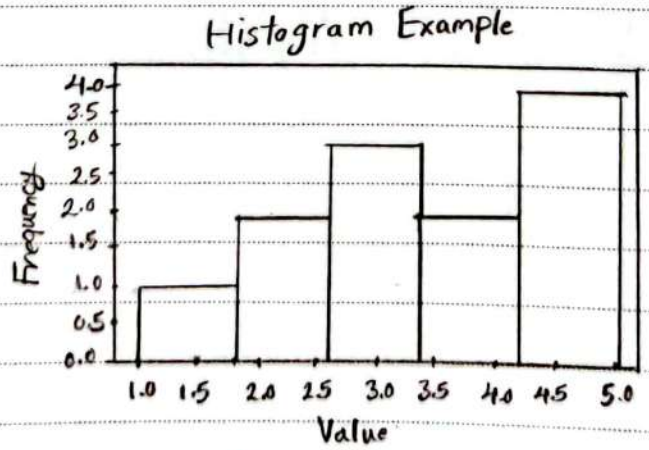
- **Bar Chart:** for comparing categorical data. Horizontal Bar chart ↳ plt.barh()

```
categories = ['A', 'B', 'C', 'D']
values = [3, 7, 8, 5]
plt.bar(categories, values, color='green')
⋮
```

- **Histogram:** for showing the frequency distribution of a dataset.

```
data = [1, 2, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5]
plt.hist(data, bins=5, color='orange', edgecolor='black')
```





۱۴۰۰/۵/۲۹

5. Customizing Other Plot Types:

- Customizing Scatter Plot:
  - Changing Marker Size (s): Use the s parameter to control the size of the markers
  - Changing Marker Color (c): Use the c parameter to assign different colors to points based on some data values.
  - Changing Marker Style (marker): Use different symbols like 'o', '^', s, x, etc.

Ex:

```
x = [5, 7, 8, 7, 2, 17, 2, 9, 4, 11]
y = [99, 86, 87, 88, 100, 86, 103, 87, 94, 78]
sizes = [30, 60, 90, 120, 150, 180, 210, 240, 270, 300]
plt.scatter(x, y, s = sizes, c = sizes, cmap = 'viridis', marker = '^')
```

* cmap: maps numeric data to colors. It works with continuous or discrete ranges of values. Use colormaps to make data visualization more informative and visually appealing.

* plt.colorbar(): Adds a colorbar to the side of the plot to help interpret the color values.

* Common Colormaps: 'viridis', 'plasma', 'cividis', 'Greens', 'coolwarm', 'seismic', 'tab10', 'Set1', 'rainbow', 'jet'

- Customizing Bar Charts:
  - Changing Bar Color: Use the color parameter.
  - Changing Bar Edge Color and Width: Use edgecolor and linewidth
  - Horizontal vs Vertical Bars: Use plt.bar() for vertical bars and plt.barh() for horizontal bars.

Ex: plt.bar(categories, values, color='skyblue', edgecolor='blue', linewidth=2)

- Adding Text to Bars: You can add Text labels to each bar with plt.text():

```
plt.bar(categories, values, color='skyblue', edgecolor='blue', linewidth=2)
for i,v in enumerate(values):
    plt.text(i, v+0.2, str(v), ha='center', color='black')
```

v+0.2: above the bar; ha='center': horizontal alignment

- Customizing Histogram:
  - Changing Number of Bins: Use the bins parameter to adjust the number of bins.
  - Color Customization: Use the color and edgecolor parameters.
  - Grid Lines: You can add grid lines to a histogram for better readability.

Ex:
```
plt.hist(data, bins=4, color='purple', edgecolor='black')
plt.grid(True)
```

۱۴۰۰/۰۵/۳۱

6. Subplots :

- Creating Subplots : Use plt.subplot() to define the layout of subplots. The syntax is:

  plt.subplot(nrows, ncols, index)

  nrows : Number of rows in the grid.

  ncols : Number of columns in the grid.

  index : The position of the current subplot (numbered left-to-right, top-to-bottom).

Ex:

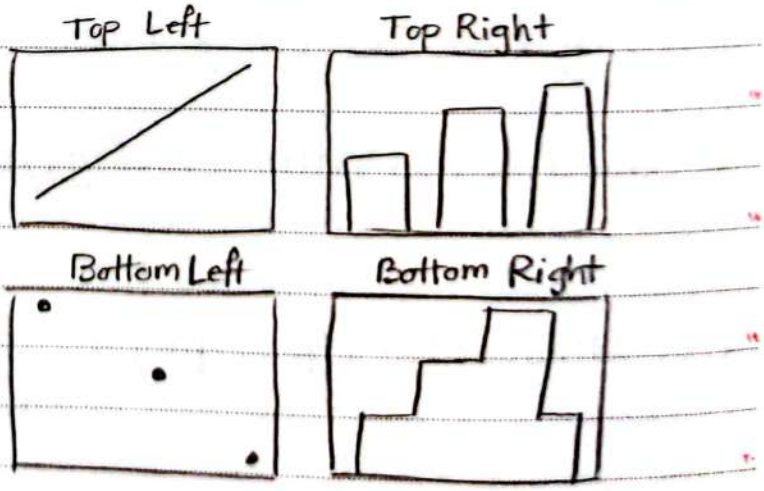
```
plt.subplot(1, 2, 1)
plt.plot([1, 2, 3], [4, 5, 6], color='blue')
plt.title("Plot 1")        add x-y labels after title for each
                           subplot.
plt.subplot(1, 2, 2)
plt.plot([1, 2, 3], [6, 5, 4], color='red')
plt.title("Plot 2")        plt.tight_layout() → Adjust spacing
plt.show()                                      to prevent overlap.
                           or use constrained_layout = True
```

۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵ ۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹ ۳۰ ۳۱

- Using plt.subplots(): A more versatile way to create subplots, returning a figure (fig) and an array of axes (ax).

Ex:

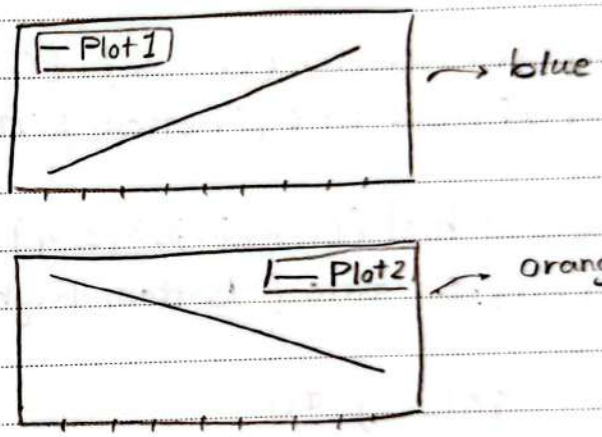
```
fig, ax = plt.subplots(2,2)  # Create a 2x2 grid of subplots.

ax[0,0].plot([1,2,3], [4,5,6])
ax[0,0].set_title("Top Left")

ax[0,1].bar([1,2,3], [3,5,7], color='green')
ax[0,1].set_title("Top Right")

ax[1,0].scatter([1,2,3], [6,5,4], color='red')
ax[1,0].set_title("Bottom Left")

ax[1,1].hist([1,2,2,3,3,3,4], bins=4, color='purple')
ax[1,1].set_title("Bottom Right")

plt.tight_layout()
plt.show()
```



- Sharing Axes: You can share x or y axes across subplots for better comparisons:

Ex:

```
fig, ax = plt.subplots(2, 1, sharex=True)

ax[0].plot([1,2,3], [4,5,6], label="Plot 1")
ax[0].legend()

ax[1].plot([1,2,3], [6,5,4], label="Plot 2", color='orange')
ax[1].legend()
plt.show()
```



7. Saving Plots as Image Files:

- Saving a Plot with savefig(): It allows you to save your figure to a file in different formats like PNG, JPG, PDF, SVG, and more.

Ex: plt.savefig('plot.png')
Automatically saves the plot in the current working directory unless you specify a path. You can specify different formats by changing the file extension (e.g., .jpg, .pdf).

- Saving with Different Formats:

  - Save as PNG: plt.savefig('plot.png')
  - Save as PDF: plt.savefig('plot.pdf')
  - Save as SVG (vector format): plt.savefig('plot.svg')
  - Save as JPG: plt.savefig('plot.jpg')

- Adjusting DPI (Dots Per Inch):

The dpi parameter allows you to control the resolution of the saved plot. Higher DPI values result in higher quality (and larger file size), which is useful for print publications or detailed visualizations.

```
Ex: plt.savefig('high_res_plot.png', dpi=300) #High resolution for printing
    plt.savefig('low_res_plot.png', dpi=72) #Low resolution for web use
```

- Specifying Figure Size: Before saving, you can also control the size of the figure (in inches) using the figsize parameter when creating the figure.

```
Ex: fig, ax = plt.subplots(figsize=(10,6))
    ax.plot([1,2,3], [4,5,6])
    plt.savefig('plot_with_size.png')
```

- Saving Without Displaying:

```
plt.plot([1,2,3], [4,5,6])
plt.savefig('Plot_without_showing.png')
```

- Adjusting Layout Before Saving: plt.tight_layout()

Jupyter Notebook

Jupyter Notebook is a web-based interactive development environment.
It allows you to:

- Write and execute code interactively.
- Document your work with text, images, and equations.
- Visualize data through plots and charts.

Initially developed for Python but now supports over 40 languages, including R, Julia, and JavaScript.

Features:

1. Interactive Coding: Run code and see output instantly in the same interface.
2. Markdown Support: Add formatted text, equations, and images for documentation.
3. Data Visualization: Integrates seamlessly with visualization libraries.
4. Shareable Notebooks: Exports to HTML, PDF, or .ipynb format for collaboration.

Installing: pip install notebook

Starting: jupyter notebook (in terminal)

The command opens a web browser at http://localhost:8888
This is the Jupyter Dashboard, where you can manage your notebooks.

The First Notebook

1. Create a New Notebook:
   From the dashboard, click New → Python 3
2. Structure of a Notebook:
   - Cells: Workspaces for writing code or text.
   - Toolbar: Buttons for saving, adding cells, running code, etc.
   - Output Panel: Displays the result of executed code.
3. Naming the Notebook:
   Click the notebook name (e.g., Untitled) at the top, and rename it.

The Notebook Dashboard

1. File Browser:
   Displays all the files and directories in the current working directory. You can open existing notebooks, create new notebooks, folders or text files and upload files to the directory.

2. Toolbar Buttons:
   - New: Create a new notebook, folder, or terminal.
   - Running: View and manage active notebooks, terminals, and other sessions.
   - Cluster: (Optional) Manage clusters for parallel computing.

3. Kernel:
   A kernel is the computational engine that runs your code. The default kernel is Python. You can install and switch to other kernels, such as R or Julia.

Components of a Jupyter Notebook.

1. The Notebook Header:
   - File Name: The name of the notebook.
   - Kernel Status:
     - Idle: No code is being executed.
     - Busy: Code execution is in progress.
   - Menus:
     - File Menu: Save, export, or close the notebook.
     - Edit Menu: Undo, redo, cut, copy, and paste cells.
     - Kernel Menu: Restart or change the kernel.

2. Notebook Toolbar:
   This toolbar provides shortcuts for commonly used actions:
   - Save (disk icon): Saves the current state of the notebook.
   - Add Cell (+): Inserts a new cell below the current one.
   - Cut, Copy, Paste: Manipulate cells (not text within cells)
   - Run Cell (play icon): Executes the current cell.
   - Stop (square icon): Stops cell execution.
   - Restart Kernel (circle icon): Resets the kernel and clears variables.

3. Notebook Workspace:
   The main area where you write and run code or add documentation.
   - Divided into cells:
     - Code Cells: For executing Python code.
     - Markdown cells: For writing text, formatting, and adding equations.
     - Raw Cells: For content not to be executed (rarely used)

Types of Cells and Their Uses

1. Code Cells:
   Used for writing and running code.
   Output (e.g., printed results, graphs) is displayed directly below the cell.

2. Markdown Cells:
   Used for documentation and formatting.
   Support Markdown syntax for styling:
   - Headers:
     # Header 1
     ## Header 2
     ### Header 3
   - Bold and Italic:
     bold text → bold text
     * italic text** → italic text
   - Lists:
     - Unordered:
       - Item 1
       - Item 2
     - Ordered:
       1. Item1
       2. Item 2
   - Links:
     [Open AI](https://www.openai.com)
   - Images:
     ![Alt Text](image_url)
   - Equations: Use LaTex syntax enclosed in $ for inline equations or $$ for Block equations:

     $$
     E = mc^2
     $$

3. Raw Cells:
   Stores content that isn't processed by the notebook.
   Rarely used in standard workflows.

Working with Cells

Running a Cell

Running All Cells: Cell menu → Run All

- Shift + Enter: Runs the current cell and moves to the next one.
- Ctrl + Enter: Runs the current cell and stays in it.

Adding and Deleting Cells

- Add a new cell:
  - Toolbar: Click the + button.
  - Keyboard Shortcut: Press A (above) or B (below).
- Delete a cell:
  - Toolbar: Use the scissors icon.
  - Keyboard Shortcut: Press D, D (double press D)

Reordering Cells:

Click and drag cells up or down to rearrange them.

Changing Cell Type:

Use the dropdown in the toolbar to switch between:

- Code (Press Y)
- Markdown (Press M)
- Raw

Saving and Autosave

Manual Save: Click the disk icon or press ctrl + S.

Autosave: Jupyter automatically saves your progress periodically.
Saved files are stored with the .ipynb extension.

۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۱۰ ۱۱ ۱۲ ۱۳ ۱۴ ۱۵ ۱۶ ۱۷ ۱۸ ۱۹ ۲۰ ۲۱ ۲۲ ۲۳ ۲۴ ۲۵ ۲۶ ۲۷ ۲۸ ۲۹ ۳۰ ۳۱

Advanced Cell Features

1. Splitting and Merging Cells:

- Splitting a Cell: Highlight a section of code or text within a cell, then split it.
  Shortcut: Place the cursor where you want the split and press: Ctrl + Shift + -.
- Merging Cells: Select multiple cells (Shift + Click for range or Ctrl + Click for specific ones), then merge them:
  Toolbar: Edit → Merge Cells.

2. Keyboard Shortcuts (Beyond the Basics):

Jupyter has two modes: Command Mode (Blue border) and Edit Mode (Green border).

- Command Mode:
  - Z: Undo last cell deletion
  - Shift + M: Merge selected cells
  - H: Display a list of all shortcuts.
- Edit Mode:
  - Ctrl + /: Comment/uncomment the selected lines in a Code cell.
  - Tab: Autocomplete code or show parameter hints.

3. Hidden Features in Markdown:

- Tables:

```
| Header 1 | Header 2 |
|----------|----------|
| Row 1    | Data 1   |
| Row 2    | Data 2   |
```

- Horizontal Lines:

⇒ ---

- Inline HTML: You can embed HTML for advanced styling:

⇒ <div style = "color :blue; font_weight :bold;> This is a blue text in Markdown.

4. Running Shell Commands from Cells:

Use the ! prefix to run shell commands directly from a code cell.

Ex:

⇒ !ls # List files in the current directory (Linux/Mac)

⇒ !dir # For Windows.

5. Using Magic Commands:

Magic commands are Jupyter_specific commands for enhanced functionality. They start with % (line magic) or %% (cell magic)

- Line Magic Commands:
  - % Time : Measures execution time of a single statement.
    ⇒ %time sum(range(1000000))
  - % Who : Lists all variables in the current notebook session.
    ⇒ %who
  - %pwd: Displays the current working directory.
    ⇒ %pwd
- Cell Magic Commands:
  - %%timeit : Measures average execution time of a block of code.

⇒
```
%%timeit
total = 0
for i in range(1000):
    total += i
```

- %%writefile : Writes the cell content to a file.
  ⇒ %%writefile example.py
  print("This is written to a file")

6. Interactive Widgets:

Jupyter supports interactive widgets for dynamic visualizations:

- Install the widgets library:
  pip install ipywidgets
- Import and use widgets:
  from ipywidgets import interact

```
def square(x):
    return x**2

interact(square, x=(0, 10));
```

7. Debugging in Cells:

Use the %debug magic to debug code directly:

- Run a cell with an error:

```
x=10
print(y)  # Undefined variable
```

- In the next cell, run %debug to inspect the error:

```
%debug
```

8. Exporting Notebooks:

Share your work with others by exporting it to different formats:

- Toolbar: File → Download as:
  - Notebook (.ipynb): Original format for sharing-editable notebooks.
  - HTML: Creates a static web view of the notebook.
  - PDF: Requires a LaTeX installation.

9. Linking Sections in Notebooks:

Create clickable links within your notebook using Markdown:

- Add an anchor using a Markdown heading:
  - ⇒ `## My Section`
- Link to the Section:
  - ⇒ `[Go to My Section](#my-section)`