PYTHON

Python Introduction

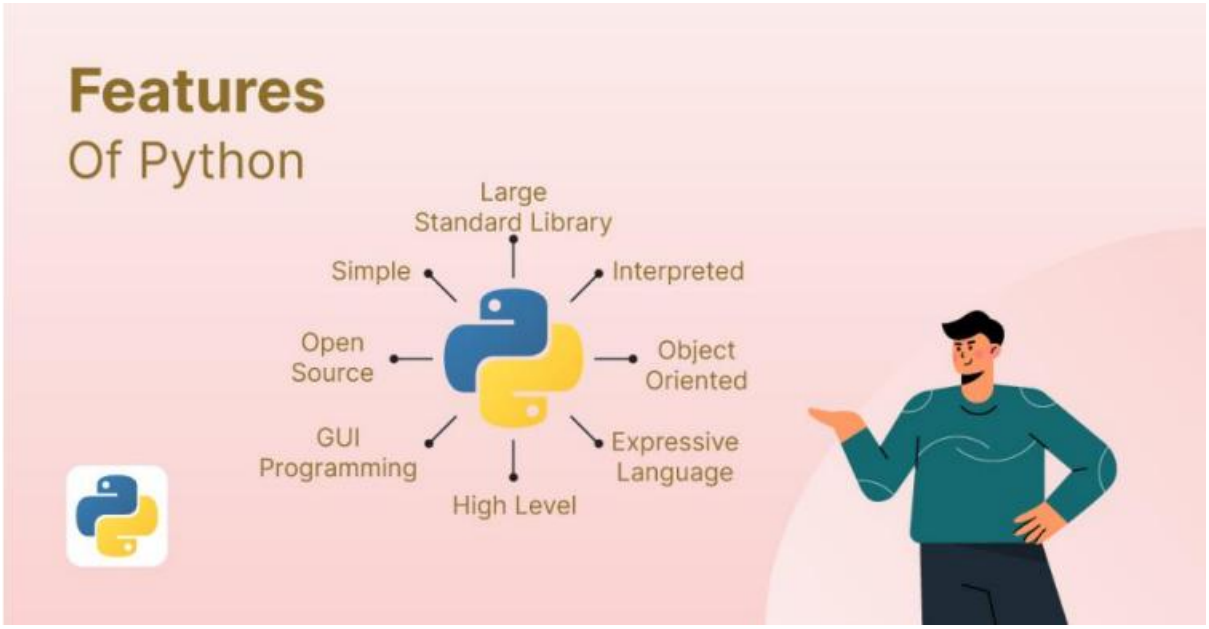
What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

- web development (server-side),
- software development,
- mathematics,
- system scripting.

Python Features Python is a dynamic, high-level, free open source, and interpreted programming language. It supports object-oriented programming as well as procedural-oriented programming. In Python, we don't need to declare the type of variable because it is a dynamically typed language. For example, x = 10 Here, x can be anything such as String, int, etc. In this article we will see what characteristics describe the python programming language.



Features in Python

1. Free and Open Source
2. Easy to code Python is a high-level programming language.
3. Easy to Read As you will see, learning Python is quite simple.
4. Object-Oriented Language One of the key features of Python is Object-Oriented programming.
5. Interpreted Language:

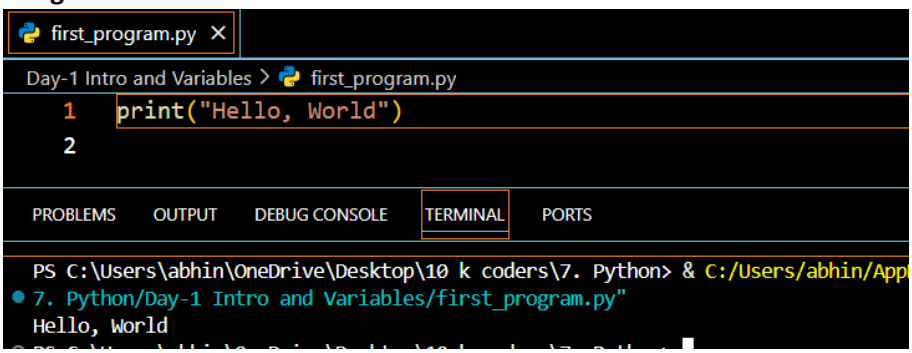
Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Python Syntax compared to other programming languages

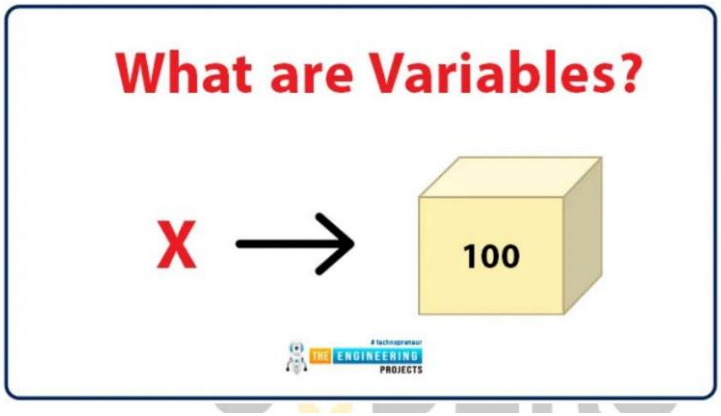
- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

First Program:



Python Variables

Python Variable is containers that store values. Python is not “statically typed”. We do not need to declare variables before using them or declare their type. A variable is created the moment we first assign a value to it. A Python variable is a name given to a memory location. It is the basic unit of storage in a program.



Example of Variable in Python

An Example of a Variable in Python is a representational name that serves as a pointer to an object. Once an object is assigned to a variable, it can be referred to by that name. In layman's terms, we can say that Variable in Python is containers that store values.

var = "10KCoders"

Variables Assignment in Python

Here, we will define a variable in python. Here, clearly we have assigned a number, a floating point number, and a string to a variable such as age, salary, and name.

```
# An integer assignment
age = 45

# A floating point
salary = 1456.8

# A string
name = "John"

print(age)
print(salary)
print(name)
```

Output:

```
45
1456.8
John
```

Declaration and Initialization of Variables

Let's see how to declare a variable and how to define a variable and print the variable.

For example:

```
# declaring the var
Number = 100

# display
print( Number)
```

Output:

```
100
```

Redeclaring variables

in Python We can re-declare the Python variable once we have declared the variable and define variable in python already.

For example:

```
# declaring the var
Number = 100

# display
print("Before declare: ", Number)

# re-declare the var
Number = 120.3

print("After re-declare:", Number)
```

Output:

```
Before declare:  100
After re-declare: 120.3
```

Python Assign Values to Multiple Variables Also,

Python allows assigning a single value to several variables simultaneously with "=" operators.

For example:

```
a = b = c = 10

print(a)
print(b)
print(c)
```

Output:

```
10
10
10
```

Assigning different values to multiple variables

Python allows adding different values in a single line with "," operators.

For example

```
a, b, c = 1, 20.2, "GeeksforGeeks"

print(a)
print(b)
print(c)
```

Output:

```
1
20.2
GeeksforGeeks
```

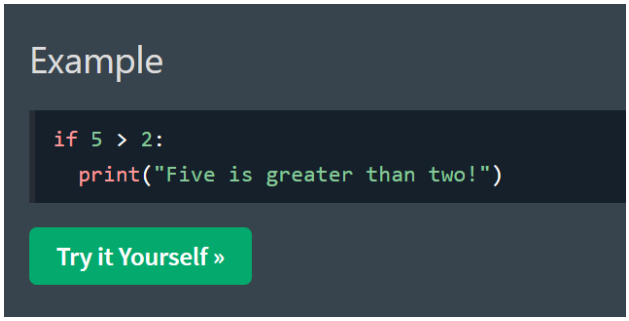
Python Installation:

https://www.tutorialspoint.com/how-to-install-python-in-windows

Python Syntax:

Python Indentation

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.



Python Comments

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.

Creating a Comment

Comments starts with a #, and Python will ignore them:

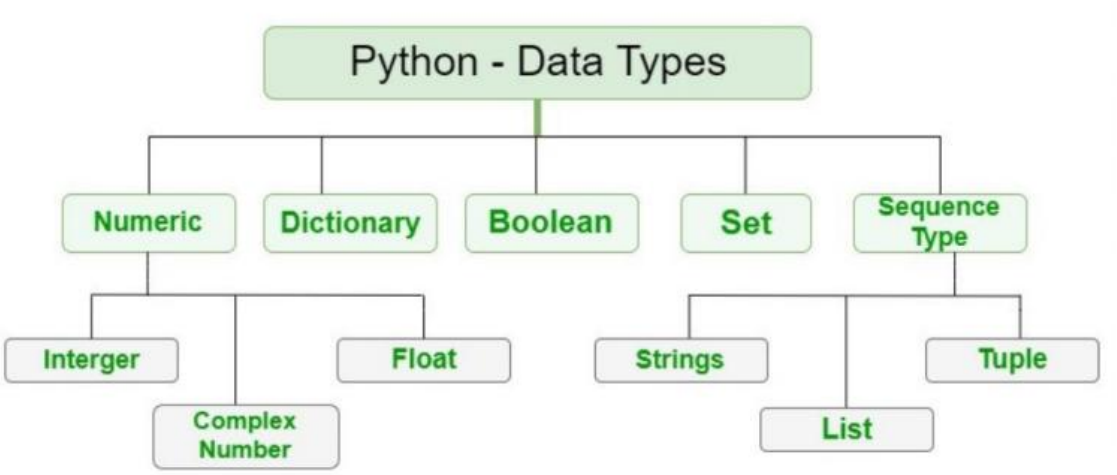
Multiline Comments

Python does not really have a syntax for multiline comments.
To add a multiline comment you could insert a # for each line:

Python Data Types

Python Data Types

Python Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, Python data types are classes and variables are instances (objects) of these classes.



The following are the standard or built-in data types in Python:

- Numeric
- Sequence Type
- Boolean
- Set
- Dictionary

Numeric Data Types in Python

The numeric data type in Python represents the data that has a numeric value. A numeric value can be an integer, a floating number, or even a complex number.

Integers – This value is represented by int class. It contains positive or negative whole numbers (without fractions or decimals). In Python, there is no limit to how long an integer value can be.

Float – This value is represented by the float class. It is a real number with a floating-point representation.

Complex Numbers – A complex number is represented by a complex class. It is specified as (real part) + (imaginary part)j .
For example – 2+3j

Note – type() function is used to determine the type of Python data type.

```
a = 5
print("Type of a: ", type(a))

b = 5.0
print("\nType of b: ", type(b))

c = 2 + 4j
print("\nType of c: ", type(c))
```

```
Type of a:  <class 'int'>
Type of b:  <class 'float'>
Type of c:  <class 'complex'>
```

Sequence Data Types in Python

The sequence Data Type in Python is the ordered collection of similar or different Python data types. Sequences allow storing of multiple values in an organized and efficient fashion. There are several sequence data types of Python:

- Python String
- Python List
- Python Tuple

String Data Type

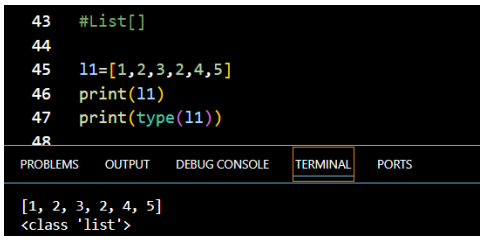
Strings in Python are arrays of bytes representing Unicode characters. A string is a collection of one or more characters put in a single quote, double-quote, or triple-quote.

Example: This Python code showcases various string creation methods. It uses single quotes, double quotes, and triple quotes to create strings with different content and includes a multiline string. The code also demonstrates printing the strings and checking their data types.

List Data Type

Lists are just like arrays, declared in other languages which is an ordered collection of data. It is very flexible as the items in a list do not need to be of the same type.

Creating a List in Python Lists in Python can be created by just placing the sequence inside the square brackets[].




```
List = []
print("Initial blank List: ")
print(List)
List = ['GeeksForGeeks']
print("\nList with the use of String: ")
print(List)
List = ["Geeks", "For", "Geeks"]
print("\nList containing multiple values: ")
print(List[0])
print(List[2])
List = [['Geeks', 'For'], ['Geeks']]
print("\nMulti-Dimensional List: ")
print(List)
```

Output:

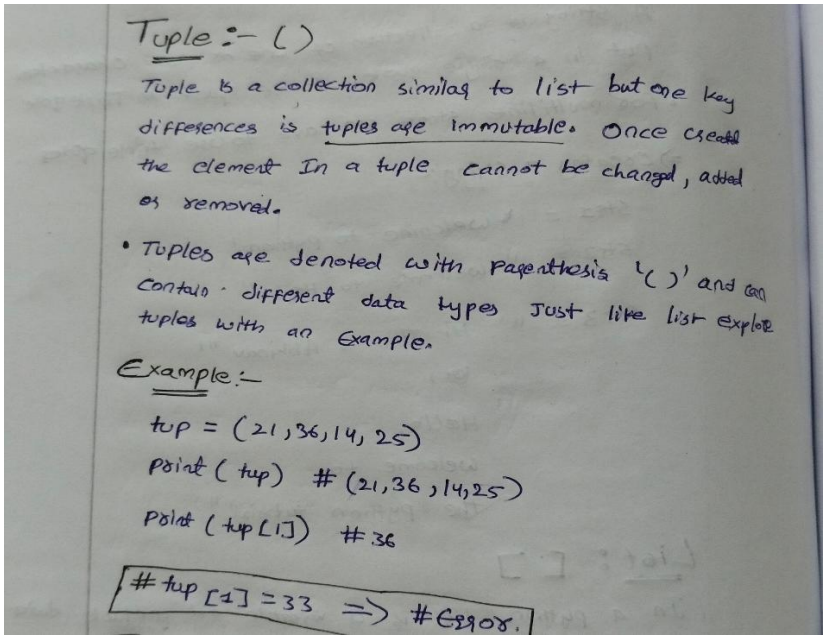
```
Initial blank List:
[]
List with the use of String:
['GeeksForGeeks']
List containing multiple values:
Geeks
Geeks
Multi-Dimensional List:
[['Geeks', 'For'], ['Geeks']]
```

Tuple Data Type

Just like a list, a tuple is also an ordered collection of Python objects. The only difference between a tuple and a list is that tuples are immutable i.e. tuples cannot be modified after it is created. It is represented by a tuple class.

Creating a Tuple in Python

In Python Data Types, tuples are created by placing a sequence of values separated by a ‘comma’ with or without the use of parentheses for grouping the data sequence



Boolean Data Type in Python

Python Data type with one of the two built-in values, True or False. Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false). However non-Boolean objects can be evaluated in a Boolean context as well and determined to be true or false. It is denoted by the class bool.

Note – True and False with capital 'T' and 'F' are valid booleans otherwise python will throw an error.

```
print(type(True))
print(type(False))
print(type(true))
```

Output:

```
<class 'bool'>
<class 'bool'>
```

```
Traceback (most recent call last):
  File "/home/7e8862763fb66153d70824099d4f5fb7.py", line 8, in
    print(type(true))
NameError: name 'true' is not defined
```

Set Data Type in Python

In Python Data Types, a Set is an unordered collection of data types that is iterable, mutable, and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.

Create a Set in Python

Sets can be created by using the built-in set() function with an iterable object or a sequence by placing the sequence inside curly braces, separated by a 'comma'. The type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

```
set1 = set()
print("Initial blank Set: ")
print(set1)
set1 = set("GeeksForGeeks")
print("\nSet with the use of String: ")
print(set1)
set1 = set(["Geeks", "For", "Geeks"])
print("\nSet with the use of List: ")
print(set1)
set1 = set([1, 2, 'Geeks', 4, 'For', 6, 'Geeks'])
print("\nSet with the use of Mixed Values")
print(set1)
```

Output:

```
Initial blank Set:
set()
Set with the use of String:
{'F', 'o', 'G', 's', 'r', 'k', 'e'}
Set with the use of List:
{'Geeks', 'For'}
Set with the use of Mixed Values
{1, 2, 4, 6, 'Geeks', 'For'}
```

Dictionary Data Type in Python

A dictionary in Python is an unordered collection of data values, used to store data values like a map, unlike other Python Data Types that hold only a single value as an element, a Dictionary holds a key: value pair. Key-value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon : , whereas each key is separated by a 'comma'

Create a Dictionary in Python

In Python, a Dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'. Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated and must be immutable.

```
Dict = {}
print("Empty Dictionary: ")
print(Dict)
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print("\nDictionary with the use of Integer Keys: ")
```

Output:

```
Empty Dictionary:
{}
Dictionary with the use of Integer Keys:
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
```

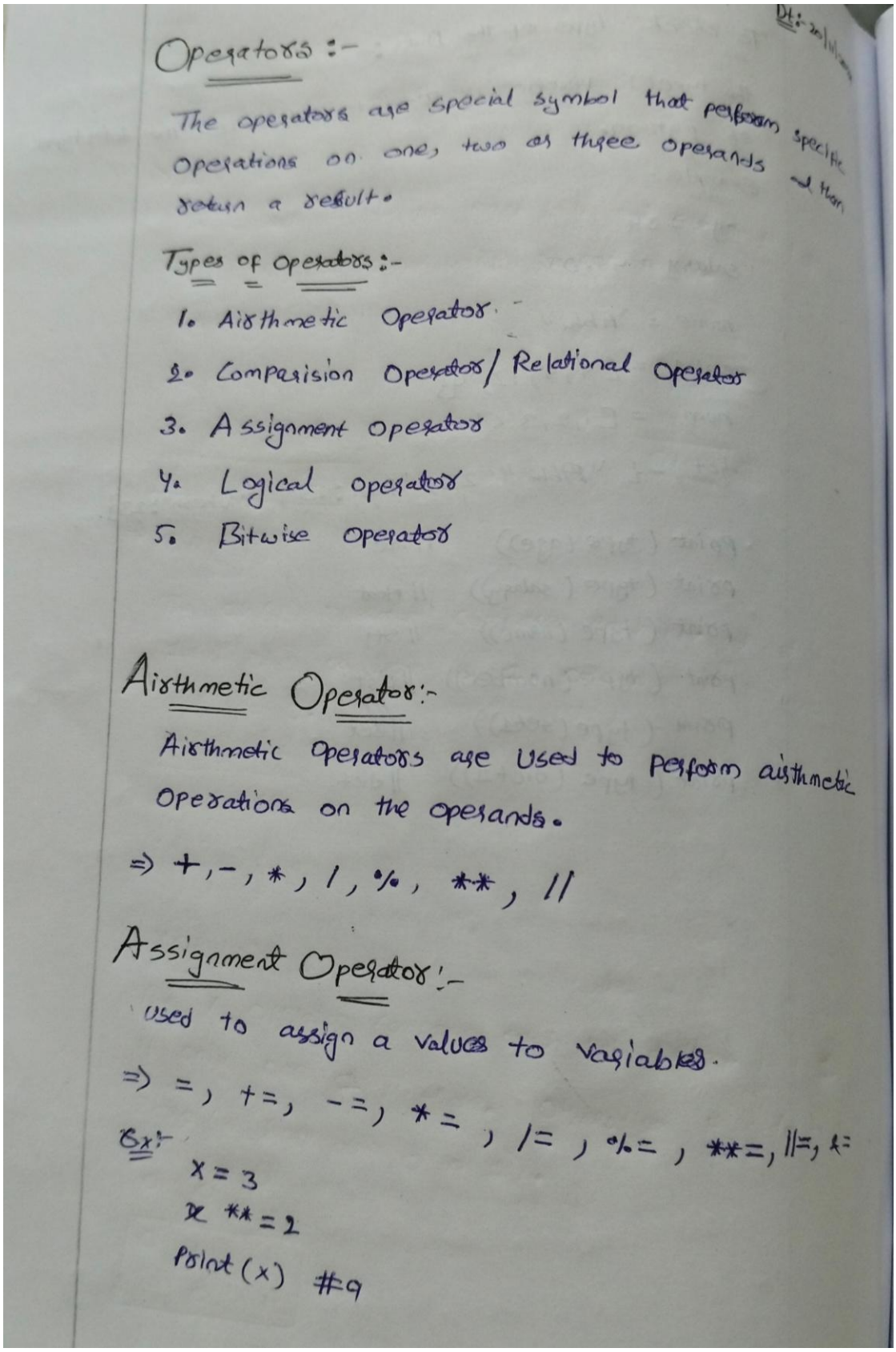
To check type of the Data :-

the type() function is used to determine the data type of python.

Example :-

```
age = 34
salary = 45.00
name = "Abhi"
set1 = {4, 5, 12, 4}
num = [1, 2, 3, 6]
dict = {"Abhi": 23, "Ravi": 23, "Mukesh": 23}
print (type (age))          // int
print (type (salary))       // float
print (type (name))         // str
print (type (numbers))      // List
print (type (set1))         // set
print (type (dict1))        // dict.
```

Operators



Operators :-

The operators are special symbol that perform specific operations on one, two or three operands and then return a result.

Types of operators :-

1. Airthmetic Operator
2. Comparision Operator / Relational Operator
3. Assignment operator
4. Logical operator
5. Bitwise operator

Airthmetic Operator :-

Airthmetic Operators are used to perform airthmetic Operations on the operands.

⇒ +, -, *, /, %, **, //

Assignment Operator :-

used to assign a values to variables.

⇒ =, +=, -=, *=, /=, %=, **=, //=, &=

Ex :-

```
x = 3
x **= 2
Print(x)  #9
```

Relational / comparision Operators:-

- Used to compare values. They return a boolean value True or False.
- ==, >, <, !=, >=, <=

Logical Operator:-

Logical operator returns a boolean value by Evaluating boolean Expression.

→ and
→ or
→ not.

Logical And:-

			o/p
⇒	true	true	true
⇒	true	False	False
⇒	False	true	false
⇒	false	false	false

Logical OR:-

			o/p
⇒	true	true	true
⇒	true	false	true
⇒	false	true	true
⇒	false	false	false.

Logical Not:-

	o/p
true	False
False	true

Bitwise Operator:-

Bitwise operators are used to compare (binary) numbers.

→ AND (&)

The operator compare each bit and set it to 1 if both are 1 otherwise it set to 0.

		o/p
0	0	0
0	1	0
1	0	0
1	1	1

Ex:- 6&3

6 = 110

3 = 011

010 → (2)

⇒ OR (|)

The OR (|) operator compares each bit and set it to 1 if one or both is 1. Otherwise it returns 0 set to 0.

		o/p
0	0	0
0	1	1
1	0	1
1	1	1

⇒ XOR (^)

The ^ operator compares each bit and set it to 1 if only one '1' is there. Otherwise it sets to zero (if both are 0 or 1).

		o/p
0	0	0
0	1	1
1	0	1
1	1	0

NOT (~)

The ~ operator inverts each bit (0 becomes 1 and 1 becomes 0)

	o/p
0	1
1	0

Ex:- 3 becomes -4

~~3 = 0 0 1 1~~

Left shift (<<)

The << operator inserts the specified number of 0's from the right and let the same amount of leftmost bits fall off.

Ex:- 3<<2

3 = 0000000000000011

12 = 0000000000001100

Right shift (>>) :-

The >> operator moves each bit the specified number of times to the right. Empty holes at the left are filled with 0's.

Ex:- 8 = 0000 0000 0000 1000

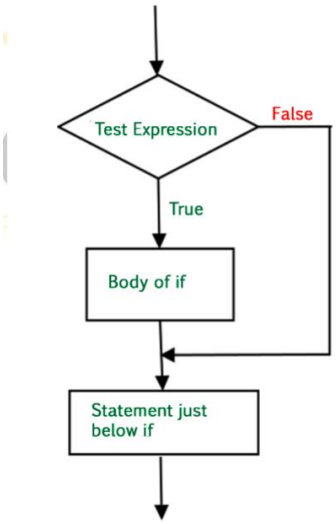
2 = 0000 0000 0000 0010

Conditional statements

If Statement:

The if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not.

Flowchart of If Statement Let's look at the flow of code in the Python If statements



Syntax of If Statement:

```
if condition:
   # Statements to execute if
   # condition is true
```

Example:

```
# if statement example
if 10 > 5:
   print("10 greater than 5")
print("Program ended")
```

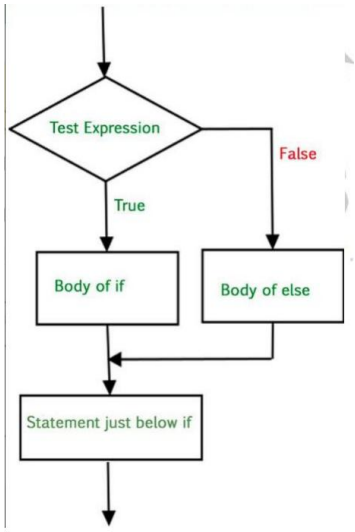
Output:

```
10 greater than 5
Program ended
```

Python If Else Statement

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But if we want to do something else if the condition is false, we can use the else statement with the if statement Python to execute a block of code when the Python if condition is false

Flowchart of If Else Statement Let's look at the flow of code in an if else Python statement.



Syntax of Python If-Else:

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```

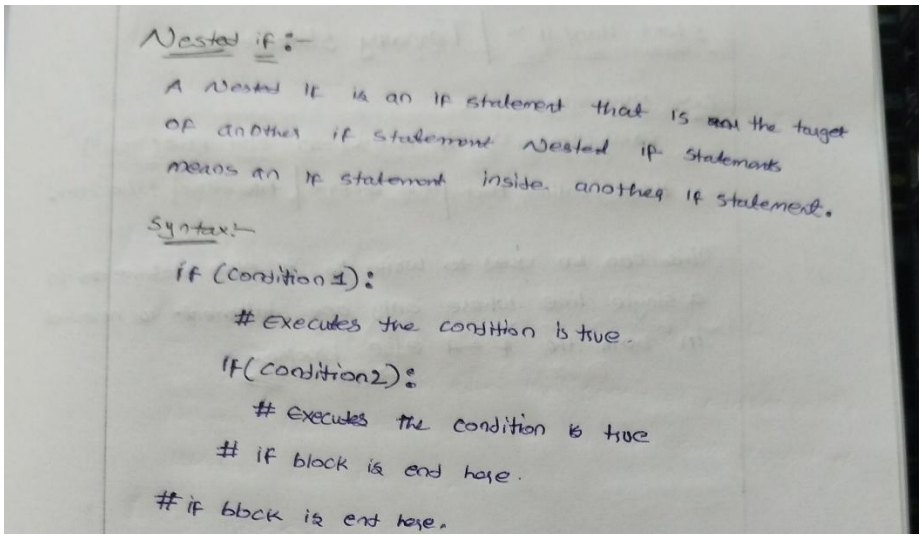
Example:

```
# if..else statement example
x = 3
if x == 4:
    print("Yes")
else:
    print("No")
```

Output: No

Python Nested If Statement

A nested if is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement.



Elif statement :-

Here, a uses can decide among multiple options. The if statements are executed from the top down.

Syntax :-

```
if (condition):
    statement
elif (condition):
    statement
    .
    .
    .
else:
    statement
```

Note :- if statement cannot be empty use pass statement to avoid getting an error.

```
if (condition):
    pass
```

Short Hand if :- / Ternary statement

Ex :- num = 11

print ("even") if num%2==0 else print("odd")

true condition block / true condition / false condition / false block.

This can be used to write the if-else statements in a single line whose only one statement is needed in both the if and else blocks.

LOOPS

For loop:-

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set or a string).

- This is less like the for keyword in other programming language, and works more like an iterator method as found in other object oriented programming languages.

Ex:-
```
# print each fruit in a fruit list
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

Ex:-
```
#Looping through a string
for x in "banana":
    print(x)
```

⇒ **Break:-** with the break statement we can stop the loop before it has looped through all the items.

Ex:-
```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
    if x == "banana":
        break
```

⇒ **Continue:-** it can stop the current iteration of the loop, and continue with the next.

Ex:-
```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

The range( ) function:-

The range function returns a sequence of numbers, starting from 0 by default increments by 1 (by default), and ends at a specified number.

Examples:-

1.
```
for x in range (6):
    Print (x)
```
↳ end number

O/P: 0, 1, 2, 3, 4, 5

Example-2:-

```
for x in range(2, 6):
    Print(x)
```
↓ start ↳ end.

Example-3:-

```
for x in range (3, 30, 2):
    print(x)
```
↓ start ↓ end ↓ Increment.

Nested Loops:-

- A nested loop is a loop inside a loop.
- The inner loop will be executed one time for each iteration of the "outer loop".

Example:-

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
    for y in fruits:
        print (x, y)
```

While loop:-

While loop is used to execute a block of statements repeatedly until a given condition is satisfied. When the condition become false, the line immediately after the loop in the program is executed.

Syntax:-

```
While Expression:
    Statement (s)
```

Example:-

```
count = 0
while (count < 3):
    count = count + 1
    Print ("Hello")
```

Example:-

```
for i in range (1, 6, 1):    # 1 2 3 4 5
    for j in range (1, 11, 1): # 1 2 3 4 5 6 7 8 9 10
        Print (i, "x", j, "=", i * j)
```

Formatted String:-

```
name = "Abhi"
Print ("hello {x}")    # hello Abhi
```

↓

formated string

Functions

Python Functions

Python Functions is a block of statements that return the specific task. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.

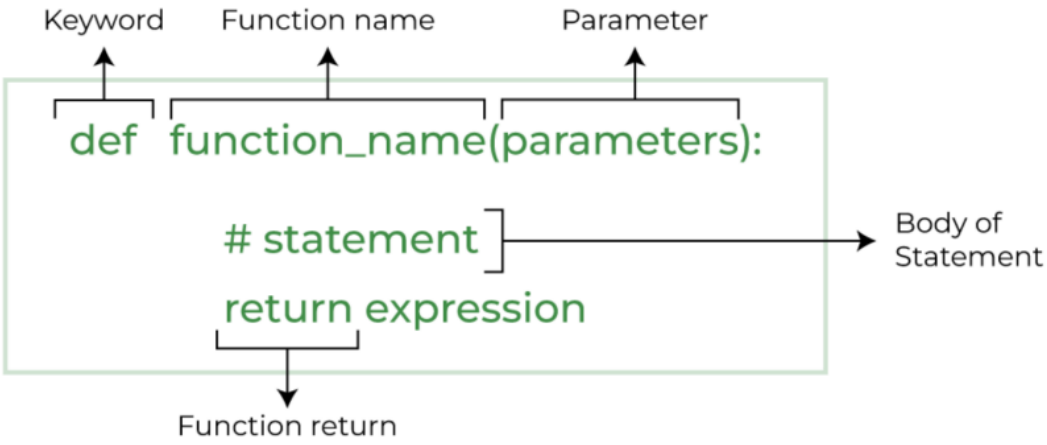
Some Benefits of Using Functions

Increase Code Readability

Increase Code Reusability

Python Function Declaration

The syntax to declare a function is:



Types of Functions in Python

• Below are the different types of functions in Python:

Built-in library function: These are Standard functions in Python that are available to use.

User-defined function: We can create our own functions based on our requirements.

Creating a Function in Python

We can define a function in Python, using the def keyword. We can add any type of functionalities and properties to it as we require. By the following example, we can understand how to write a function in Python. In this way we can create Python function definition by using def keyword.

Syntax:

```
# A simple Python function
def fun():
    print("Welcome to GFG")
```

Calling a Function in Python

After creating a function in Python we can call it by using the name of the functions Python followed by parenthesis containing parameters of that particular function. Below is the example for calling def function Python.

Example:

```
# A simple Python function
def fun():
   print("Welcome to GFG")

# Driver code to call a function
fun()
```

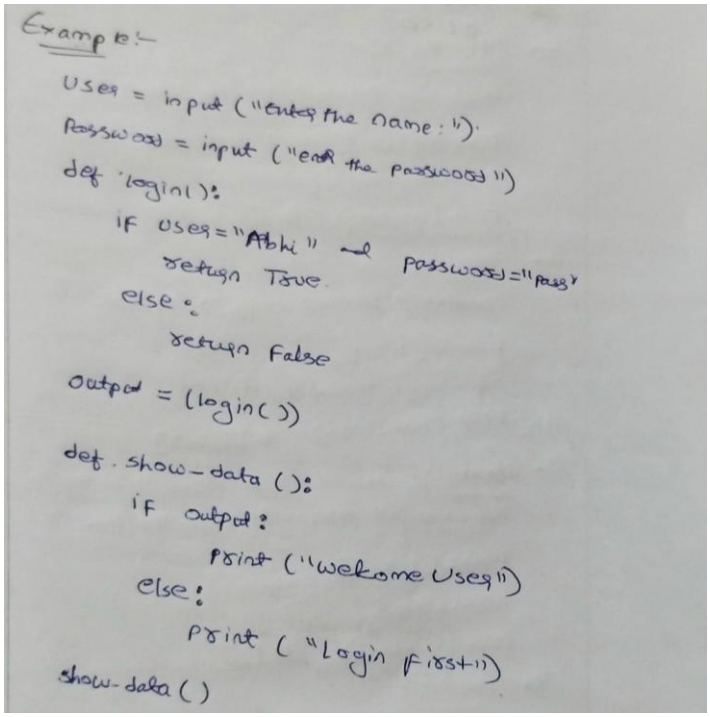
Python Function Arguments

Arguments are the values passed inside the parenthesis of the function. A function can have any number of arguments separated by a comma.

In this example, we will create a simple function in Python to check whether the number passed as an argument to the function is even or odd.

Example:

```
# A simple Python function to check
# whether x is even or odd
def evenOdd(x):
   if (x % 2 == 0):
      print("even")
   else:
      print("odd")

# Driver code to call the function
evenOdd(2)
evenOdd(3)
```



Python Default arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument. The following example illustrates Default arguments.

Example: We call myFun() with the only argument.

```
# Python program to demonstrate
# default arguments
def myFun(x, y = 50):
    print("x: ", x)
    print("y: ", y)

# Driver code
myFun(10)
```

Output:

```
x:  10
y:  50
```

Taking input in Python

input () function first takes the input from the user and converts it into a string. The type of the returned object always will be <class 'str'>. It does not evaluate the expression it just returns the complete statement as String.

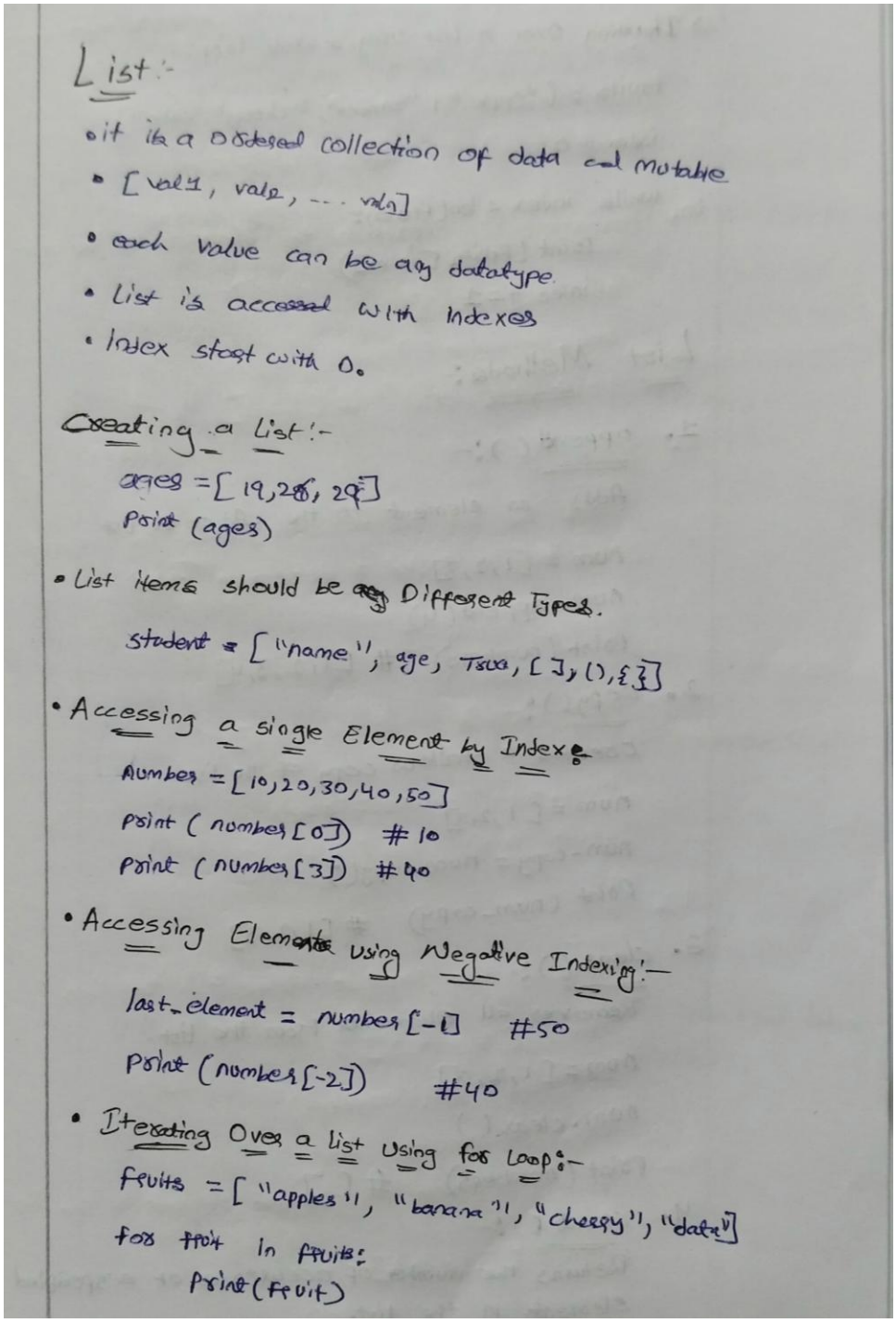
Example-1:

```
# Python program showing
# a use of input()
val = input("Enter your value: ")
print(val)
```

Example-2:

```
username = input("Enter username:")
print("Username is: " + username)
```

List Methods



List :-

- it is a Ordered collection of data and mutable
- [val1, val2, ... valn]
- each value can be any datatype
- List is accessed with indexes
- Index start with 0.

Creating a List :-

```
ages = [19, 28, 29]
print (ages)
```

- List items should be Different Types.

```
student = ["name", age, True, [], (), {}]
```

- Accessing a single Element by Index :-

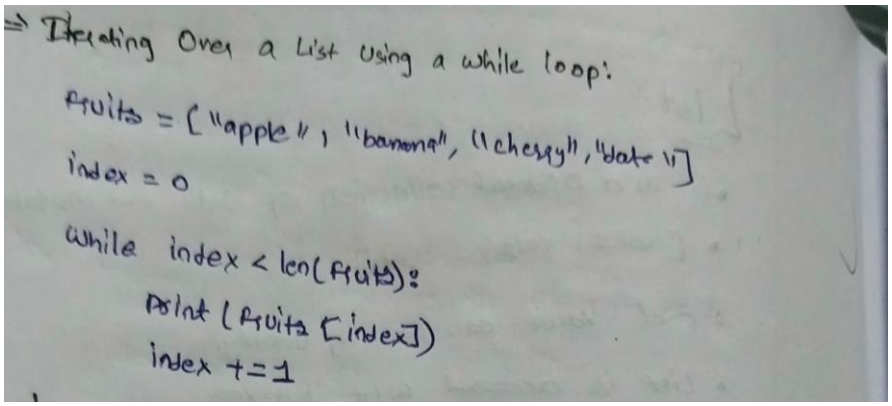
```
numbers = [10, 20, 30, 40, 50]
print (numbers[0])   # 10
print (numbers[3])   # 40
```

- Accessing Elements using Negative Indexing :-

```
last_element = numbers[-1]   #50
print (numbers[-2])          #40
```

- Iterating Over a list using for loop :-

```
fruits = ["apples", "banana", "cherry", "date"]
for fruit in fruits:
    print(fruit)
```



⇒ Iterating Over a List using a while loop :

```
fruits = ["apple", "banana", "cherry", "date"]
index = 0

while index < len(fruits):
    print (fruits[index])
    index += 1
```

List Methods:-

1. append():-

Adds an element to the end of the list

```
num = [1,2,3]
num.append(4)
print(numbers)  # [1,2,3,4]
```

2. copy():-

Creates a shallow copy of the list.

```
num = [1,2,3]
num_copy = num.copy()
print(num_copy)  # [1,2,3]
```

3. clear():-

Removes all elements from the list.

```
num = [1,2,3]
num.clear()
print(numbers)  # []
```

4. count():-

Returns the number of occurences of a specified element in the list.

```
num = [1,2,2,3]
count_of_two = num.count(2)
print(count_of_two)  #2
```

5. Extend() :-

Adds elements of an iterable (like another list) to the end of the list.

```
num = [1,2,3]
num.extend([4,5])
print(numbers)  # [1,2,3,4,5]
```

6. index() :-

Returns the index of the first occurrence of a specified element.

```
num = [1,2,3,2]
index_of_two = numbers.index(2)
print(index_of_two)  #1
```

7. Insert() :-

Inserts an element at a specific position in the list

```
num = [1,2,3]
num.insert(2,3)
print(num)  # [1,2,3,4]
```

8. pop() :-

Removes and returns the element at the specified index (or the last element if no index is specified).

```
num = [1,2,3]
last_ele = num.pop()
Print(last_ele)  #3
Print(num)  # [1,2]
```

9. remove( ) :- removes the first occurrence of a specified element.

```
Num = [1,2,3,2]
num.remove(2)
Print (num)        # [1,3,2]
```

10. reverse( ) :- Reverses the elements of the list.

```
num = [1,2,3]
num.reverse( )
Print (numbers)   # [3,2,1]
```

11. Sort( ) :-

sorts the list in ascending order by default (can also be sorted in descending order)

```
num = [3,1,4,2]
num = sort( )
Print (num)        # [1,2,3,4]

num.sort (reverse=True)
Print (num)        # [4,3,2,1]
```

12. min( ) :-

Returns the smallest element in the list.

```
num = [3,1,4,2]
minimum = min (numbers)
Print (minimum)   #1
```

13. Max( ) :- Returns the largest element in the list

```
num = [3,1,4,2]
maximum = max (numbers)
Print (maximum)   #4
```

List Comprehension:-

List comprehension provides a concise way to create lists in python. It's a shorthand for looping through an iterable and applying an expression to each item.

Example:- Creates a list of squares.

```
squares = [x**2 for x in range(5)]
print(squares)  #[0,1,4,9,16]
```

Example:- Create a list of even numbers.

```
even = [x for x in range(10) if x%2==0]
print(evens)  #[0,2,4,6,8]
```

Nested Lists:-

A nested list is a list that contains other lists as its elements. You can access element of a nested list using indexing.

Example:- simple nested list

```
matrix = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
    ]
print(matrix[1][2])  #6
```

Example:- Iterates through a nested list

```
for row in matrix:
    for element in row:
        Print(element, end=" ")
    Print()
```

O/P:-

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Tuple methods

Tuples:-

- Tuple is similar to list (It is ordered collection of objects) but main difference is immutable.

⇒ creating a tuple:-

```
tup1 = (2, "Hello", "Python")
print (tup1)
```

⇒ Tuple characteristics:-

- ordered
- immutable
- Allow duplicates.

⇒ Access Tuple Items:-

- Index value starts with zero (0)

Ex:-

```
Lang = ("Python", 'C', "C++")
Print (lang[0])  #Python
Print (lang[2])  #C++
```

⇒ Tuples cannot be modified:-

It will get an error.

```
Lang[2] = "Java";  # error
```

⇒ Iterate through a tuple:-

```
fruits = ("apple", "banana", "orange")
for fruit in fruits:
    print(fruit).
```

Python Tuple Methods :-

1. count() :- it returns the number of times the specified element appears in the tuple

Ex :-

```
vowels = ('a', 'e', 'i', 'o', 'i', 'u')
count = vowels.count('i')
print(count)  #2
```

2. Index() :- returns the index of specified element in the tuple.

Ex :-

```
index = vowels.index('e')
print(index)  #1
```

Nested Tuples :-

Nested tuples are tuples that contain other tuples as their element. They can be thought as a multidimensional tuples.

Characteristics of Nested Tuples :-

- Immutability
- Accessing Elements
- Fixed structure.

Ex :-

```
nested = (
        (1, 2, 3),
        (4, 5, 6),
        (7, 8, 9)
    )

#Accessing Nested Tuples.
print(nested[0][0])  #1
```

Find a num in Matrix :-

```
matrix = [(1,2,3),(4,5,6),(2,8,9)]
num = 7
exists = False
for i in matrix:
    for j in i:
        if (j == num):
            exists = True

if (exists):
    print("True")
else:
    print("False")
```

Dictionary Methods

Dictionagi

- Stores the element in key value pairs.
- they are enclosed in {} curly braces / Hauge braces.
- key should be always string
- we can store multiple data types in dict.

Example :-

```
thisdict = {
        "brand": "Ford",
        "model": "Mustang",
        "year": 1964
    }
print(thisdict)
```

=> Dictionary Item :-

- dictionary items are ordered, changable, and do not allow duplicates.

Ex:-

```
thisdict = {
        "brand" : "Ford",
        "Model" : "Mustang",
        "year" : 1964
    }
print ( thisdict ["brand"])
```

O/P :- Ford.

=> changable

```
thisdict ["year"] = 1965
```

=> Duplicates not allowed

It automatically removes past element.

=> Dictionary length:-

```
print (len (thisdict))   #3
```

=> Accessing Items :-

```
thisdict = {
        "brand" : "Ford",
        "Model" : "Mustang",
        "year" : "1964"
    }
x = thisdict ["Model"]
(or)
x = thisdict.get ("Model")
# Mustang
```

=> Get keys :-

it returns list of keys in the dictionary

Ex:- x = thisdict.keys()

dict_keys (['brand', 'Model', 'year'])

=> Get values :-

It returns a list of all the values in the dictionary

Ex:-

x = thisdict.values()

dict_values (['Ford', 'Mustang', 1964])

=> Get Items :-

It returns each item in a dictionary, as tuples in a list.

Ex:-

x = thisdict.items()

dict_items ([('Brand', 'Ford'), ('Model', 'Mustang'), ('year', 1964)])

=> Update Ditionary :-

The update() method will update the dictionary with the items from the given argument.

Ex:-

```
thisdict = {
        "brand": "Ford",
        "Model": "Mustang",
        "year": 1964
    }
this dict.update ({"year": 2020})
```

Adding Items:-

Ex:- thisdict["color"] = "Red"

Removing Items:-

They are several methods to remove items from a dictionary.

Ex-1:- thisdict.pop("Model")

Ex-2:- thisdict.popItem("Model")

Ex-3:- del thisdict["Model"]

Ex-4:- del thisdict #delete the dictionary completely.

clear():- method empties the dictionary

Ex:- thisdict.clear()

{}

Loop through a dictionary:-

- you can loop through a dictionary by using a for loop.
- when looping through a dictionary, the default value are the keys of the dictionary, but there are methods to return values as well.

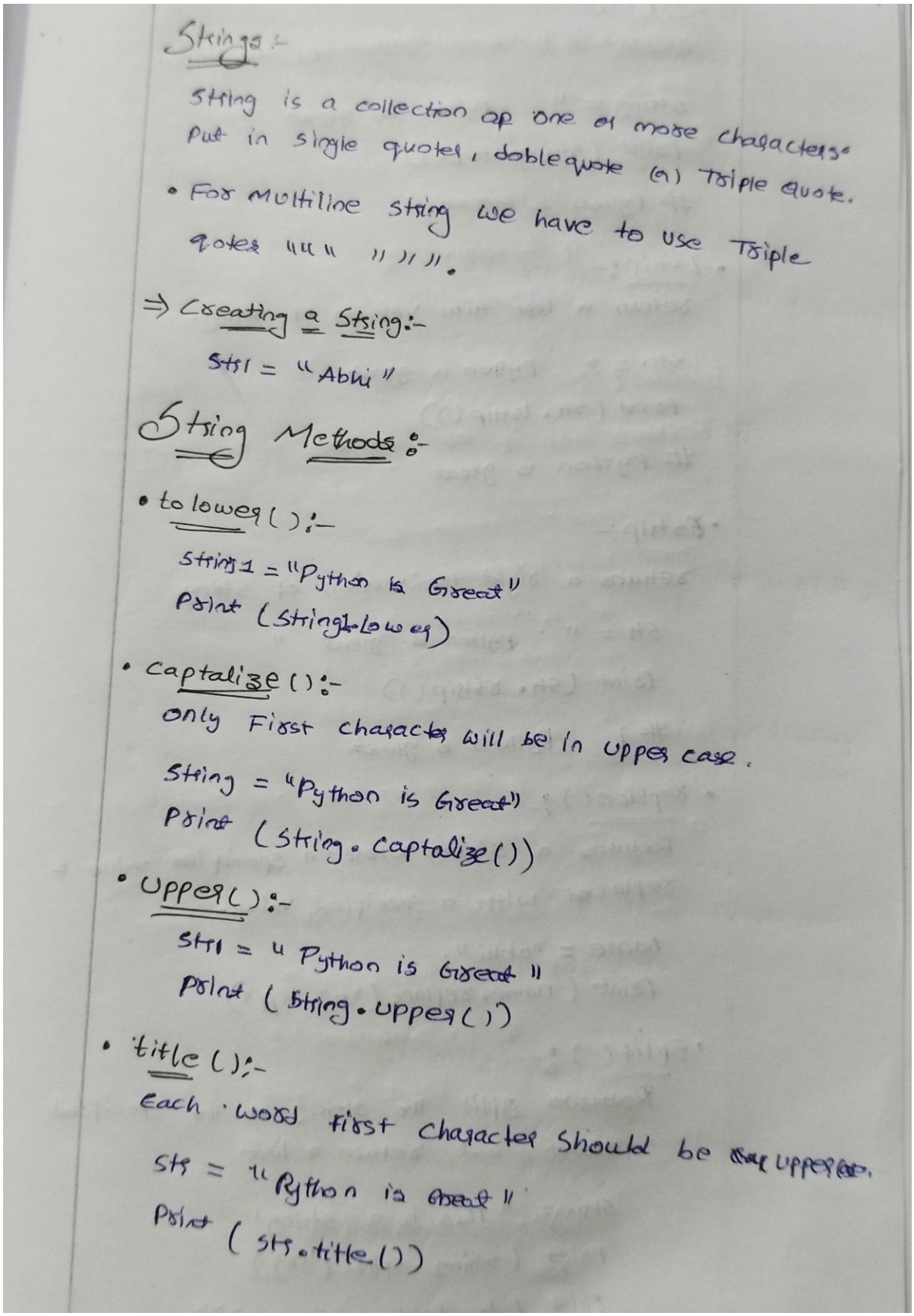
Ex:-

```
for x in thisdict:
    print(x)
    # Brand Model Year
```

Ex-2:-

```
for x in thisdict:
    print(thisdict[x])
# Ford Mustang 1964
```

String and string methods:



Strings :-

String is a collection of one or more characters put in single quotes, double quote (or) Triple quote.

- For Multiline string we have to use Triple quotes """ """ """.

⇒ Creating a String :-

```
Str1 = "Abhi"
```

String Methods :-

- to lower() :-

```
string1 = "Python is Great"
Print (string1.lower)
```

- captalize() :-

only First character will be in upper case.

```
String = "python is Great"
Print (String.captalize())
```

- Upper() :-

```
Str1 = " Python is Great "
Print (String.upper())
```

- title() :-

Each word first character should be upper.

```
Str = " Python is Great "
Print (Str.title())
```

- strip():-
returns a trimmed version of the string.

```
string = "    Python is great   "
print (string.strip())
# Python is Great.
```

- lstrip:-
return a left trim version of string.

```
str = "    python is great   "
print (str.lstrip())
# Python is great   "
```

- rstrip:-
returns a right trim version of string

```
str = "   Python is great   "
print (str.rstrip())
#    Python is great.
```

- replace():-
Returns a string where a specified value is replaced with a specified value.

```
name = "abhi";
print ( name.replace ("a", "A"))
```

- Split():-
~~Returns~~ split the string at the specified seperator, and return a list.

```
string= " This is a string"
print ( string .split (" "))
print ( len (string .split(" ")))
```

join():-

Joins the elements of an iterable to the end of the string.

```
String = "This is a string"
Print ("-".join(string.split(" ")))
```

Find():-

Searches the string for a specified value and returns the position of where it was found.

returns the first occurence of the index.

Ex:-

```
String = "this is string"
Print (string.Find("t"))   #0
```

Index():-

Searches the string for a specified value and returns the position of where it was found.

Ex:-

```
string = "String"
Print (string.index("s"))
```

startswith():- Returns true if the string starts with the specified value.

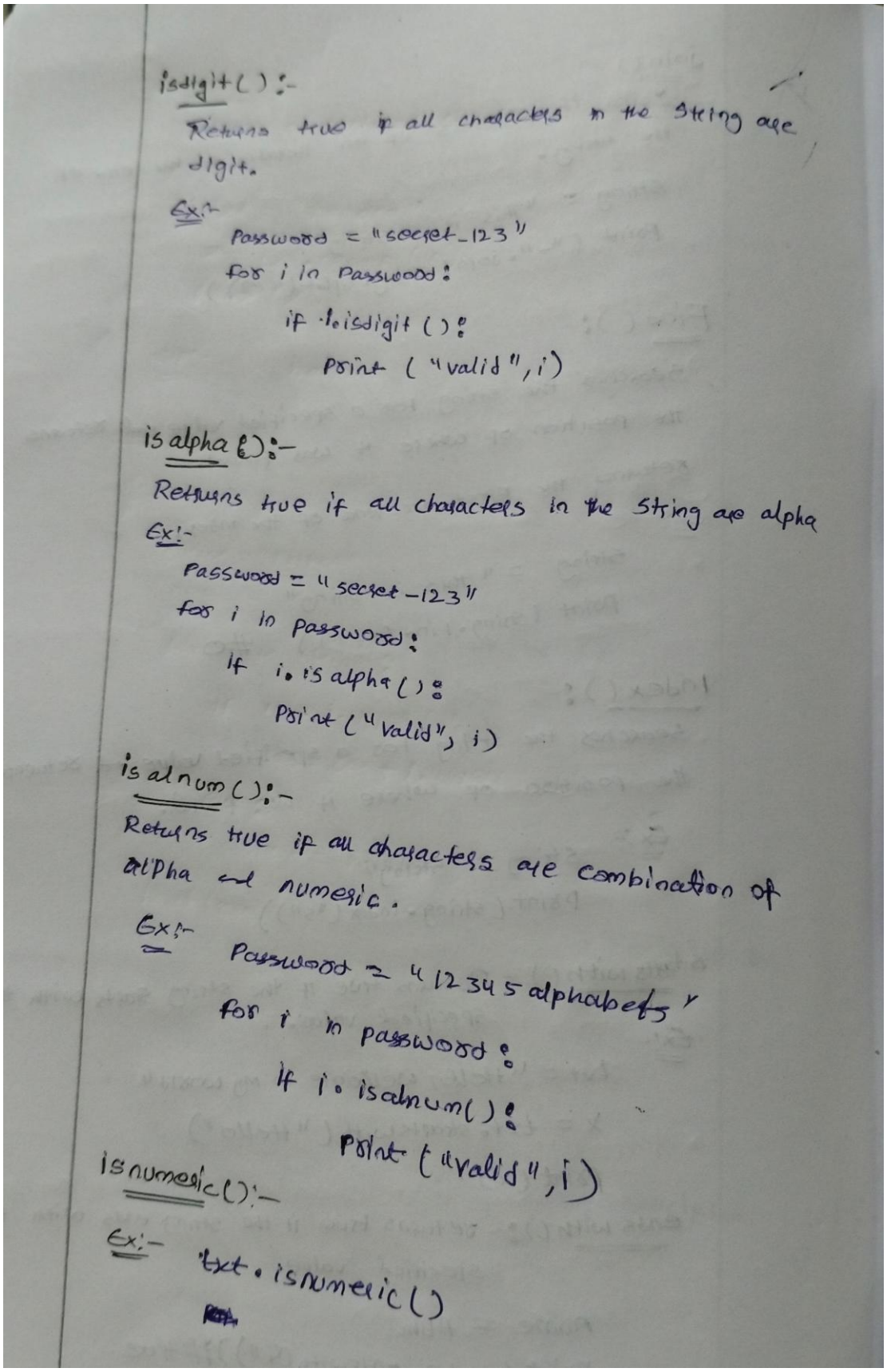
Ex:-

```
txt = "Hello, welcome to my world"
x = txt.startswith("Hello")
print (x)
```

endswith():- returns true if the string ends with the specified value.

```
name = Abhi
Print (name.endswith("i"))  #true.
```

The isalpha() method returns True if all the characters are alphabet letters (a-z).



isdigit():-

Returns true if all characters in the string are digit.

Ex:-

```
Password = "secret_123"
for i in Password:
    if i.isdigit():
        print ("valid", i)
```

isalpha():-

Returns true if all characters in the string are alpha

Ex:-

```
Password = "secret_123"
for i in password:
    if i.isalpha():
        print ("valid", i)
```

isalnum():-

Returns true if all characters are combination of alpha and numeric.

Ex:-

```
Password = "12345 alphabets"
for i in password:
    if i.isalnum():
        print ("valid", i)
```

isnumeric():-

Ex:-

```
txt.isnumeric()
```

isspace():-

return true if all characters in the string are whitespace.

Ex:-

```
txt = "   "
x = txt.isspace()
print(x).
```

swapcase:-

swapcase, lowercase become uppercase and vice versa.

Ex:-

```
name = "Abhi"
print(name.swapcase())
# aBHI
```

Zfill():-

Fills the string with a specified number of 0 values at the beginning.

Ex:-

```
txt = "50"
x = txt.zfill(10)
print(x)
# 0000000050
```

rjust:-

Returns a right justified version of a string.

Ex:-

```
txt = "banana"
x = txt.rjust(20)
print(x)  # banana.
```

ljust:- Returns a left justified version of a string.

Ex:-

```
txt = "banana"
x = txt.ljust(10)
print(x)  # banana
```

center():-

Returns a centered string.

Ex:-

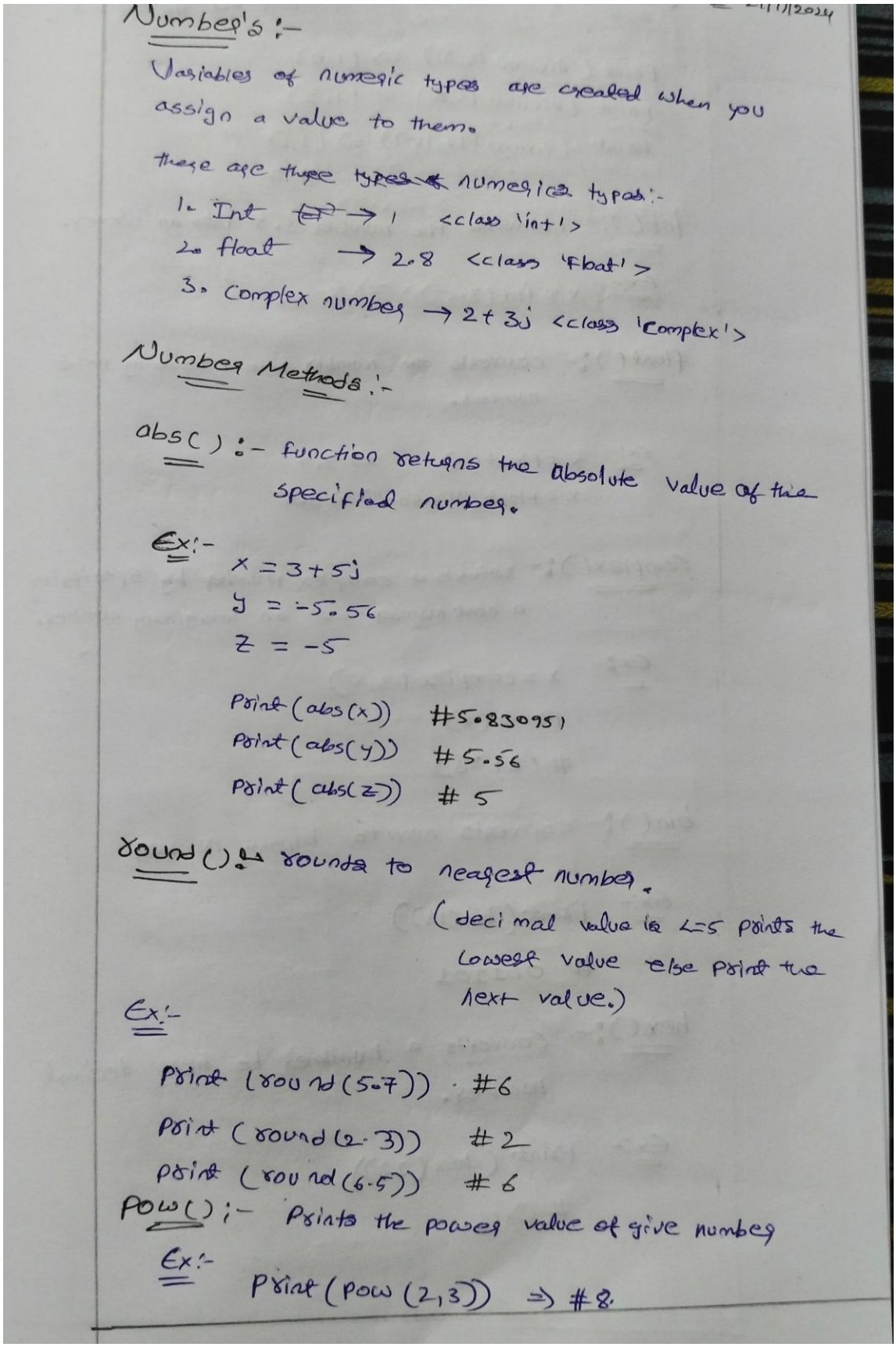
```
txt = "banana"
x = txt.center(10)
print(x)
```

O/P:- banana.

Example:-

```
String = "Python"
res = ""
for i in String:
    if i == i.upper():
        res = res + i.lower()
    else:
        res += i.upper()
Print(res).
```

Number and Number methods:



Numbers :-

Variables of numeric types are created when you assign a value to them.

these are three types of numerical types :-

1. Int → 1 <class 'int'>
2. float → 2.8 <class 'float'>
3. Complex number → 2+3j <class 'complex'>

Number Methods :-

abs() :- function returns the absolute value of the specified number.

Ex :-

```
x = 3+5j
y = -5.56
z = -5

Print(abs(x))   #5.830951
Print(abs(y))   #5.56
Print(abs(z))   #5
```

round() :- rounds to neagest number.
(decimal value is <=5 prints the lowest value else print the next value.)

Ex :-

```
Print(round(5.7))   #6
Print(round(2.3))   #2
print(round(6.5))   #6
```

pow() :- Prints the powers value of give number

Ex :-

```
Print(pow(2,3))  => #8.
```

divmod():- it returns the quotient and the remainder.

print (divmod (3,3)) => (1,0)
print (divmod (5,3)) => (1, 2)
print (divmod (17,15)) => (1,2)

int():- converts the number 3.5 into an integer.

Ex:- x = int(3.5) => 3

float():- converts the number into a floating point number.

Ex:- x = float(3) => 3.0
x = float("3.500") => 3.5

Complex():- returns a complex number by specifying a real number and an imaginary number.

Ex:-
x = complex(3,5)
print(x)
#(3+5j)

bin():- converts num to binary number.

Ex:- Print (bin(13))
0b1101

hex():- Converts a number to hexa decimal number.

Ex:- print (hex(22))
#0x16

oct():- converts a number into the oct number

Ex:-

```
print (oct(16))
#0020
```

Aggregate Functions:-

max():- returns the max value of list or tuple

Ex:-

```
a = (2,3,4,5,7,8,9)
print (max(a))
#9
```

Min():- returns the minimum value

```
a = (2,3,4,5,6,7,8)
print (min(a))
#2
```

Sum():- returns the sum value.

```
print (sum(2,10,8,3,7))
#30
```

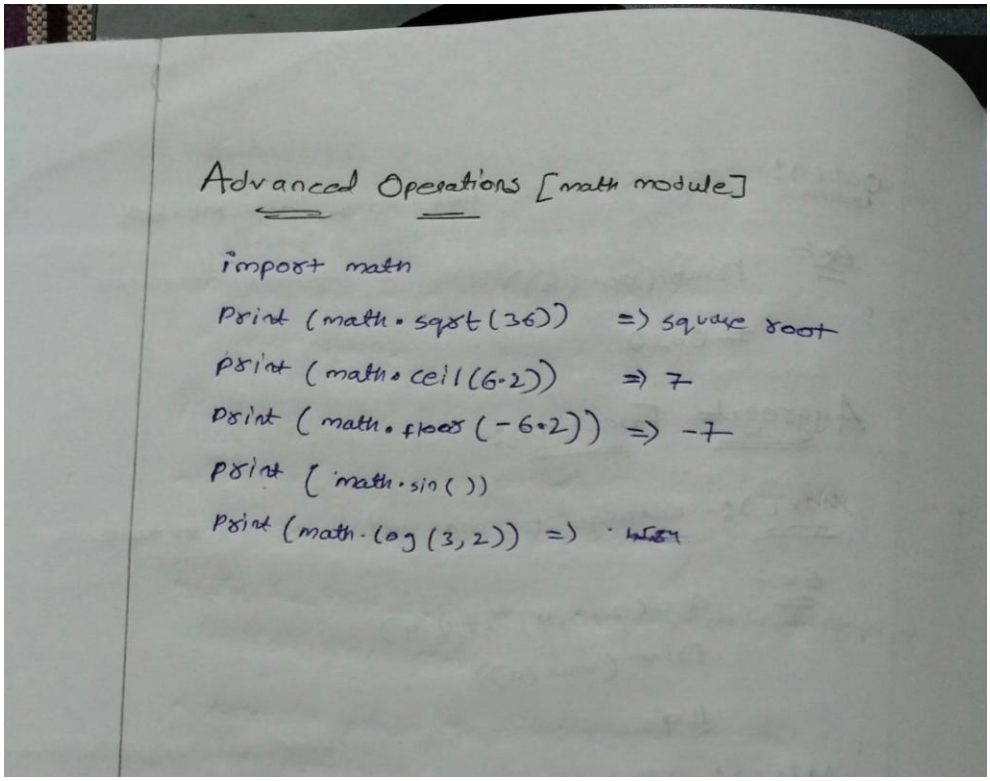
Avg():-

Example:-(without methods)

```
tup = (1,2,3,4)
max = 0

for i in tup:
    if i > max:
        max = i
print (max)
```

(or)

```
max = tup[0]
for i in tup:
    if i > max:
        max = i
print (max)
```



Advanced Operations [math module]

```
import math
Print (math.sqrt(36))      => square root
Print (math.ceil(6.2))     => 7
Print (math.floor(-6.2))   => -7
Print (math.sin())
Print (math.log(3,2))      =>
```

Modules:

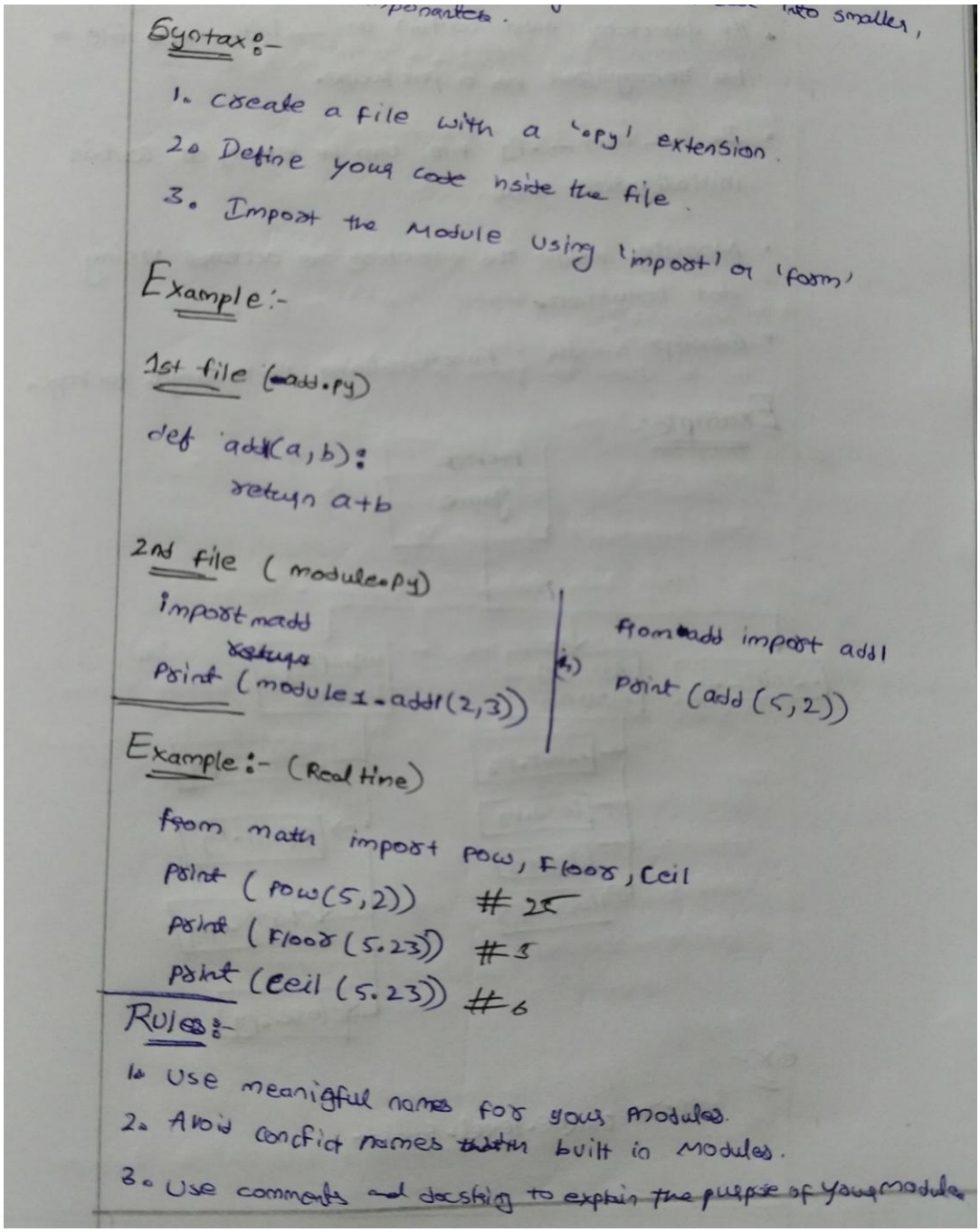
Python Module is a file that contains built-in functions, classes,**its** and variables. There are many **Python modules**, each with its specific work.

In this article, we will cover all about Python modules, such as How to create our own simple module, Import Python modules, From statements in Python, we can use the alias to rename the module, etc.

What is Python Module

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code.

Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.



Packages:

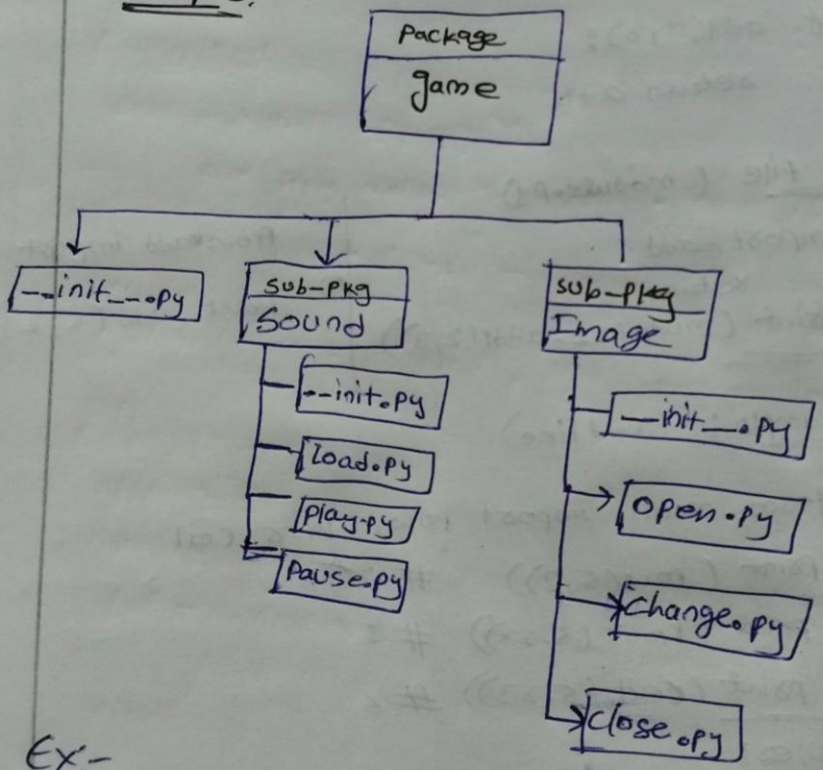
Package:-

A Package is a collection of modules organised in a dictionary.

Rules:-

- The directory must contain an __init__.py file to be recognized as a package.
- The __init__.py file can be empty or contain initialization code.
- Modules inside the directory are accessed using dot notation.
- Group similar functionalities in a single package.

Example:-



Ex:-

```
import Game.level.start
```

Example:- Startwith() method functionality

```
S = "Python is COOL"
chas = "py"
def start_with(S, chas):
    temp = ""
    for i in range(len(chas)):
        temp += S[i]

    if temp == chas:
        return True
    else:
        return False

Print(Start_with(S, chas))
```

Special Symbols used for Passing arguments:-

- *args (Non-keyword Argument)
- **kwargs (keyword Arguments)

*args:- it is used to pass a non-keyworded, variable-length argument list. (to access multiple values)

Example:1:-

```
def myFun(*argv):
    for arg in argv:
        Print(arg)

myFun('Hello', 'welcome', 'to', 'GFG').
```

O/P:-

```
Hello
welcome
to
GFG
```

Example-2:-

```
def myFun(arg1, *argv):
    print("1st Arg:", arg1)
    for arg in argv:
        print(arg)

myFun('Hello', 'welcome', 'to', 'GFG')
```

O/P:-

1st Arg: Hello
welcome
to
GFG

** kwargs:-

It is used to pass a keyworded, variable-length argument list.

Example-1:-

```
def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" %(key, value))

myFun(first='Geeks', mid='for', last='Geeks')
```

O/P:-

first == Geeks
mid == for
last == Geeks.

File Handling:

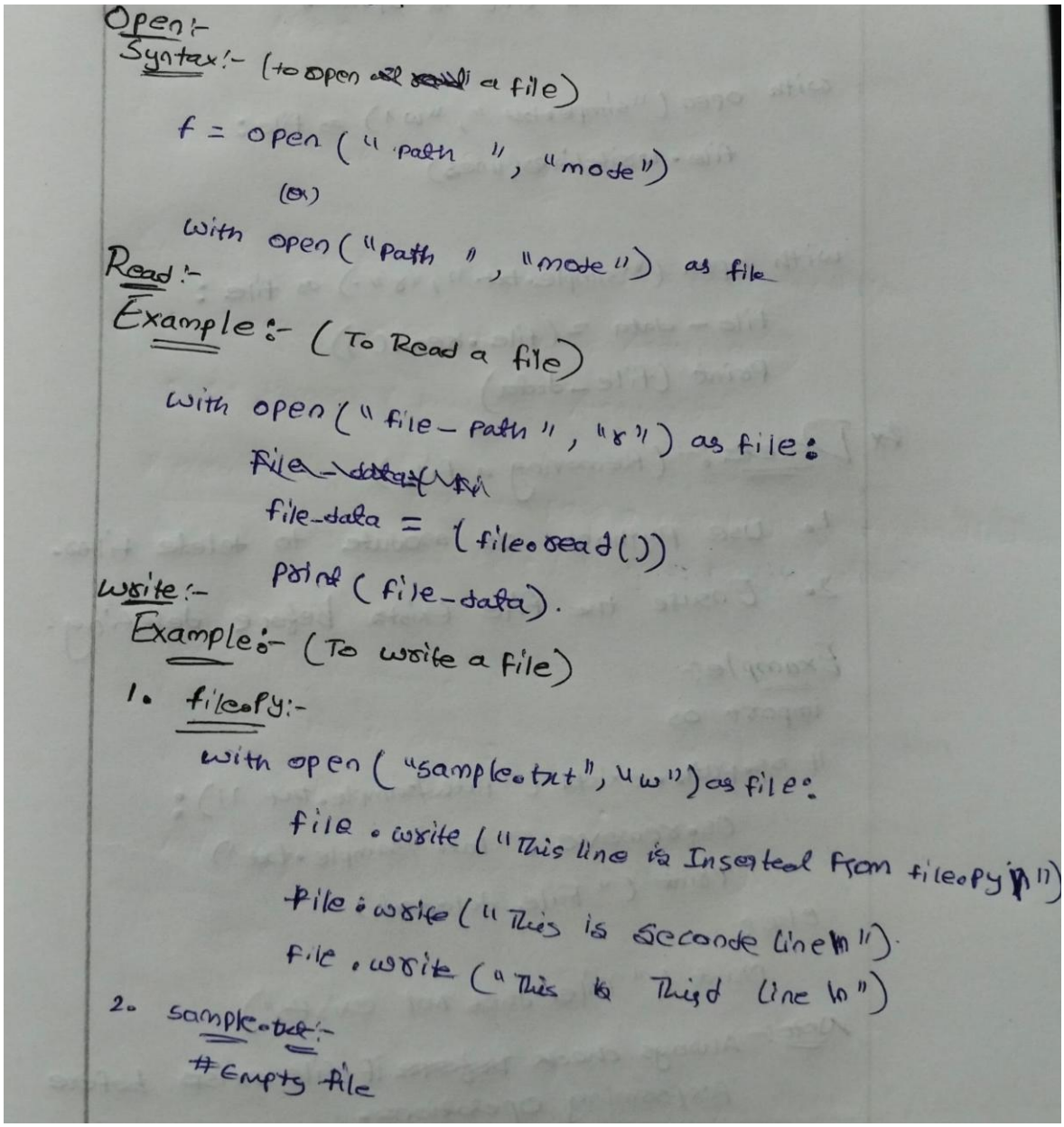
file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files.

Advantages of File Handling in Python: Versatility **,** Flexibility**,** User – friendly and Cross-platform

File Handling

The key function for working with files in Python is the open() function.
The open() function takes two parameters; *filename*, and *mode*.
There are four different methods (modes) for opening a file:

- "r" - Read - Default value. Opens a file for reading, error if the file does not exist
- "a" - Append - Opens a file for appending, creates the file if it does not exist
- "w" - Write - Opens a file for writing, creates the file if it does not exist
- "x" - Create - Creates the specified file, returns an error if the file exists



Read Lines :-

- readline() :- reads only one line
- readlines() :- reads all lines as a page

Update :-

Example :-

```
with open ("simple.txt", "r") as file:
    lines = file.readlines()
lines[1] = "This line is updated"

with open ("simple.txt", "w") as file:
    file.writelines(lines)

with open ("simple.txt", "r") as file:
    file_data = (file.read())
    Print (file_data)
```

Delete :- (Removing a file)

1. Use Python's os module to delete files.
2. Ensure the file exists before deleting.

Example :-

```
import os
if os.path.exists ("Path/sample.txt"):
    os.remove ("Path:/sample.txt")
    print ("File deleted")
else:
    print ("File does not exists")
```

Note :- Always check ~~before~~ if file exists before performing operations.

⇒ If we want to remove a particular line in a file just update the written line with empty string.

Example :-

```
with open ("path/simple.txt", "r") as file:
    lines = file.readlines()
Lines[i] = ""

with open ("path/simple.txt", "w") as file:
    file.writelines (lines)

with open ("path/simple.txt", "r") as file:
    file_data = (file.read())
    print (file_data)
```

File Paths :-

Relative path :- Refers to the file location relative to the script.

Ex :- 'subfolder/file.txt'

Absolute path :- Refers to the complete file location

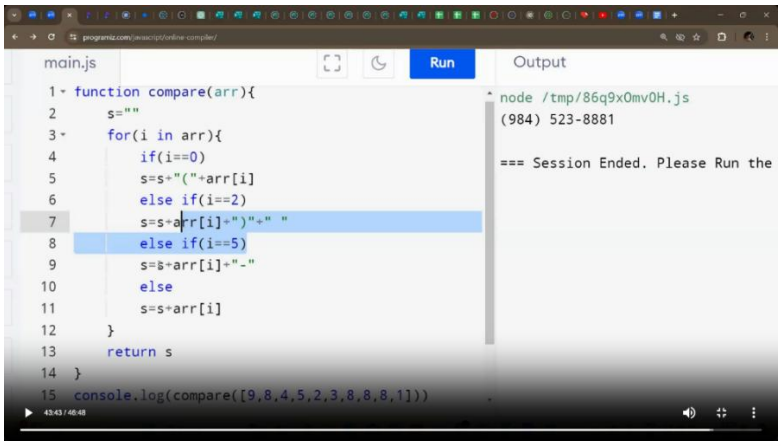
Ex :- 'C:/Users/Name/Documents/file.txt'.

Example :- (Read function)

```
def read (f_name, m):
    with open (f_name, m) as file:
        temp = file.read()
        return temp.

print (read ("sample.text", "r"))
```

TASK :- Create a file perform CURD operations.

Regular Expressions



Introduction

- What is the `re` Module?
- - Provides support for regular expressions in Python.
- - Enables pattern matching, text search, and text manipulation.

- Why Use Regular Expressions?
- - Efficient string processing.
- - Solves complex search and replace problems.

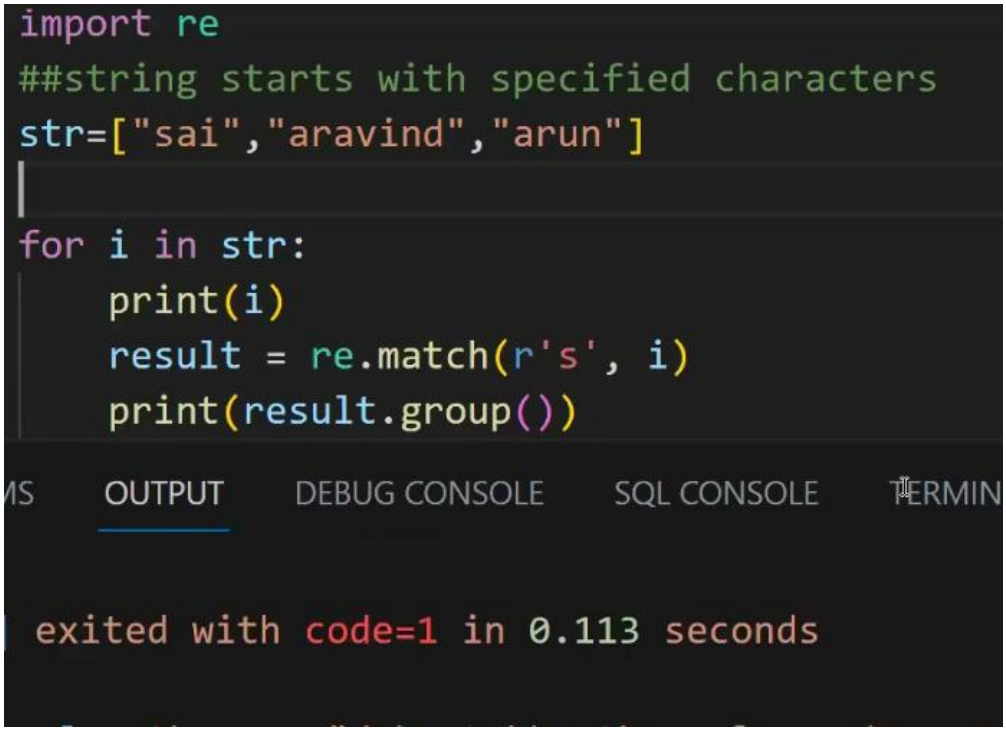
Key Functions

- - `re.match()`: Matches at the beginning of a string.
- - `re.search()`: Searches for a match anywhere in the string.
- - `re.findall()`: Returns all matches as a list.
- - `re.finditer()`: Returns an iterator of match objects.
- - `re.sub()`: Replaces matches with a specified string.
- - `re.split()`: Splits a string by the pattern.

`re.match()`

- Definition: Matches a pattern at the start of the string.

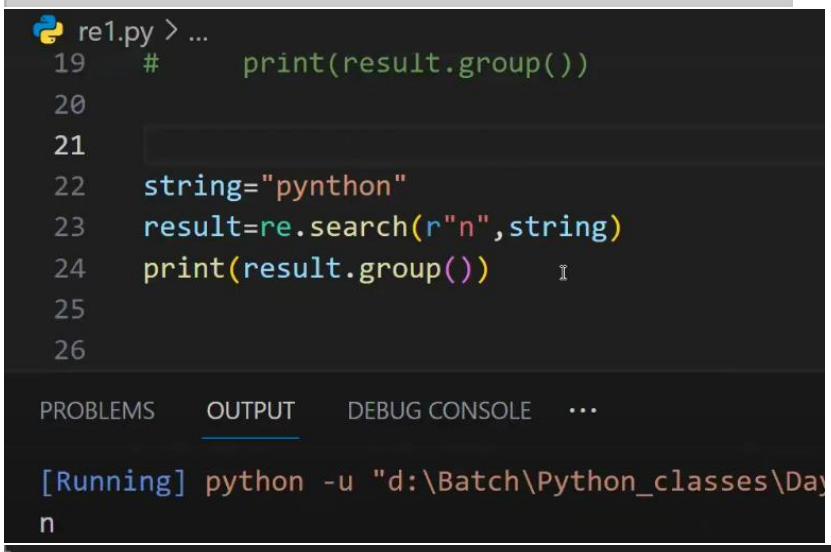
- Example:
- import re
- result = re.match(r'hello', 'hello world')
- print(result.group()) # Output: hello

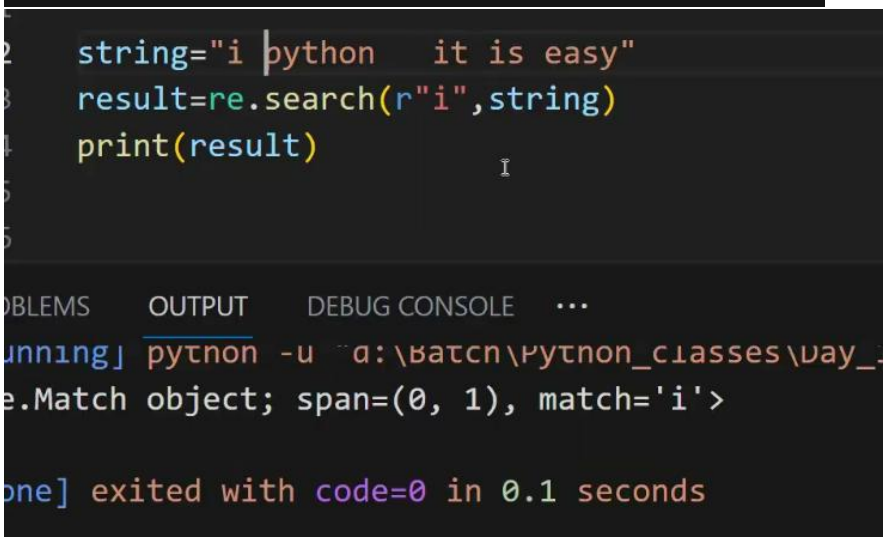


`re.search()`

- Definition: Searches the string for the first match.

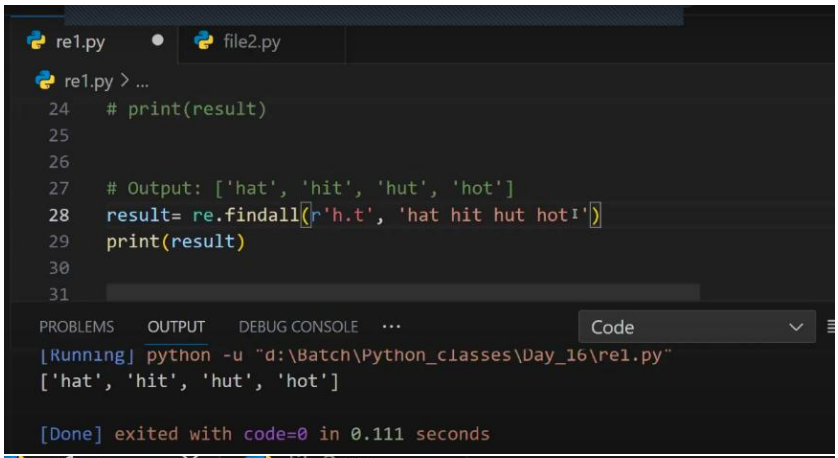
- Example:
- import re
- result = re.search(r'world', 'hello world')
- print(result.group()) # Output: world

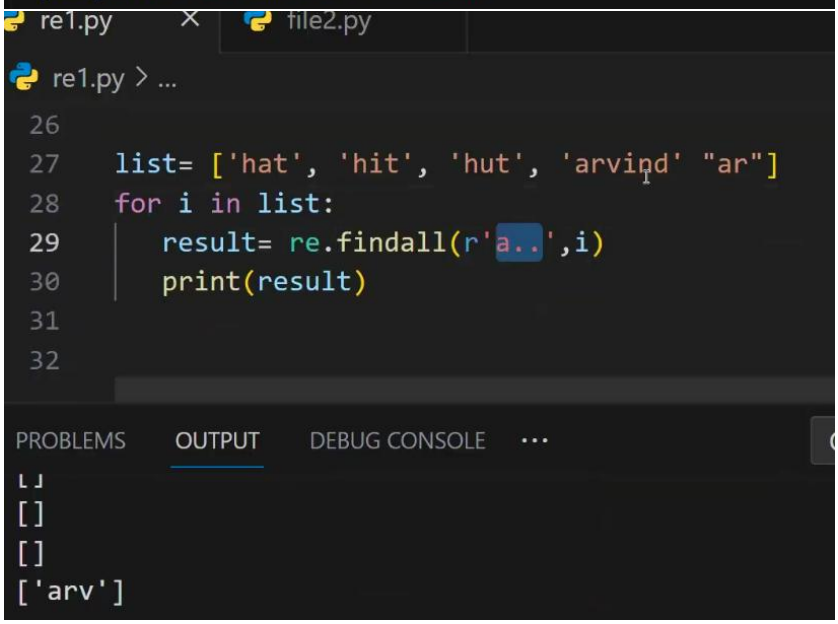




Special Characters

- - `.`: Matches any character except newline.
- - `^`: Matches the start of the string.
- - `$`: Matches the end of the string.
- - `*`: Matches 0 or more repetitions.
- - `+`: Matches 1 or more repetitions.
- - `?`: Matches 0 or 1 repetition.
- - `{m,n}`: Matches between m and n repetitions.





Special Characters Examples

- - `.`: Matches any character except newline.
- Example: re.findall(r'h.t', 'hat hit hut hot') # Output: ['hat', 'hit', 'hut', 'hot']

- - `^`: Matches the start of the string.
- Example: re.findall(r'^Hello', 'Hello world! Hello again!') # Output: ['Hello']

- - `$`: Matches the end of the string.
- Example: re.findall(r'world!$', 'Hello world!')

```
result = re.findall(r'd$', "anand")  # Output: ['world!']
print(result)
```

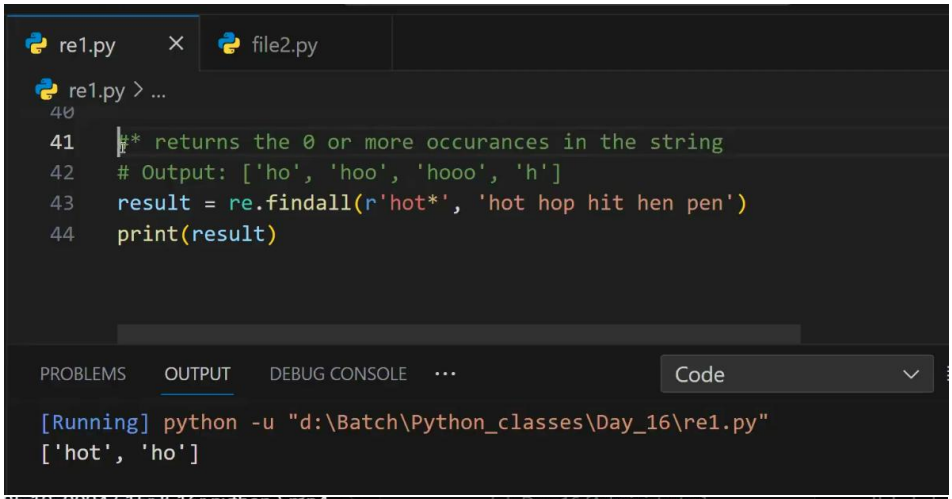
```
OBLEMS  OUTPUT  DEBUG CONSOLE  ...  Code
unning] python -u "d:\Batch\Python_classes\Day_16\re1.py"
d']
```

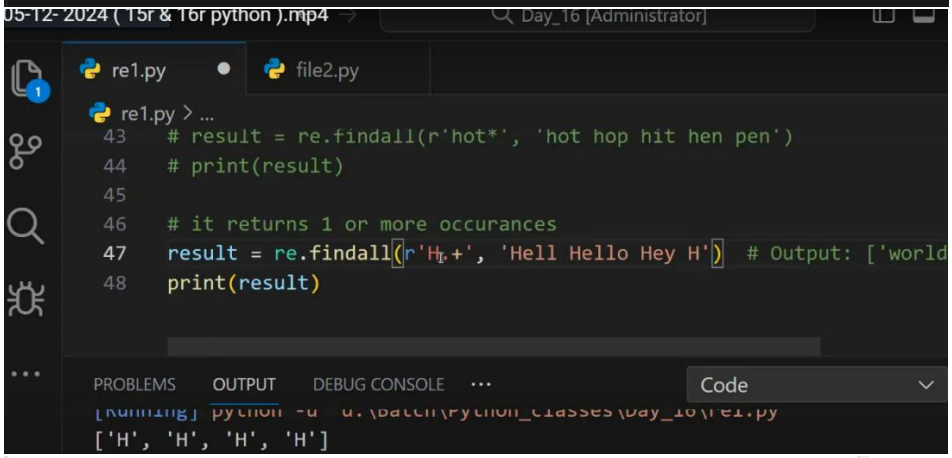
```
37
38  result = re.findall(r'^a...d$', "anand")  # Output: ['world!']
39  print(result)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  ...  Code
['anand']
```

- - `*`: Matches 0 or more repetitions.
- Example: re.findall(r'ho*', 'ho hoo hooo h') # Output: ['ho', 'hoo', 'hooo', 'h']

- - `+`: Matches 1 or more repetitions.





Special Sequences Examples

- - `\d`: Matches any digit (0–9).
- Example: re.findall(r'\d+', 'Phone: 123-456-7890') # Output: ['123', '456', '7890']

- - `\w`: Matches any word character (alphanumeric + `_`).
- Example: re.findall(r'\w+', 'Python_3.9') # Output: ['Python_3', '9']

re1.py file2.py

re1.py > ...

```
51  # \d
52  import re
53
54  # Output: ['123', '456', '7890']
55  result = re.findall(r'\d+', 'Phone: 123-456-7890')
56  print(result)
```

PROBLEMS OUTPUT DEBUG CONSOLE ... Code

```
[Running] python -u "d:\Batch\Python_classes\Day_16\re1.py"
['123', '456', '7890']
```

re1.py file2.py

re1.py > ...

```
53
54  # Output: ['123', '456', '7890']  (0-9)
55  result = re.findall(r'\d+',"secret@1")
56  if(len(result)>0):
57      print("valid")
58  else:
59      print("invalid")
```

PROBLEMS OUTPUT DEBUG CONSOLE ... Code

```
[Running] python -u "d:\Batch\Python_classes\Day_16\re1.py
valid
```

re1.py file2.py

re1.py > ...

```
62
63  #\w   -  word character
64  result=re.findall(r'\w+', ' -# Python_3.13 @latest  ')
65  print(result)
```

PROBLEMS OUTPUT DEBUG CONSOLE ... Code

```
[Running] python -u "d:\Batch\Python_classes\Day_16\re1.py"
['Python_3', '13', 'latest']
```

- - `\s`: Matches any whitespace character.
- Example: re.findall(r'\s+', 'Hello World') # Output: [' ']

```
67
68    result=re.findall(r'\s+', '       ')
69    print(result)
```

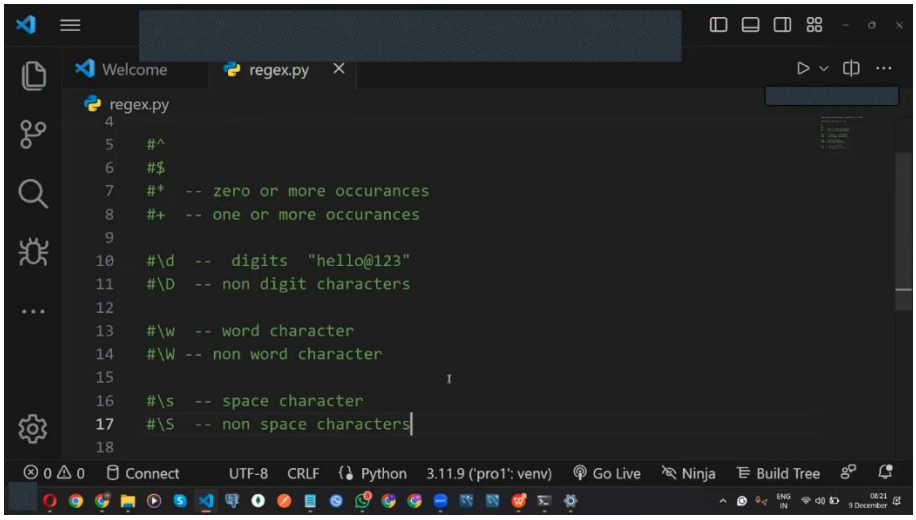
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  SQL CONSOLE  ...  Code
[Running] python -u "d:\Batch\Python_classes\Day_16\re1.py"
['       ']
```

- - `\b`: Matches word boundaries.
- Example: re.findall(r'\bword\b', 'A word in words.') # Output: ['word']

- - `\B`: Matches non-boundaries.
- Example: re.findall(r'\Bword', 'inword') # Output: ['word']

```
73    result=re.findall(r'\D+', ' -# Python_3.13 @latest  ')
74    print(result)
75    result=re.findall(r'\W', ' -# Python_3.13 @latest  ')
76    print(result)
77    result=re.findall(r'\S+', ' -# Python_3.13 @latest  ')
78    print(result)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  SQL CONSOLE  ...  Code
[Running] python -u "d:\Batch\Python_classes\Day_16\re1.py"
[' -# Python_', '.', ' @latest  ']
[' ', '-', '#', ' ', '.', ' ', '@', ' ', ' ']
['-#', 'Python_3.13', '@latest']
```



Combining Character Classes

- Combine different character classes in a single pattern.

- Example:
- re.findall(r'\d\w\s', '3a ')
- Output: ['3a ']

Grouping and Repetition

- Use parentheses to group parts of a pattern and apply repetition.

- Example:
- re.findall(r'(abc){2,3}', 'abcabcabc')
- Output: ['abcabcabc']

```
20  string="hello world#123 hello everyone"
21
22  # res=re.findall(r'\S', string)
23  # print(res)
24
25  print(re.findall(r'(hello)', string))
26
27
```

PROBLEMS OUTPUT DEBUG CONSOLE SQL CONSOLE TERMINA

```
[Running] python -u "d:\Batch\Python_classes\Day_17
['hello', 'hello']
```

```
24  #() is used to group the characters
```

Nested Patterns with Anchors

- Combine anchors and quantifiers for complex matching.

- Example:
- re.findall(r'^A\d+Z$', 'A123Z')
- Output: ['A123Z']

Lookahead and Lookbehind

- - Positive Lookahead: Ensures a pattern is followed by another.
- Example:
- re.findall(r'foo(?=bar)', 'foobar') # ['foo']
- - Positive Lookbehind: Ensures a pattern is preceded by another.
- Example:
- re.findall(r'(?<=foo)bar', 'foobar') # ['bar']

```
31
32  print(re.findall(r'foo(?=bar)', 'foobar football footpath'))
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  SQL CONSOLE  TERMINAL  Code
[Running] python -u "d:\Batch\Python_classes\Day_17\regex.py"
['foo']
```

```
33
34  print(re.findall(r'(?<=foot)ball', 'football baseball vollyball'))
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  SQL CONSOLE  TERMINAL  Code
[Running] python -u "d:\Batch\Python_classes\Day_17\regex.py"
['ball']
```

- email_regex = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

- password_regex = r'^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$'

```
44
45  string=" uhello everyone"
46  #grouping of characters [aeiou]
47  print(re.findall(r'^[A-z,0-9]',string))
48
```

PROBLEMS OUTPUT DEBUG CONSOLE SQL CONSOLE TERMINAL Code

```
[Running] python -u "d:\Batch\Python_classes\Day_1\regex.py"
[]
```

```
1   #regular expressions
2   #pattern mathcing
3
4   #\d \w \s  \D \W \S
5   #() -- pattern to be checked  is given in this braces
6   #[] -- [abc]  options to be grouped
7   #{} -- repetetions are given in this braces {2,5}
8   #lookup foot(?=ball)
9   #look behind   (?<=foot)ball
10
11
```

Errors and Exceptions

Errors and Their Types in Python

Errors in Python occur when the interpreter encounters something it cannot execute. Errors can be broadly classified into **syntax errors** and **exceptions**.

---

1. Syntax Errors (Compile-Time Errors)

A **SyntaxError** occurs when the Python interpreter finds an incorrect syntax (wrong grammar of Python).

Example:

```
if True
   print("Hello")
```

Error Output:

```
SyntaxError: expected ':'
```

✓ **Fix:** Add a colon : after if True.

---

2. Exceptions (Runtime Errors)

Exceptions occur **at runtime** when a valid syntax is executed but results in an error.

Types of Exceptions in Python

Here are some common types of exceptions:

(i) NameError

Occurs when trying to use a variable that is not defined.

```
print(x)  # x is not defined
```

Error Output:

```
NameError: name 'x' is not defined
```

✓ **Fix:** Define x before using it.

(ii) TypeError

Occurs when an operation is performed on an incompatible type.

```
print(5 + "hello")  # Integer + String
```

Error Output:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

✓ **Fix:** Convert data types before operations:

```
print(str(5) + "hello")  # Output: 5hello
```

(iii) ValueError

Occurs when a function receives an argument of the correct type but an inappropriate value.

```
num = int("abc")  # Cannot convert "abc" to an integer
```

Error Output:

```
ValueError: invalid literal for int() with base 10: 'abc'
```

✓ **Fix:** Ensure valid input before conversion.

(iv) IndexError

Occurs when trying to access an index that is out of range.

```
lst = [1, 2, 3]
print(lst[5])  # Index out of
range
```

Error Output:

```
IndexError: list index out of range
```

✓ **Fix:** Check list length before accessing indexes.

(v) KeyError

Occurs when trying to access a dictionary key that does not exist.

```
my_dict = {"a": 1, "b": 2}
print(my_dict["c"])  # Key 'c' does not exist
```

Error Output:

```
KeyError: 'c'
```

✓ **Fix:** Use .get() to handle missing keys:

```
print(my_dict.get("c", "Key not found"))  # Output: Key not found
```

(vi) AttributeError

Occurs when trying to access an attribute that does not exist.

```
x = 10
x.append(5)  # Integers do not have an append() method
```

Error Output:

```
AttributeError: 'int' object has no attribute 'append'
```

✓ **Fix:** Use correct data types.

(vii) ZeroDivisionError

Occurs when trying to divide a number by zero.

```
print(10 / 0)
```

Error Output:

```
ZeroDivisionError: division by zero
```

✓ **Fix:** Ensure the denominator is not zero before dividing.

(viii) FileNotFoundError

Occurs when trying to open a file that does not exist.

```
with open("nonexistent_file.txt", "r") as f:
    content = f.read()
```

Error Output:

```
FileNotFoundError: [Errno 2] No such file or directory: 'nonexistent_file.txt'
```

✓ **Fix:** Check if the file exists before opening.

(ix) ImportError / ModuleNotFoundError

Occurs when trying to import a module that does not exist.

```
import non_existent_module
```

Error Output:

```
ModuleNotFoundError: No module named 'non_existent_module'
```

✓ **Fix:** Install or check the module name.

Exception Handling in Python

Exception handling in Python allows you to gracefully handle runtime errors, preventing program crashes. This is done using try, except, else, and finally blocks.

---

1. Basic Exception Handling using try-except

A try block is used to enclose code that may raise an exception. If an error occurs, the except block handles it.

Example: Handling Division by Zero

```
try:
    result = 10 / 0  # Raises ZeroDivisionError
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
```

✓ **Output:**

```
Error: Cannot divide by zero!
```

---

2. Handling Multiple Exceptions

You can handle different types of exceptions separately.

Example: Handling ZeroDivisionError and ValueError

```
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 number.")
```

✓ **Input:**

```
Enter a number: abc
```

✓ **Output:**

```
Error: Invalid input! Please enter a number.
```

---

3. Catching Multiple Exceptions in One except Block

Instead of writing multiple except blocks, you can use a tuple to catch multiple exceptions in a single block.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ZeroDivisionError, ValueError) as e:
    print(f"Error: {e}")
```

---

4. Using else with try-except

The else block runs **only if no exceptions occur**.

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 number.")
```

```
else:
    print("Success! The result is:", result)
```

✓ **Input:** 5

✓ **Output:**

```
Success! The result is: 2.0
```

---

5. Using finally Block

The finally block **always executes**, whether an exception occurs or not.

Example:

```
try:
    f = open("file.txt", "r")
    content = f.read()
except FileNotFoundError:
    print("Error: File not found!")
finally:
    print("Execution completed.")
```

✓ **Output:**

```
Error: File not found!
Execution completed.
```

---

6. Raising Custom Exceptions Using raise

You can manually raise exceptions using raise.

Example: Raising ValueError

```
age = int(input("Enter your age: "))
if age < 0:
    raise ValueError("Age cannot be negative!")
```

✓ **Input:** -5

✓ **Output:**

```
ValueError: Age cannot be negative!
```

---

7. Creating Custom Exceptions

You can define your own exception classes by inheriting from Exception.

Example: Custom Exception for Negative Numbers

```
class NegativeNumberError(Exception):
    pass

num = int(input("Enter a positive number: "))
if num < 0:
    raise NegativeNumberError("Negative numbers are not allowed!")
```

✓ **Input:** -3

✓ **Output:**

```
NegativeNumberError: Negative numbers are not allowed!
```

---

8. Handling All Exceptions (Exception)

Using Exception in except can catch **all** types of errors.

Example:

```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
except Exception as e:
    print(f"An error occurred: {e}")
```

✓ **Input:** "abc"

✓ **Output:**

An error occurred: invalid literal for int() with base 10: 'abc'

Conclusion

- **try-except**: Handles exceptions.
- **else**: Runs when no exceptions occur.
- **finally**: Runs **always**, even if an exception occurs.
- **raise**: Manually raises an exception.
- **Custom Exceptions**: Create user-defined exception classes.

Using exception handling properly ensures your program is **robust, user-friendly, and error-free!**

finally Block in Python

The finally block is used in Python to execute code **regardless of whether an exception occurs or not**. It is commonly used for **cleanup operations** like closing files, releasing resources, or disconnecting from databases.

---

1. Syntax of finally Block

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Handling exception
finally:
    # Code that always executes
```

---

2. Example: finally Executes Always

```
try:
    print("Try block executing...")
    result = 10 / 2  # No exception
except ZeroDivisionError:
    print("Cannot divide by zero!")
finally:
    print("Finally block always executes!")
```

✓ **Output:**

```
Try block executing...
Finally block always executes!
```

---

3. finally Block When an Exception Occurs

Even if an exception occurs, the finally block still executes.

```
try:
    print("Trying to divide by zero...")
    result = 10 / 0  # Causes ZeroDivisionError
except ZeroDivisionError:
    print("Caught ZeroDivisionError!")
finally:
    print("Finally block executed!")
```

✓ **Output:**

```
Trying to divide by zero...
Caught ZeroDivisionError!
Finally block executed!
```

---

4. Using finally for Resource Cleanup

A common use case of finally is ensuring that a file or database connection is properly closed.

Example: Closing a File

```
try:
    f = open("example.txt", "r")
    content = f.read()
except FileNotFoundError:
    print("File not found!")
finally:
    print("Closing the file...")
    f.close()  # This ensures the file is always closed
```

✓ **Output (if file is missing):**

```
File not found!
Closing the file...
```

Even though an exception occurs (FileNotFoundError), the finally block executes, ensuring the file is closed.

---

5. finally with return Statement

Even if a function has a return statement inside try or except, the finally block **still executes before returning**.

```
def test_finally():
    try:
        return "Try block executed"
    finally:
        print("Finally block executed!")

print(test_finally())
```

✓ **Output:**

```
Finally block executed!
Try block executed
```

Even though return is inside try, finally executes **before returning**.

---

6. finally with raise

If an exception is raised inside try and not caught in except, the finally block **still runs before the program crashes**.

```
try:
    print("Before exception")
    raise ValueError("Something went wrong!")  # Raising an exception
finally:
    print("Finally executed before crashing!")
```

✓ **Output:**

```
Before exception
Finally executed before crashing!
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: Something went wrong!
```

The finally block **executes before the exception terminates the program**.

---

7. finally in Nested Try Blocks

A finally block inside a nested try-except also executes.

```
try:
    try:
```

```
        print("Inner try block")
        raise ZeroDivisionError
    finally:
        print("Inner finally block")
except ZeroDivisionError:
    print("Exception handled in outer block")
finally:
    print("Outer finally block")
```

✓ **Output:**

```
Inner try block
Inner finally block
Exception handled in outer block
Outer finally block
```

Both finally blocks **execute regardless of the exception**.

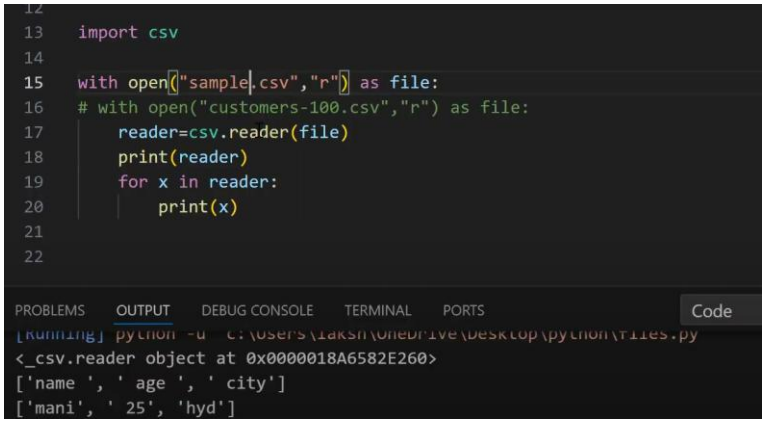
Conclusion

- finally **always executes**, even if an exception occurs.
- Used for **cleanup tasks** like closing files, releasing memory, or disconnecting databases.
- Executes **before returning** if return is present.
- **Executes even if an exception is raised** and not caught.

The finally block ensures **reliable cleanup** and helps prevent resource leaks in programs! 🚀

CSV Files

- A simple way to store big data sets is to use CSV files (comma separated files).
- CSV files contains plain text and is a well know format that can be read by everyone including Pandas.
- In Python, We can work with CSV (Comma Separated Values) files using the built-in csv module or the popular Pandas library. The csv module provides tools for reading and writing CSV files, while Pandas offers a more user-friendly approach with its read_csv() and to_csv() functions.



1. Importing the csv Module

Before working with CSV files, you need to import Python's built-in csv module.

import csv

2. Reading a CSV File

You can read a CSV file using the csv.reader() method.

Example: Reading a CSV File

Assume **data.csv** contains:

Name,Age,City
Alice,25,New York
Bob,30,London
Charlie,22,Sydney

Code:

```
import csv
with open("data.csv", "r") as file:
   reader = csv.reader(file)  # Read file
   for row in reader:
      print(row)  # Each row is a list
```

✓ Output:

['Name', 'Age', 'City']
['Alice', '25', 'New York']
['Bob', '30', 'London']
['Charlie', '22', 'Sydney']

Skipping Headers while Reading

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

✓ Output (Without Headers):

['Alice', '25', 'New York']

['Bob', '30', 'London']
['Charlie', '22', 'Sydney']

---

3. Writing to a CSV File

You can write data to a CSV file using csv.writer().

Example: Writing Data to a CSV File

```
import csv
data = [
   ["Name", "Age", "City"],
   ["David", 28, "Berlin"],
   ["Emma", 24, "Paris"]
]

with open("output.csv", "w", newline="") as file:
   writer = csv.writer(file)  # Create writer object
   writer.writerows(data)  # Write all rows at once
```

✓ Creates output.csv with:

```
Name,Age,City
David,28,Berlin
Emma,24,Paris
```

Writing One Row at a Time

```
with open("output.csv", "w", newline="") as file:
   writer = csv.writer(file)
   writer.writerow(["Name", "Age", "City"])  # Write header
   writer.writerow(["John", 29, "New York"])  # Write single row
```

---

4. Appending to a CSV File

Use "a" mode to append data **without overwriting**.

Example: Appending Data

```
with open("output.csv", "a", newline="") as file:
   writer = csv.writer(file)
   writer.writerow(["Sophia", 27, "Toronto"])
```

✓ Adds new row to output.csv:

```
Name,Age,City
John,35,Chicago
Anna,28,Los Angeles
Sophia,27,Toronto
```

---

5. Reading CSV as Dictionary

Instead of lists, you can read CSV files as **dictionaries** using csv.DictReader().

Example: Using DictReader

```
import csv

with open("data.csv", "r") as file:
   reader = csv.DictReader(file)
   for row in reader:
      print(row)  # Each row is an OrderedDict
```

✓ Output:

```
{'Name': 'Alice', 'Age': '25', 'City': 'New York'}
{'Name': 'Bob', 'Age': '30', 'City': 'London'}
{'Name': 'Charlie', 'Age': '22', 'City': 'Sydney'}
```

---

6. Writing CSV as Dictionary

You can write a dictionary into a CSV file using csv.DictWriter().

Example: Using DictWriter

```
import csv

data = [
   {"Name": "John", "Age": 35, "City": "Chicago"},
   {"Name": "Anna", "Age": 28, "City": "Los Angeles"}
]

with open("output.csv", "w", newline="") as file:
   fieldnames = ["Name", "Age", "City"]
   writer = csv.DictWriter(file, fieldnames=fieldnames)

   writer.writeheader()  # Write header
   writer.writerows(data)  # Write multiple rows
```

✓ Creates output.csv with:

```
Name,Age,City
John,35,Chicago
Anna,28,Los Angeles
```

---

7. Handling Different Delimiters (e.g., ;, |)

By default, CSV files use a **comma (,)** as a separator. You can change this using the delimiter argument.

Example: Using ; as a Separator

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

✓ For a file containing:

```
Name;Age;City
Alice;25;New York
```

✓ Output:

```
['Name', 'Age', 'City']
['Alice', '25', 'New York']
```

Example: writing a new delimeter

```
import csv
data = [
   ["Name", "Age", "City"],
   ["Sophia", 27, "Toronto"],
   ["John", 30, "New York"]
]
with open("data2.csv", "w", newline="") as file:
   writer = csv.writer(file, delimiter="|")  # Use '|' as delimiter
   writer.writerows(data)
```

---

8. Handling Quotes in CSV Files

Sometimes, CSV fields contain **commas or quotes**. You can handle them using the quotechar argument.

Example: Handling Quotes

```
with open("data.csv", "r") as file:
```

```
    reader = csv.reader(file, quotechar=' " ')
    for row in reader:
        print(row)
```

---

9. Checking if a CSV File Exists Before Writing

```
import os
import csv

filename = "output.csv"

if not os.path.exists(filename):
    with open(filename, "w", newline="") as file:
        writer = csv.writer(file)
        writer.writerow(["Name", "Age", "City"])  # Write header
```

This **prevents overwriting existing files**.

---

10. Using pandas for CSV Files (Alternative)

The pandas library provides a simpler way to work with CSV files.

Reading a CSV File

```
import pandas as pd

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

Writing a CSV File

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

✓ **This automatically handles headers, delimiters, and formatting!** 🚀

Conclusion

- ✅ csv.reader() → Reads CSV file as a **list of lists**
- ✅ csv.writer() → Writes data to a CSV file
- ✅ csv.DictReader() → Reads CSV file as **dictionary**
- ✅ csv.DictWriter() → Writes data to a CSV file as **dictionary**
- ✅ pandas → Easier CSV handling with read_csv() and to_csv()

Python's csv module provides a **powerful yet simple way** to handle structured data!

NUMPY LIBRARIES

NumPy(Numerical Python)

NumPy(Numerical Python) is a fundamental library for Python **numerical computing**. It provides efficient multi-dimensional array objects and various mathematical functions for handling large datasets making it a critical tool for professionals in fields that require heavy computation.

NumPy (Numerical Python) is a **powerful library** for numerical computing in Python. It is mainly used for **working with arrays, matrices, and performing mathematical operations** efficiently.

Installing NumPy in Python

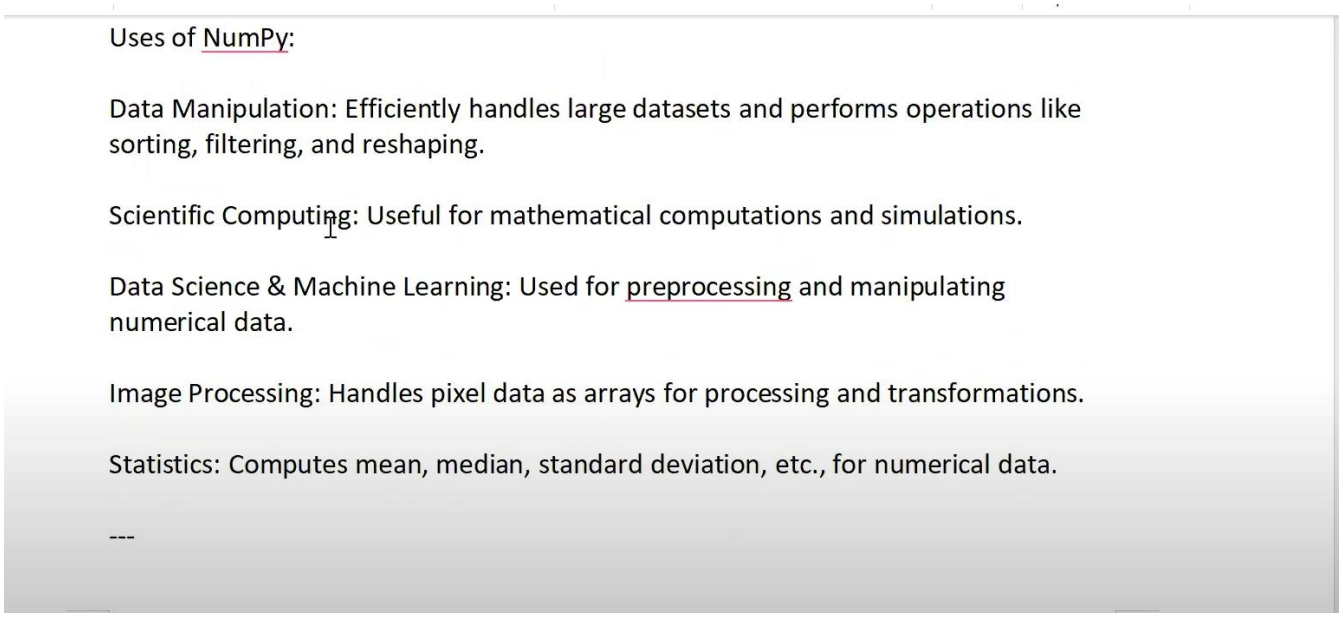
To begin using NumPy, you need to install it first. This can be done through pip command:

pip install numpy

Key Features of NumPy

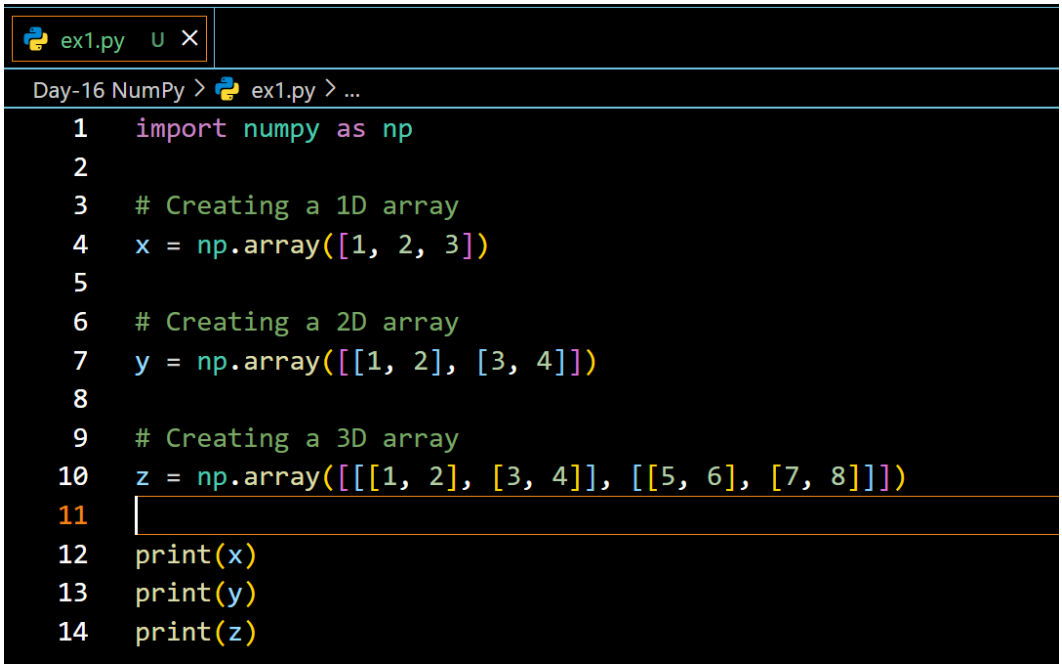
NumPy has various features that make it popular over lists.

- **N-Dimensional Arrays**: NumPy's core feature is ndarray, a powerful N-dimensional array object that supports homogeneous data types.
- **Arrays with High Performance**: Arrays are stored in contiguous memory locations, enabling faster computations than Python lists (Please see Numpy Array vs Python List for details).
- **Broadcasting:** This allows element-wise computations between arrays of different shapes. It simplifies operations on arrays of **various shapes** by automatically aligning their dimensions without creating new data.
- **Vectorization:** Eliminates the need for explicit Python loops by applying operations directly on entire arrays.
- **Linear algebra**: NumPy contains routines for linear algebra operations, such as matrix multiplication, decompositions, and determinants.



Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called ndarray.
We can create a NumPy ndarray object by using the array() function.

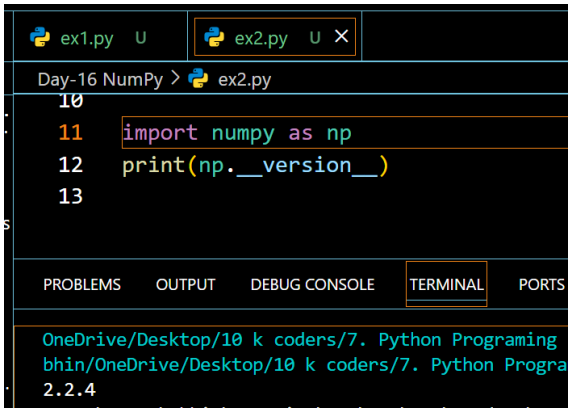


Output

```
[1 2 3]
[[1 2]
 [3 4]]
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```

Checking NumPy Version

The version string is stored under __version__ attribute.



Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])    // 1
```

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

Example

Access the element on the first row, second column:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st row: ', arr[0, 1])
```

NumPy Array Slicing

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [*start*:*end*].
We can also define the step, like this: [*start*:*end*:*step*].

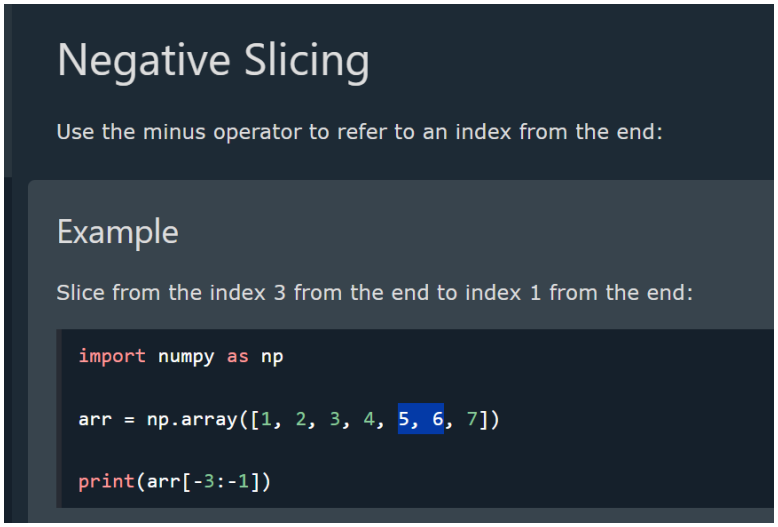
If we don't pass start its considered 0
If we don't pass end its considered length of array in that dimension
If we don't pass step its considered 1

```
import numpy as np

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

print(arr[1:5])
```

```
[2 3 4 5]
```

```
import numpy as np

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

print(arr[-3:-1])
```

```
[5 6]
```




```
import numpy as np

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

print(arr[1:5:2])
```

```
[2 4]
```

```
import numpy as np

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

print(arr[::2])
```

```
[1 3 5 7]
```

```
#Slicing 2-D Arrays
#Example
#From the second element, slice elements from index 1 to index 4 (not included):

import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[1, 1:4])
```

```
[7 8 9]
```

Data Types in NumPy

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

- i - integer
- b - boolean
- u - unsigned integer
- f - float
- c - complex float
- m - timedelta
- M - datetime
- O - object
- S - string
- U - unicode string

- V - fixed chunk of memory for other type (void)

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr.dtype)
```

```
int64
```

Creating Arrays With a Defined Data Type

We use the array() function to create arrays, this function can take an optional argument: dtype that allows us to define the expected data type of the array elements:

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='S')

print(arr)
print(arr.dtype)
```

```
[b'1' b'2' b'3' b'4']
|S1
```

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='i4')

print(arr)
print(arr.dtype)
```

```
[1 2 3 4]
int32
```

Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the astype() method.

The astype() function creates a copy of the array, and allows you to specify the data type as a parameter.

```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')

print(newarr)
print(newarr.dtype)
```

```
[1 2 3]
int32
```

NumPy Array Properties

Day-16 NumPy > array_properties.py > ...

```python
import numpy as np

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

print(arr.shape)  # (2, 3) → 2 rows, 3 columns
print(arr.ndim)   # 2 → Number of dimensions
print(arr.size)   # 6 → Total number of elements
print(arr.dtype)  # int32 → Data type of elements
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
"
(2, 3)
2
6
int64
```

Reshape From 1-D to 2-D

Example

Convert the following 1-D array with 12 elements into a 2-D array.

The outermost dimension will have 4 arrays, each with 3 elements:

```python
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3)

print(newarr)
```

Reshape From 1-D to 3-D

Example

Convert the following 1-D array with 12 elements into a 3-D array.

The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

```python
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(2, 3, 2)

print(newarr)
```

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

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

Generating Special Arrays

Day-16 NumPy > special_arrays.py

```python
import numpy as np

print(np.zeros((2, 3)))      # 2x3 matrix filled with 0s
print(np.ones((3, 3)))       # 3x3 matrix filled with 1s
print(np.eye(4))             # 4x4 identity matrix
print(np.arange(1, 10, 2) )  # [1, 3, 5, 7, 9] (like range())
print(np.linspace(0, 5, 10)) # 10 evenly spaced numbers from 0 to 5
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[[0. 0. 0.]
 [0. 0. 0.]]
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
[1 3 5 7 9]
[0.         0.55555556 1.11111111 1.66666667 2.22222222 2.77777778
 3.33333333 3.88888889 4.44444444 5.        ]
PS C:\Users\abhin\OneDrive\Desktop\10 k coders\7. Python Programing Language>
```

Array Operations using numPy

Day-16 NumPy > operators_np.py > ...

```python
import numpy as np

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

print(a + b)   # [5 7 9]  → Element-wise addition
print(a - b)   # [-3 -3 -3]  → Element-wise subtraction
print(a * b)   # [4 10 18]  → Element-wise multiplication
print(a / b)   # [0.25 0.4 0.5]  → Element-wise division
print(a ** 2)  # [1 4 9]  → Element-wise exponentiation


A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
print(A @ B)      # Matrix multiplication
print(A.T)        # Transpose of A
print(np.linalg.inv(A))  # Inverse of A
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\abhin\OneDrive\Desktop\10 k coders\7. Python Programing Language> & "c:/Users/abhin/OneDrive/Desktop/10 k coders/7. Python Programing Language/.venv/Scripts/python.exe" "c:/Users/abhin/OneDrive/Desktop/10 k coders/7. Python Programing Language/Day-16 NumPy/operators_np.py"
[5 7 9]
[-3 -3 -3]
[ 4 10 18]
[0.25 0.4  0.5 ]
[1 4 9]
[[19 22]
 [43 50]]
[[1 3]
 [2 4]]
[[-2.   1. ]
[0.25 0.4  0.5 ]
[1 4 9]
[[19 22]
 [43 50]]
[[1 3]
 [2 4]]
[[-2.   1. ]
[1 4 9]
[[19 22]
 [43 50]]
[[1 3]
 [2 4]]
[[-2.   1. ]
[[19 22]
 [43 50]]
[[1 3]
 [2 4]]
[[-2.   1. ]
 [43 50]]
[[1 3]
 [2 4]]
[[-2.   1. ]
 [2 4]]
[[-2.   1. ]
[[-2.   1. ]
 [ 1.5 -0.5]]
PS C:\Users\abhin\OneDrive\Desktop\10 k coders\7. Python Programing Language>
```

Logical Operators

NumPy allows **element-wise logical comparisons**.

python Copy Edit

```python
arr = np.array([10, 20, 30, 40, 50])

print(arr > 25)   # [False False  True  True  True]
print(arr[arr > 25])  # [30 40 50] (Elements greater than 25)
```

- `np.dot()`: Performs matrix multiplication.

```
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])
result = np.dot(matrix1, matrix2)
print(result)
# Output: [[19 22]
#          [43 50]]
```

Aggregate Functions

NumPy provides built-in functions for mathematical operations.

python Copy Edit

```python
arr = np.array([1, 2, 3, 4, 5])

print(np.sum(arr))    # 15 → Sum of elements
print(np.mean(arr))   # 3.0 → Mean (average)
print(np.median(arr)) # 3.0 → Median
print(np.std(arr))    # 1.41 → Standard deviation
print(np.min(arr))    # 1 → Minimum value
print(np.max(arr))    # 5 → Maximum value
```

NumPy Array Iterating

Iterating means going through elements one by one.
As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.
If we iterate on a 1-D array it will go through each element one by one.

```python
import numpy as np

arr = np.array([1, 2, 3])
for x in arr:
  print(x)
```

```
1
2
3
```

```python
import numpy as np

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

for x in arr:
  print(x)
```

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

4. Array Manipulation:
   - `np.reshape()`: Changes the shape of an array.

```
reshaped = np.reshape(data, (1, 5))
```

- `np.concatenate()`: Combines multiple arrays.

```
array3 = np.array([7, 8, 9])
combined = np.concatenate((data, array3))
print(combined)  # Output: [1 2 3 4 5 7 8 9]
```

- `np.split()`: Splits an array into sub-arrays.

```
split_arrays = np.split(data, 5)
print(split_arrays)  # Output: [array([1]), array([2]), ..., array([5])]
```

```
84  list=[1,2,3,4,5,6]
85  # print(np.reshape(list,(2,3)))
86  # print(list)
87  data=np.array(list)
88  # array3 = np.array([7, 8, 9])
89  # combined = np.concatenate((list, array3))
90  # print(combined)
91
92  print(np.split(data,3))
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

[Running] python -u "c:\Users\laksh\OneDrive\Desktop\numpy\num.py"
[array([1, 2]), array([3, 4]), array([5, 6])]
```

Searching Arrays

You can search an array for a certain value, and return the indexes that get a match.

To search an array, use the **where()** method.

```
import numpy as np

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

x = np.where(arr == 4)

print(x)
```

```
(array([3, 5, 6]),)
```

Example

Find the indexes where the values are even:

```
import numpy as np

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

x = np.where(arr%2 == 0)

print(x)
```

Search Sorted

There is a method called **searchsorted()** which performs a binary search in the array, and returns the index where the specified value would be inserted to maintain the search order.

```
import numpy as np

arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7)

print(x)
```

```
1
```

Example

Find the indexes where the value 7 should be inserted, starting from the right:

```
import numpy as np

arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7, side='right')

print(x)
```

Multiple Values

To search for more than one value, use an array with the specified values.

Example

Find the indexes where the values 2, 4, and 6 should be inserted:

```
import numpy as np

arr = np.array([1, 3, 5, 7])

x = np.searchsorted(arr, [2, 4, 6])

print(x)
```

Sorting Arrays

Sorting means putting elements in an *ordered sequence*.

Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called sort(), that will sort a specified array.

```
import numpy as np

arr = np.array([3, 2, 0, 1])

print(np.sort(arr))
```

```
[0 1 2 3]
```

```
import numpy as np

arr = np.array(['banana', 'cherry', 'apple'])

print(np.sort(arr))
```

```
['apple' 'banana' 'cherry']
```

```python
import numpy as np

arr = np.array([[3, 2, 4], [5, 0, 1]])

print(np.sort(arr))
```

```
[[2 3 4]
 [0 1 5]]
```

Filtering Arrays

Getting some elements out of an existing array and creating a new array out of them is called *filtering*.

In NumPy, you filter an array using a *boolean index list*.

If the value at an index is True that element is contained in the filtered array, if the value at that index is False that element is excluded from the filtered array.

Example

Create an array from the elements on index 0 and 2:

```python
import numpy as np

arr = np.array([41, 42, 43, 44])

x = [True, False, True, False]

newarr = arr[x]

print(newarr)
```

The example above will return `[41, 43]`, why?

Random Number Operations

```python
np.random.rand(3, 3)  # 3x3 matrix with random values (0 to 1)
np.random.randint(1, 10, (2, 2))  # Random integers from 1 to 10
```

5. Random Numbers:
- `np.random.rand()`: Generates random numbers in the range [0, 1].

```
random_array = np.random.rand(3)
print(random_array)  # Output: [0.5488135  0.71518937 0.60276338]
```

- `np.random.randint()`: Generates random integers within a range.

```
random_integers = np.random.randint(0, 10, 5)
print(random_integers)  # Output: [3 7 9 2 4]
```

- `np.random.normal()`: Creates a normal distribution.

```
normal_dist = np.random.normal(0, 1, 5)
print(normal_dist)  # Output: [-0.10321885  0.4105985  0.14404357 -1.45427351
0.76103773]
```

```
94  print(np.random.rand())
95  otp=""
96  for i in np.random.randint(1,10,6):
97      otp+=str(i)
98  print(otp)
99
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[Running] python -u "c:\Users\laksh\OneDrive\Desktop\null
0.44765357711326426
151327
```

Broadcasting in numPy

Broadcasting in NumPy

Broadcasting is a powerful feature in NumPy that allows you to perform operations on arrays of different shapes. It enables arithmetic operations on arrays without explicitly reshaping or replicating them. This feature simplifies coding and makes computations efficient by avoiding unnecessary memory usage.

How Broadcasting Works

When performing operations on two arrays, NumPy compares their shapes element by element. Broadcasting is applied when the following rules are met:

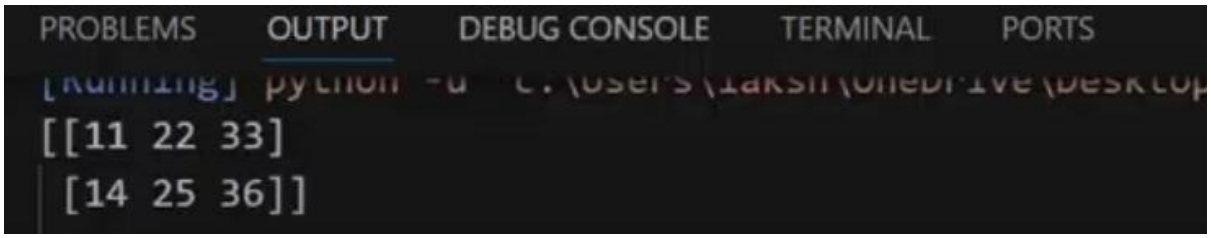
When Broadcasting Fails

Broadcasting fails when the dimensions of the arrays are incompatible. For example:

```
array1 = np.array([1, 2, 3])
array2 = np.array([[1, 2], [3, 4]])
result = array1 + array2  # This will raise a ValueError
```

The shapes (3,) and (2, 2) are incompatible for broadcasting.

```
124    array1 = np.array([[1, 2, 3], [4, 5, 6]])
125    array2 = np.array([10, 20, 30])
126    result = array1 + array2
127    print(result)
```



A scalar value is automatically broadcasted to match the shape of the array.

```
import numpy as np
array = np.array([1, 2, 3])
result = array + 5
print(result)  # Output: [6 7 8]
```

Here, the scalar 5 is "broadcast" across all elements of the array.

Adding Two Arrays of Different Shapes

When one array has a shape (m, n) and the other has a shape (1, n), broadcasting allows the smaller array to expand along the missing dimension.

```
array1 = np.array([[1, 2, 3], [4, 5, 6]])
array2 = np.array([10, 20, 30])
result = array1 + array2
print(result)
# Output:
```

Pandas

What is Pandas?

- Pandas is a Python library used for working with data sets
- Pandas is a **powerful library** for **data analysis and manipulation** in Python. It provides **DataFrames** and **Series**, which make handling structured data easy.
- It has functions for analyzing, cleaning, exploring, and manipulating data.
- The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

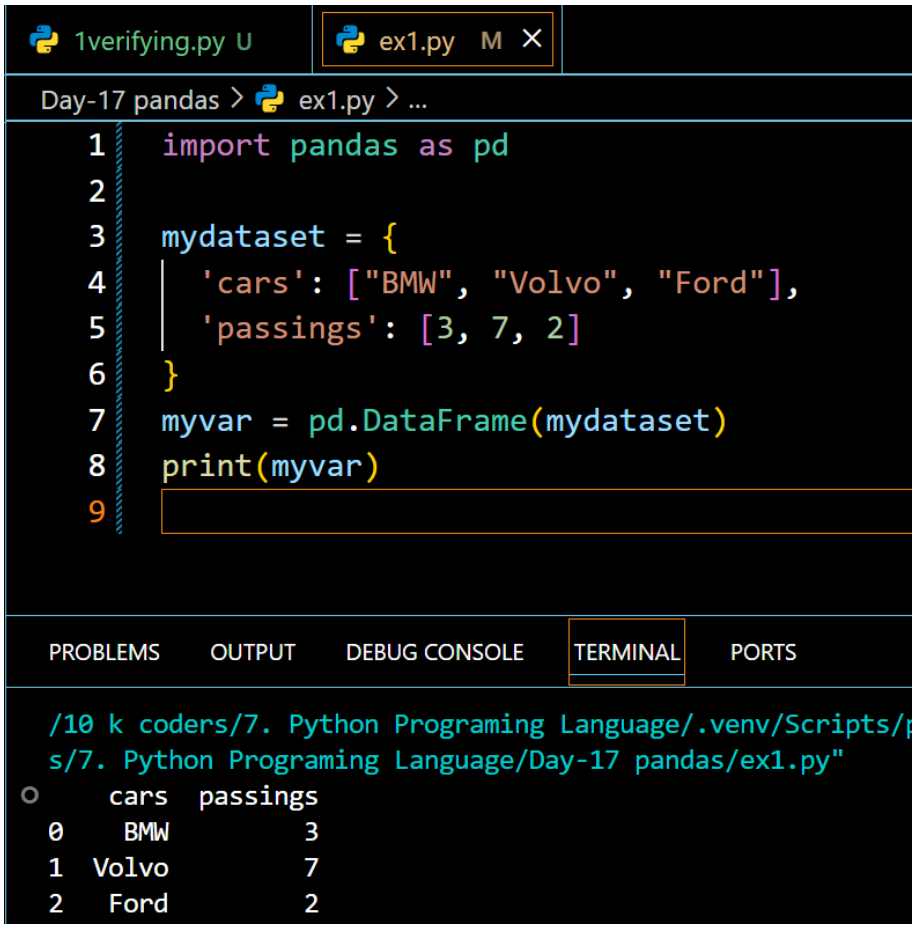
Installation of pandas:

To install pandas, follow these steps based on your setup:
1. Using pip (Recommended)
If you have Python installed, you can install pandas using pip:
pip install pandas
If you need to upgrade pandas, use:
pip install --upgrade pandas

Verifying Installation

After installation, check if pandas is installed by running:

import pandas as pd
print(pd.__version__)

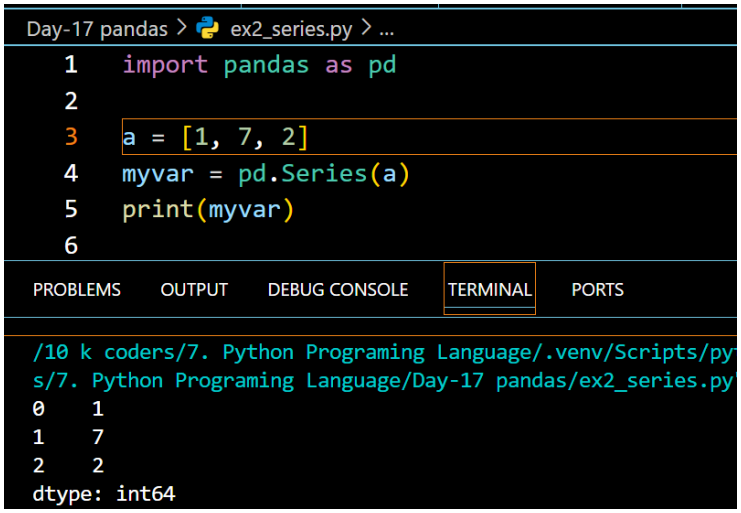


Pandas Series

What is a Series?

A Pandas Series is like a column in a table.

It is a one-dimensional array holding data of any type.



Labels

If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.

This label can be used to access a specified value.

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a)
print(myvar[1])
```

```
7
```

Create Labels

With the index argument, you can name your own labels.

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a, index = ["x", "y", "z"])
```

```
x    1
y    7
z    2
dtype: int64
```

Key/Value Objects as Series

You can also use a key/value object, like a dictionary, when creating a Series.

```
import pandas as pd

calories = {"day1": 420, "day2": 380, "day3": 390}

myvar = pd.Series(calories)

print(myvar)
```

```
day1    420
day2    380
day3    390
dtype: int64
```

To select only some of the items in the dictionary, use the index argument and specify only the items you want to include in the Series.

```
import pandas as pd

calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories, index = ["day1", "day2"])
print(myvar)
```

```
day1    420
day2    380
dtype: int64
```

DataFrames

A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
df = pd.DataFrame(data)
print(df)
```

```
   calories  duration
0       420        50
1       380        40
2       390        45
```

Locate Row

As you can see from the result above, the DataFrame is like a table with rows and columns.
Pandas use the loc attribute to return one or more specified row(s)

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
#load data into a DataFrame object:
df = pd.DataFrame(data)
print(df.loc[0])
```

```
calories    420
duration     50
Name: 0, dtype: int64
```

Example

Return row 0 and 1:

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
#load data into a DataFrame object:
df = pd.DataFrame(data)
print(df.loc[[0, 1]])
```

```
   calories  duration
0       420        50
1       380        40
```

Named Indexes

With the index argument, you can name your own indexes.

Example

Add a list of names to give each row a name:

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
df = pd.DataFrame(data, index = ["day1", "day2", "day3"])
print(df)
```

```
      calories  duration
day1       420        50
day2       380        40
day3       390        45
```

Locate Named Indexes

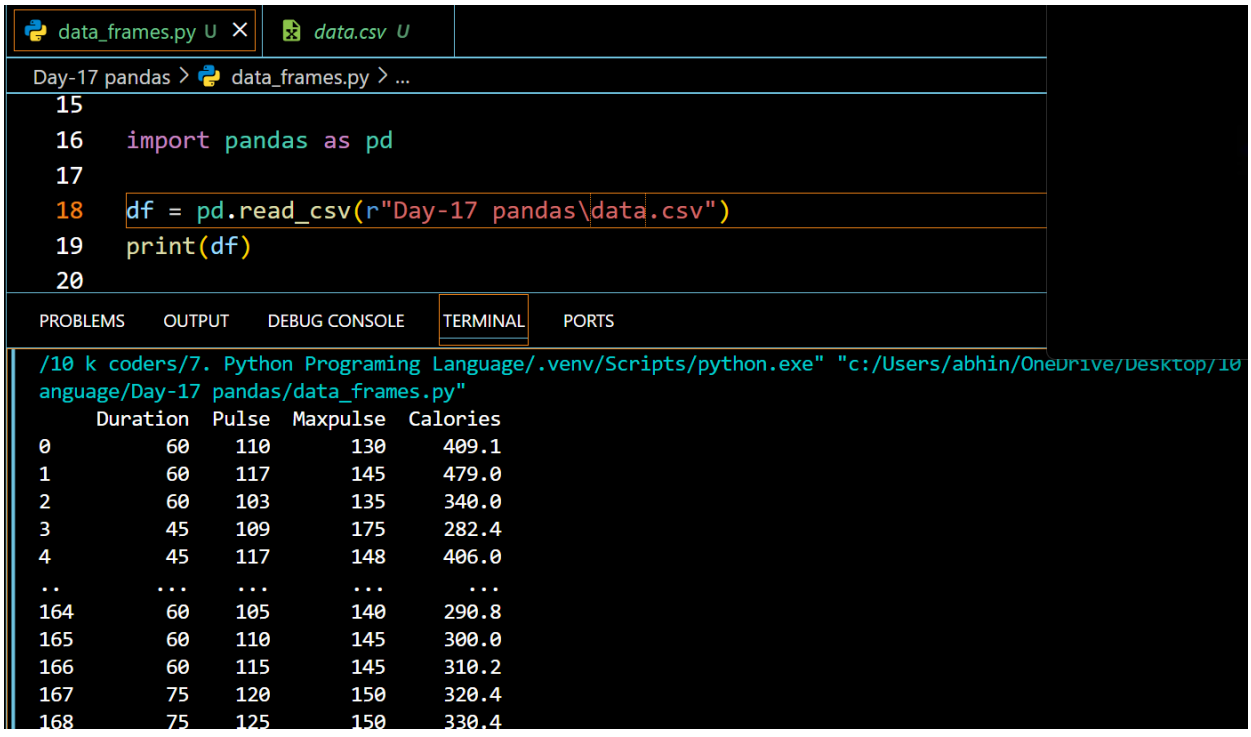
Use the named index in the loc attribute to return the specified row(s).

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
df = pd.DataFrame(data, index = ["day1", "day2", "day3"])
print(df.loc["day2"])
```

```
calories    380
duration     40
Name: day2, dtype: int64
```

Read CSV Files

- A simple way to store big data sets is to use CSV files (comma separated files).
- CSV files contains plain text and is a well know format that can be read by everyone including Pandas.
- In our examples we will be using a CSV file called 'data.csv'.
- Download data.csv. or Open data.csv.



```
Day-17 pandas > data_frames.py > ...
15
16  import pandas as pd
17
18  df = pd.read_csv(r"Day-17 pandas\data1.csv")
19  print(df.loc[1])
20
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\abhin\OneDrive\Desktop\10 k coders\7. Python Programing Language>
/10 k coders/7. Python Programing Language/.venv/Scripts/python.exe" "c:/User
s/7. Python Programing Language/Day-17 pandas/data_frames.py"
Duration      60.0
  Pulse      103.0
  Maxpulse   135.0
  Calories   340.0
Name: 1, dtype: float64
```

Tip: use to_string() to print the entire DataFrame.

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.to_string())
```

```
    Duration  Pulse  Maxpulse  Calories
0         60    110       130     409.1
1         60    117       145     479.0
2         60    103       135     340.0
3         45    109       175     282.4
4         45    117       148     406.0
5         60    102       127     300.5
6         60    110       136     374.0
7         45    104       134     253.3
8         30    109       133     195.1
9         60     98       124     269.0
```

Read JSON

Big data sets are often stored, or extracted as JSON.
JSON is plain text, but has the format of an object, and is well known in the world of programming, including Pandas.
In our examples we will be using a JSON file called 'data.json'

```
import pandas as pd
df = pd.read_json('data.json'
print(df.to_string())
```

```
    Duration  Pulse  Maxpulse  Calories
0         60    110       130     409.1
1         60    117       145     479.0
2         60    103       135     340.0
3         45    109       175     282.4
4         45    117       148     406.0
5         60    102       127     300.5
6         60    110       136     374.0
7         45    104       134     253.3
8         30    109       133     195.1
9         60     98       124     269.0
10        60    103       147     329.3
11        60    100       120     250.7
```

Basic Methods:

Display the First Few Rows:

```
print(df.head()) # Default is 5 rows
print(df.head(3)) # First 3 rows
```

Display the Last Few Rows:

```
print(df.tail()) # Default is 5 rows
```

Get DataFrame Information:

```
print(df.info())
```

Get Summary Statistics:

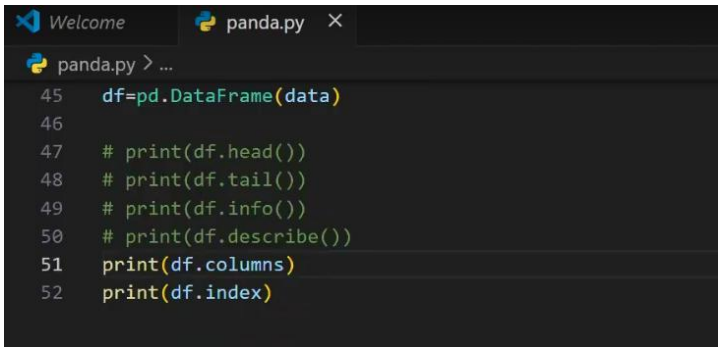
```
print(df.describe())
```

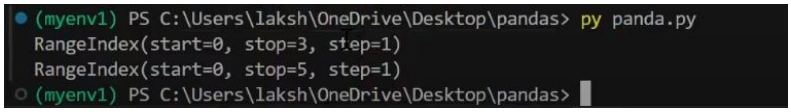
Get Column Names:

```
print(df.columns)
```

Get Index Information:

```
print(df.index)
```





Selecting methods

Selecting a Single Column:

```
print(df["Name"])
```

Selecting Multiple Columns:

```
print(df[["Name", "Age"]])
```

Selecting Rows by Index:

```
print(df.loc[0])  # Select the first row
```

Selecting Rows and Columns:

```
print(df.loc[0, "Name"])  # Row 0, Column "Name"
```

Selecting Using iloc (Position-Based Indexing):

```
print(df.iloc[0, 1])  # First row, second column
```

4. Filtering Data

Filtering Rows Based on a Condition:

```
filtered_df = df[df["Age"] > 30]
print(filtered_df)
```

Filtering Rows Using Multiple Conditions:

```
filtered_df = df[(df["Age"] > 25) & (df["City"] == "Chicago")]
print(filtered_df)
```

5. Modifying Data

Adding a New Column:

```
df["Salary"] = [50000, 60000, 70000]
print(df)
```

Updating a Column:

```
df["Age"] = df["Age"] + 5
print(df)
```

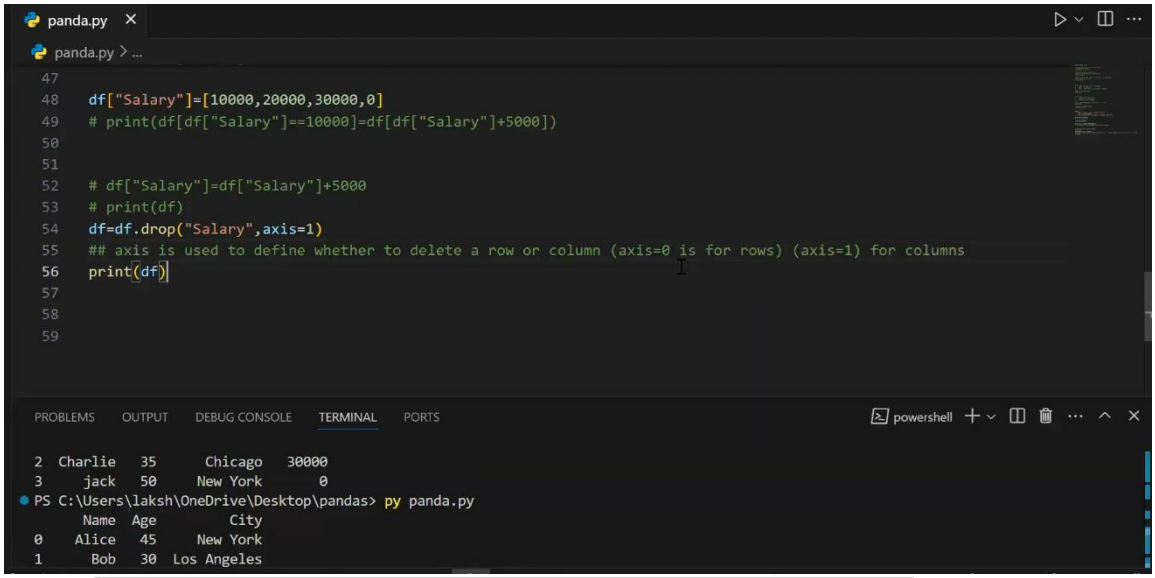
Deleting a Column:

```
df = df.drop("Salary", axis=1)
print(df)
```

panda.py > ...

```
43  df=pd.DataFrame(data)
44
45  # print(df["Name"])
46  # print(df["City"])
47
48  df["Salary"]=[10000,20000,30000,0]
49  print(df[df["Salary"]==10000])
50
```

```
PS C:\Users\laksh\OneDrive\Desktop\pandas> py panda.py
    Name  Age      City  Salary
0  Alice   45  New York   10000
PS C:\Users\laksh\OneDrive\Desktop\pandas>
```



6. Sorting Data

Sorting by a Column:

```
sorted_df = df.sort_values("Age")
print(sorted_df)
```

Sorting in Descending Order:

```
sorted_df = df.sort_values("Age", ascending=False)
print(sorted_df)
```

Sorting by Multiple Columns:

```
sorted_df = df.sort_values(["Age", "City"])
print(sorted_df)
```

7. Handling Missing Data

Detecting Missing Values:

```
print(df.isnull())
```

Filling Missing Values:

```
df["Age"] = df["Age"].fillna(30)
print(df)
```

Dropping Rows with Missing Values:

```
df = df.dropna()
print(df)
```

```
62  df=pd.DataFrame([{"a":10,"b":20},{"a":10,"b":20,"c":40}])
63  print(df)
64
65
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

1      Bob   30  Los Angeles   20000
PS C:\Users\laksh\OneDrive\Desktop\pandas> py panda.py
    a   b     c
0  10  20   NaN
1  10  20  40.0
PS C:\Users\laksh\OneDrive\Desktop\pandas>
```

```
61
62  df=pd.DataFrame([{"a":10,"b":20},{"a":10,"b":20,"c":40}])
63  # print(df.isnull())
64  print(df["c"].fillna(0))
65
66
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

Name: c, dtype: float64
PS C:\Users\laksh\OneDrive\Desktop\pandas> py panda.py
0     0.0
1    40.0
Name: c, dtype: float64
PS C:\Users\laksh\OneDrive\Desktop\pandas>
```

```
62  df=pd.DataFrame([{"a":10,"b":20},{"a":10,"b":20,"c":40}])
63  print(df.isnull())
64  print(df.dropna())
65
66
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

       a      b      c
0  False  False   True
1  False  False  False
    a   b     c
1  10  20  40.0
```

8. Grouping and Aggregating

Grouping Data by a Column:

```
grouped = df.groupby("City").mean()
print(grouped)
```

Aggregating Data:

```
aggregated = df.groupby("City").agg({"Age": "max", "Salary": "sum"})
print(aggregated)
```

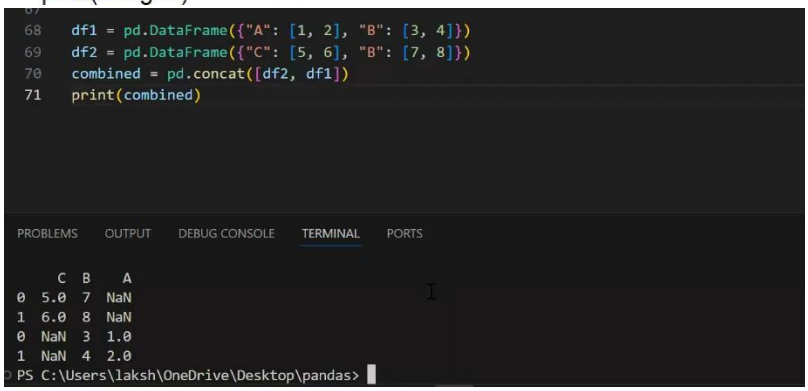
9. Combining DataFrames

Concatenating DataFrames:

```
df1 = pd.DataFrame({"A": [1, 2], "B": [3, 4]})
df2 = pd.DataFrame({"A": [5, 6], "B": [7, 8]})
combined = pd.concat([df1, df2])
print(combined)
```

Merging DataFrames:

```
df1 = pd.DataFrame({"Name": ["Alice", "Bob"], "Age": [25, 30]})
df2 = pd.DataFrame({"Name": ["Alice", "Bob"], "City": ["New York", "Chicago"]})
merged = pd.merge(df1, df2, on="Name")
print(merged)
```



Writing to a CSV file:

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

Removing Duplicates

To discover duplicates, we can use the duplicated() method.
The duplicated() method returns a Boolean values for each row:

Pandas - Data Correlations

Finding Relationships

A great aspect of the Pandas module is the corr() method.

The corr() method calculates the relationship between each column in your data set.

The examples in this page uses a CSV file called: 'data.csv'.

Download data.csv. or Open data.csv

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.corr())
```

```
          Duration     Pulse  Maxpulse  Calories
Duration  1.000000 -0.059452 -0.250033  0.344341
Pulse    -0.059452  1.000000  0.269672  0.481791
Maxpulse -0.250033  0.269672  1.000000  0.335392
Calories  0.344341  0.481791  0.335392  1.000000
```

Result Explained

- The Result of the corr() method is a table with a lot of numbers that represents how well the relationship is between two columns.
- The number varies from -1 to 1.
- 1 means that there is a 1 to 1 relationship (a perfect correlation), and for this data set, each time a value went up in the first column, the other one went up as well.
- 0.9 is also a good relationship, and if you increase one value, the other will probably increase as well.
- -0.9 would be just as good relationship as 0.9, but if you increase one value, the other will probably go down.
- 0.2 means NOT a good relationship, meaning that if one value goes up does not mean that the other will.

Plotting

- Pandas uses the plot() method to create diagrams.
- We can use Pyplot, a submodule of the Matplotlib library to visualize the diagram on the screen.
- Read more about ploting

Matplotlib

What is Matplotlib?

- Matplotlib is a low level graph plotting library in python that serves as a visualization utility.
- Matplotlib was created by John D. Hunter.
- Matplotlib is open source and we can use it freely.
- Matplotlib is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.

Installing Matplotlib

If not installed, use:

pip install matplotlib

Import Matplotlib

Once Matplotlib is installed, import it in your applications by adding the import *module* statement:

import matplotlib

Now Matplotlib is imported and ready to use:

Checking Matplotlib Version

The version string is stored under __version__ attribute.

Example

```
import matplotlib
print(matplotlib.__version__)
```

Importing Pyplot

```
import matplotlib.pyplot as plt
```

Pyplot

pyplot is a **module in Matplotlib** used for **creating visualizations** like line plots, bar charts, histograms, and scatter plots. It provides an **interface similar to MATLAB** and makes it easy to plot and customize graphs.

- Plotting x and y points
- The plot() function is used to draw points (markers) in a diagram.
- By default, the plot() function draws a line from point to point.
- The function takes parameters for specifying points in the diagram.
- Parameter 1 is an array containing the points on the x-axis.
- Parameter 2 is an array containing the points on the y-axis.

Markers

You can use the keyword argument marker to emphasize each point with a specified marker:

Marker Reference

You can choose any of these markers:

Marker	Description
'o'	Circle
'*'	Star
'.'	Point
','	Pixel
'x'	X
'X'	X (filled)
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond

'd'	Diamond (thin)
'p'	Pentagon
'H'	Hexagon
'h'	Hexagon
'v'	Triangle Down
'^'	Triangle Up
'<'	Triangle Left
'>'	Triangle Right
'1'	Tri Down
'2'	Tri Up
'3'	Tri Left
'4'	Tri Right
'|'	Vline
'_'	Hline

Line Reference

Line Syntax	Description
'-'	Solid line
':'	Dotted line
'--'	Dashed line
'-.'	Dashed/dotted line

Color Reference

Color Syntax	Description
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

Marker Size

You can use the keyword argument **markersize** or the shorter version, **ms** to **set the size of the markers:**

Example

Set the size of the markers to 20:

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

ypoints = np.array([3, 8, 1, 10])
plt.plot(ypoints, marker = 'o', ms = 20)
plt.show()
```

Marker Color

You can use the keyword argument **markeredgecolor** or the shorter **mec** to **set the color of the *edge* of the markers**:

Example

Set the EDGE color to red:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mec = 'r')
plt.show()
```

Result:

You can use the keyword argument **markerfacecolor** or the shorter **mfc** to **set the color inside the edge of the markers**:

Example

Set the FACE color to red:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mfc = 'r')
plt.show()
```

Result:

Line Color

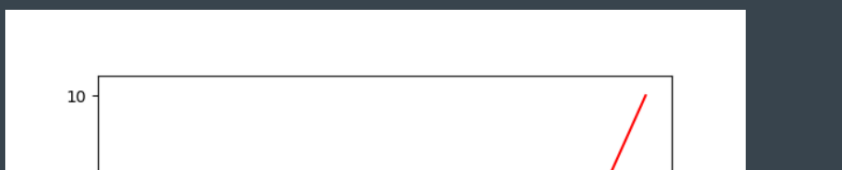
You can use the keyword argument color or the shorter **c** to **set the color of the line**:

Example

Set the line color to red:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, color = 'r')
plt.show()
```

Result:



Line Width

You can use the keyword argument linewidth or the shorter **lw** to change the width of the line.

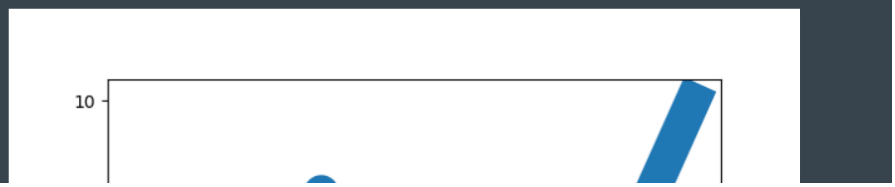
The value is a floating number, in points:

Example

Plot with a 20.5pt wide line:

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linewidth = '20.5')
plt.show()
```

Result:



Multiple Lines

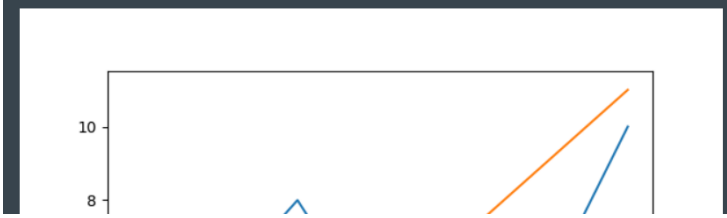
You can plot as many lines as you like by simply adding more **plt.plot()** functions:

Example

Draw two lines by specifying a `plt.plot()` function for each line:

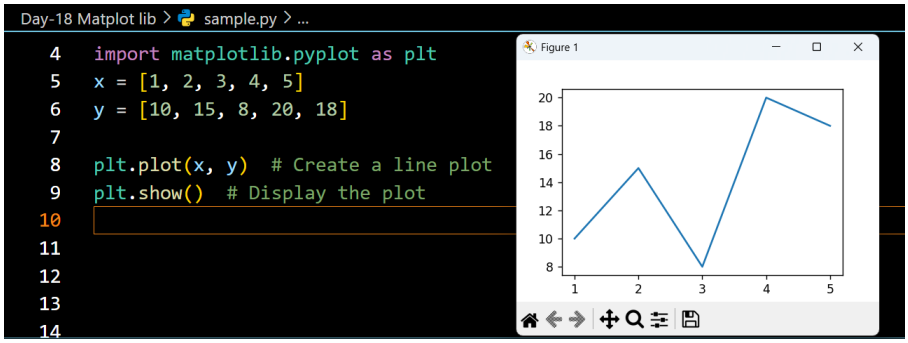
```
import matplotlib.pyplot as plt
import numpy as np

y1 = np.array([3, 8, 1, 10])
y2 = np.array([6, 2, 7, 11])

plt.plot(y1)
plt.plot(y2)

plt.show()
```

Result:



Basic Structure of Pyplot

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [10, 15, 8, 20, 18]

plt.plot(x, y)  # Create a line plot
plt.show()  # Display the plot
```

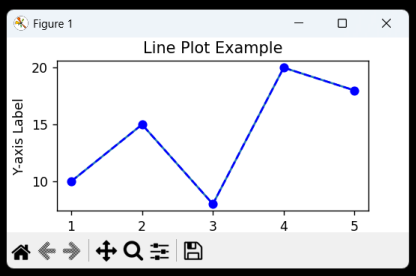


Adding Labels & Title

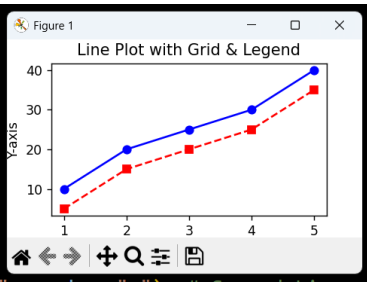
- With Pyplot, you can use the **xlabel() and ylabel() functions to set a label for the x- and y-axis**.
- With Pyplot, you can use the **title()** function to set a title for the plot.
- You can use the loc parameter in **title()** to **position the title**. Legal values are: 'left', 'right', and 'center'. Default value is 'center'.

```
plt.plot(x, y, marker="o", linestyle="--", color="b")

plt.xlabel("X-axis Label")  # Label for X-axis
plt.ylabel("Y-axis Label")  # Label for Y-axis
plt.title("Line Plot Example")  # Title of the Graph
plt.show()
```

```python
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [10, 15, 8, 20, 18]

plt.plot(x, y)  # Create a line plot
#plt.show()  # Display the plot
plt.plot(x, y, marker="o", linestyle="--", color="b")

plt.xlabel("X-axis Label")  # Label for X-axis
plt.ylabel("Y-axis Label")  # Label for Y-axis
plt.title("Line Plot Example")  # Title of the Graph
plt.show()
```




```python
import matplotlib.pyplot as plt

# Sample Data
x = [1, 2, 3, 4, 5]
y1 = [10, 20, 25, 30, 40]
y2 = [5, 15, 20, 25, 35]

# Create the plot
plt.plot(x, y1, label="Line 1", color="blue", marker="o")
plt.plot(x, y2, label="Line 2", color="red", linestyle="--", marker="s")  # Second Line

# Add Title & Labels
plt.title("Line Plot with Grid & Legend")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
```



Adding a Grid & Legend

- In **Matplotlib**, you can add a **grid** using **plt.grid(True)** and a **legend** using **plt.legend()**
- You can use the axis parameter in the **grid()** function to specify which grid lines to display.
- Legal values are: 'x', 'y', and 'both'. Default value is 'both'.
- You can also set the line properties of the grid, like this: grid(color = '*color*', linestyle = '*linestyle*', linewidth = *number*).

```python
import matplotlib.pyplot as plt

# Sample Data
x = [1, 2, 3, 4, 5]
y1 = [10, 20, 25, 30, 40]
y2 = [5, 15, 20, 25, 35]

# Create the plot
plt.plot(x, y1, label="Line 1", color="blue", marker="o")  # First Line
plt.plot(x, y2, label="Line 2", color="red", linestyle="--", marker="s")  # Second Line

# Add Title & Labels
plt.title("Line Plot with Grid & Legend")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")

# # Add Grid
plt.grid(True)  # Show grid lines

# # Add Legend
plt.legend()  # Show legend

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

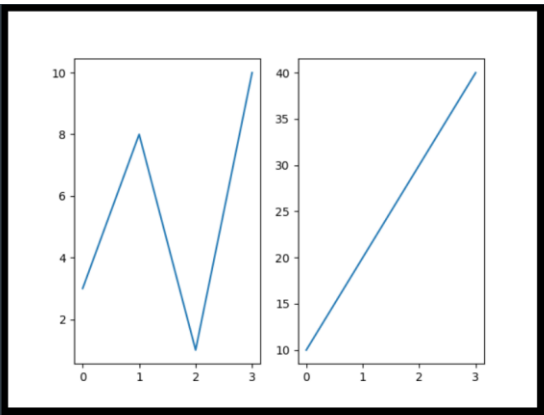
Display Multiple Plots

With the **subplot()** function you can draw multiple plots in one figure:

The subplot() Function

- The **subplot()** function takes three arguments that describes the layout of the figure.
- The layout is organized in rows and columns, which are represented by the *first* and *second* argument.
- The third argument represents the index of the current plot.

```
#Three lines to make our compiler able to draw:
import sys
import matplotlib
matplotlib.use('Agg')

import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)

plt.show()

#Two  lines to make our compiler able to draw:
plt.savefig(sys.stdout.buffer)
sys.stdout.flush()
```



Different Plot Types

Line Plot

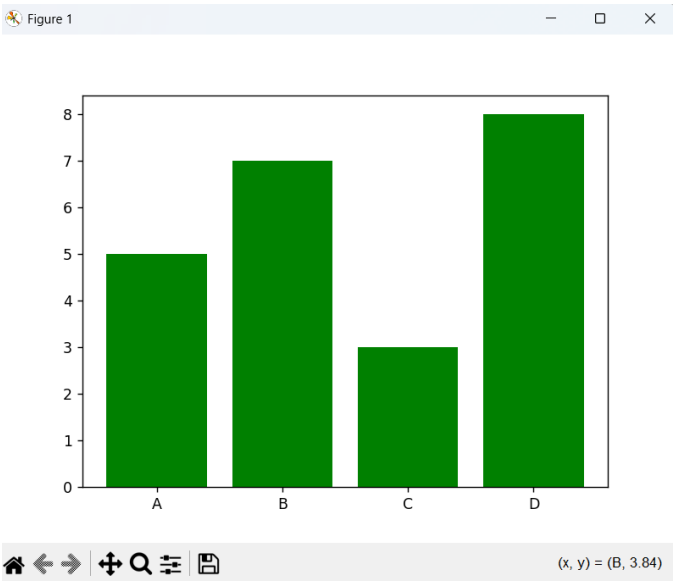
plt.plot(x, y, marker="o", linestyle="-", color="blue")
plt.show()

Bar Chart

- With Pyplot, you can use the **bar() function to draw bar graphs**
- If you want the bars to be **displayed horizontally** instead of vertically, use the **barh()** function
- The **bar() and barh()** take the keyword argument color to set the color of the bars
- plt.bar(x, y, color = "red")
- The **bar()** takes the keyword argument **width** to **set the width of the bars**
- The **barh()** takes the keyword argument **height** to **set the height of the bars**

categories = ["A", "B", "C", "D"]
values = [5, 7, 3, 8]

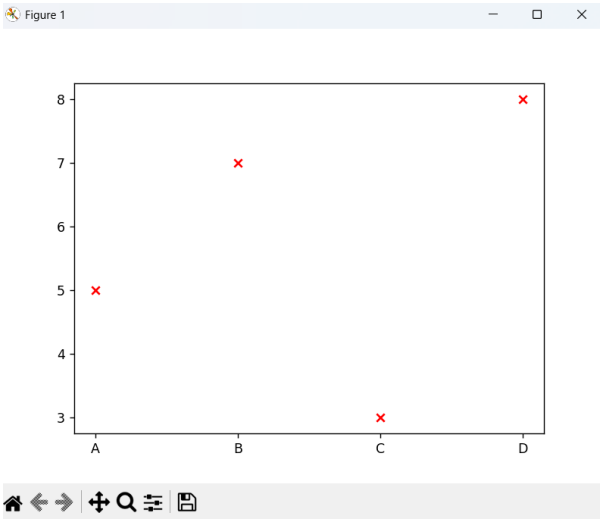
plt.bar(categories, values, color="green")
plt.show()



Scatter Plot

- With Pyplot, you can use the **scatter()** function to draw a scatter plot.
- The **scatter()** function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis
- You can even set a specific **color for each dot** by using an array of colors as value for the **c argument**
- You can change the **size of the dots** with the **s** argument.
- You can **adjust the transparency** of the dots with the **alpha** argument.

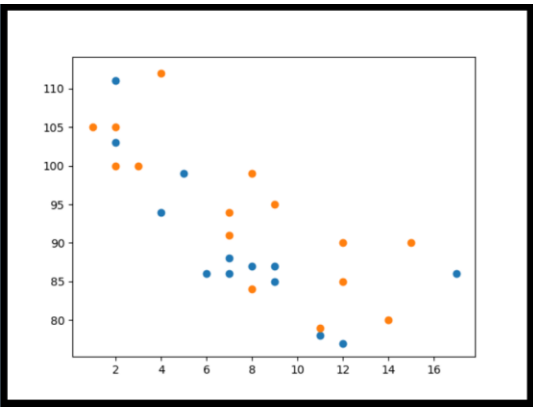
```
plt.scatter(categories, values, color="red", marker="x")
plt.show()
```



Compare Plots

In the example above, there seems to be a relationship between speed and age, but what if we plot the observations from another day as well? Will the scatter plot tell us something else?

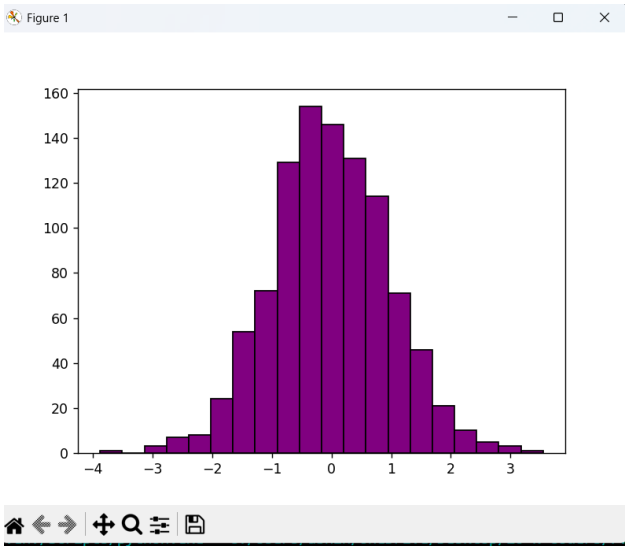
```
#Three lines to make our compiler able to draw:
import sys
import matplotlib
matplotlib.use('Agg')

import matplotlib.pyplot as plt
import numpy as np

#day one, the age and speed of 13 cars:
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y)

#day two, the age and speed of 15 cars:
x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y)

plt.show()

#Two  lines to make our compiler able to draw:
plt.savefig(sys.stdout.buffer)
sys.stdout.flush()
```



Histogram

- A histogram is a graph showing *frequency* distributions.
- It is a graph showing the number of observations within each given interval.
- The **hist()** function will read the array and produce a histogram

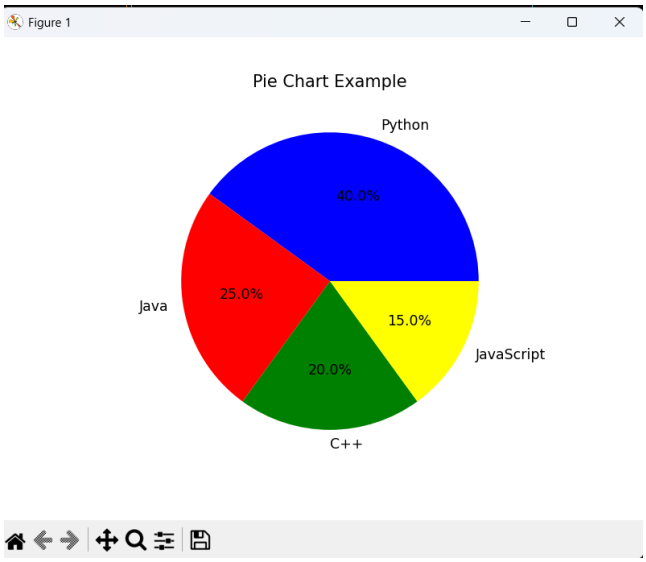
```
import numpy as np

data = np.random.randn(1000)  # Random data
plt.hist(data, bins=20, color="purple", edgecolor="black")
plt.show()
```



Pie Chart

- With Pyplot, you can use the **pie()** function to draw pie charts
- Add labels to the pie chart with the **labels** parameter.
- The labels parameter must be an array with one label for each wedge
- Maybe you want one of the wedges to stand out? The explode parameter allows you to do that.
- The **explode parameter**, if specified, and not None, must be an array with one value for each wedge.
- **Add a shadow** to the pie chart by setting the **shadows** parameter to True
- To **add a list of explanation for each wedge**, use the **legend()** function
- To **add a header to the legend**, add the **title parameter to the legend function**.

```
labels = ["Python", "Java", "C++", "JavaScript"]
sizes = [40, 25, 20, 15]

plt.pie(sizes, labels=labels, autopct="%1.1f%%", colors=["blue", "red", "green", "yellow"])
plt.title("Pie Chart Example")
plt.show()
```



Object-Oriented Programming (OOP)

What is OOP?

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects, which represent real-world entities. These objects encapsulate **data (attributes)** and **behavior (methods)**, making the code more modular and reusable.

(or)

"OOPS" uses objects, which are created from classes, to structure data and code, enabling modular, reusable, and scalable code through concepts like encapsulation, inheritance, polymorphism, and abstraction.

Key Concepts of OOP in Python

1. **Class** – A blueprint for creating objects.
2. **Object** – An instance of a class.
3. **Encapsulation** – Hiding the internal details of an object and restricting direct access to some of its components.
4. **Abstraction** – Hiding complex implementation details and exposing only the necessary parts.
5. **Inheritance** – A mechanism that allows a new class to derive properties and behavior from an existing class.
6. **Polymorphism** – The ability to take multiple forms, e.g., methods with the same name but different implementations in different classes.

OOP follows four main principles:

1. **Encapsulation** – Hiding the internal state of an object and restricting access to it.
2. **Abstraction** – Hiding complex implementation details and exposing only the necessary parts.
3. **Inheritance** – Enabling new classes to derive properties and methods from existing ones.
4. **Polymorphism** – Allowing the same interface to be used for different underlying data types.

Why Do We Use OOP?

OOP is used because it provides:

✅ **Better Code Organization** – It structures code into objects, making it more readable and maintainable.
✅ **Reusability** – Classes can be reused in different parts of a program, reducing redundancy.
✅ **Scalability** – Easier to expand and modify code without affecting other parts of the program.
✅ **Data Security** – Encapsulation ensures that sensitive data is not directly accessible.
✅ **Code Maintenance** – The modular approach allows updates and debugging without affecting the entire system.

Example Without OOP (Procedural Approach)

```
# Procedural Programming
car_brand = "Toyota"
car_model = "Corolla"

def display_info():
    print(f"Car: {car_brand} {car_model}")

display_info()
```

❌ The above approach lacks **encapsulation**, **reusability**, and **modularity**.

Example With OOP (Object-Oriented Approach)

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

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

car1 = Car("Toyota", "Corolla")
car1.display_info()
```

✅ **More structured, reusable, and scalable!**

Difference Between OOP in Python and Other Languages

Difference Between OOP in Python and Other Languages

Feature	Python	Java	C++	C#
Syntax	Simple & Dynamic	Strict & Verbose	Complex	Similar to Java
Multiple Inheritance	✅ Yes	❌ No (only interfaces)	✅ Yes	❌ No (only interfaces)
Encapsulation	✅ Uses `_` and `__` (weak enforcement)	✅ Uses `private`, `protected`, `public`	✅ Strong encapsulation	✅ Strong encapsulation
Polymorphism	✅ Dynamic typing	✅ Strict typing	✅ Compile-time & runtime	✅ Strong polymorphism
Memory Management	✅ Automatic (Garbage Collection)	✅ Automatic (JVM)	❌ Manual	✅ Automatic
Performance	Slower (interpreted)	Faster (compiled)	Very fast	Faster than Python

Key Differences:

1. Python is more **dynamic** and **flexible**, while Java, C++, and C# require **strict type declarations**.
2. Python supports **multiple inheritance**, while Java and C# use interfaces.
3. Python is **interpreted**, making it slower than compiled languages like C++ and Java.
4. Python has **automatic memory management**, whereas C++ requires manual handling.

Benefits of OOP

✅ **1. Code Reusability (DRY Principle)**

- Inheritance allows reusing existing code without rewriting it.
- Example: A Car class can be reused in multiple projects.

✅ **2. Modularity (Easier Code Organization)**

- Classes help break down large code into smaller, manageable parts.
- Example: A BankAccount class can have different methods like deposit() and withdraw().

✅ **3. Encapsulation (Data Security)**

- Prevents direct modification of sensitive data.
- Example: A private __balance attribute in a BankAccount class.

✅ **4. Scalability & Flexibility**

- Polymorphism allows the same method to work for different types.
- Example: A make_sound() method can work for both Dog and Cat classes.

✅ **5. Real-World Representation**

- Objects model real-world entities, making software design intuitive.
- Example: Car, Employee, Customer, etc.

What is a Class?

A **class** in Python is a blueprint or template for creating objects. It defines attributes (variables) and methods (functions) that the objects will have.

Syntax of a Class in Python

```
class ClassName:
    # Constructor (optional)
    def __init__(self, param1, param2):
        self.param1 = param1
        self.param2 = param2
```

```
    # Method
    def method_name(self):
        print("Method executed")
```

- class keyword is used to define a class.
- __init__() is a special method (constructor) that initializes object attributes.
- self represents the instance of the class.

What is an Object?

An **object** is an instance of a class. When a class is defined, no memory is allocated until an object is created.

Creating an Object in Python

```
# Define a class
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

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

# Creating an object of the class
car1 = Car("Toyota", "Corolla")
car1.display_info()
```

Explanation:

- Car is a class that has attributes brand and model and a method display_info().
- car1 = Car("Toyota", "Corolla") creates an object of the class.
- car1.display_info() calls the method to display the details of the car.

---

1. Class Attributes and Instance Attributes

- **Class Attributes** are shared among all objects of the class.
- **Instance Attributes** are unique to each object.

```
class Car:
    wheels = 4  # Class attribute (common to all instances)

    def __init__(self, brand, model):
        self.brand = brand  # Instance attribute
        self.model = model  # Instance attribute

car1 = Car("Toyota", "Corolla")
car2 = Car("Honda", "Civic")
```

```
print(car1.wheels)  # Output: 4
print(car2.wheels)  # Output: 4
print(car1.brand)   # Output: Toyota
print(car2.brand)   # Output: Honda
```

---

2. The __init__() Method (Constructor)

- It initializes attributes when an object is created.
- It automatically runs when an object is instantiated.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person1 = Person("Alice", 25)
print(person1.name)  # Output: Alice
print(person1.age)   # Output: 25
```

---

3. Methods in a Python Class – Detailed Explanation

In Python, methods are functions defined inside a class that operate on class attributes. There are three types of methods:

1. Instance Methods – Work on specific objects (instances) of a class.
2. Class Methods – Work on the class itself, not on instances.
3. Static Methods – Independent functions inside a class that don't modify class or instance attributes.

1. Instance Methods (self)

Definition

- Instance methods work with **specific instances** of a class.
- They **can modify instance attributes** and **access instance-specific data**.
- They take self as the first parameter, which represents the instance of the class.

Example of Instance Method

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand  # Instance variable
        self.model = model  # Instance variable

    def display_info(self):  # Instance method
        print(f"Car: {self.brand} {self.model}")

# Creating an object
car1 = Car("Toyota", "Corolla")
car1.display_info()  # Output: Car: Toyota Corolla
```

Key Points

✅ Uses self to access instance attributes.
✅ Can **read and modify** instance variables.
✅ Called using an **object of the class** (car1.display_info()).

---

2. Class Methods (@classmethod)

Definition

- Class methods work with **the class itself**, not with instances.
- They take cls as the first parameter, which represents the class.
- They **can modify class attributes** (but not instance attributes).
- Decorated with @classmethod.

Example of Class Method

```
class Car:
    wheels = 4  # Class variable

    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    @classmethod
    def change_wheels(cls, new_wheels):
        cls.wheels = new_wheels  # Modifying class attribute

    @classmethod
    def show_wheels(cls):
        print(f"All cars have {cls.wheels} wheels")

# Using class method without creating an object
Car.change_wheels(6)  # Changing class attribute
Car.show_wheels()  # Output: All cars have 6 wheels
```

Key Points

✅ Uses cls instead of self to access class attributes.
✅ Modifies class variables, affecting all instances.
✅ Can be called using either **a class or an instance** (Car.show_wheels() or car1.show_wheels()).

---

3. Static Methods (@staticmethod)

Definition

- Static methods **do not access instance (self) or class (cls) variables**.
- They behave like regular functions but are **inside a class** for better organization.
- Decorated with @staticmethod.

Example of Static Method

```
class MathOperations:
    @staticmethod
```

```
    def add(x, y):
        return x + y

    @staticmethod
    def multiply(x, y):
        return x * y

# Calling static methods without creating an object
print(MathOperations.add(5, 10))       # Output: 15
print(MathOperations.multiply(4, 6))   # Output: 24
```

Key Points

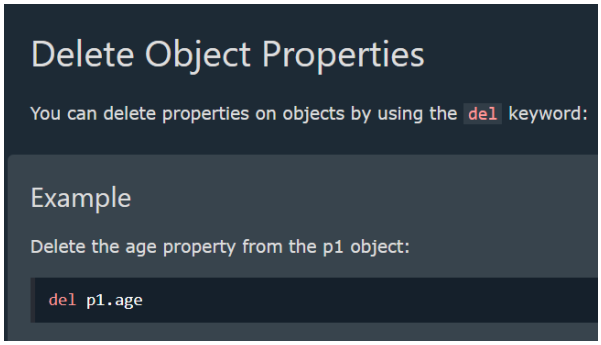
- ✅ **No self or cls**, meaning it cannot access instance or class attributes.
- ✅ Works like a normal function but is grouped logically inside a class.
- ✅ Can be called using **a class or an instance** (MathOperations.add(5, 10)).

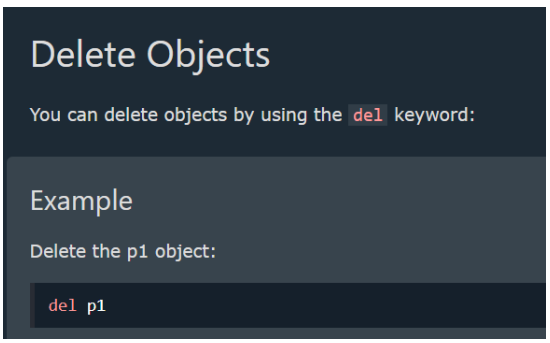
Comparison of Instance, Class, and Static Methods

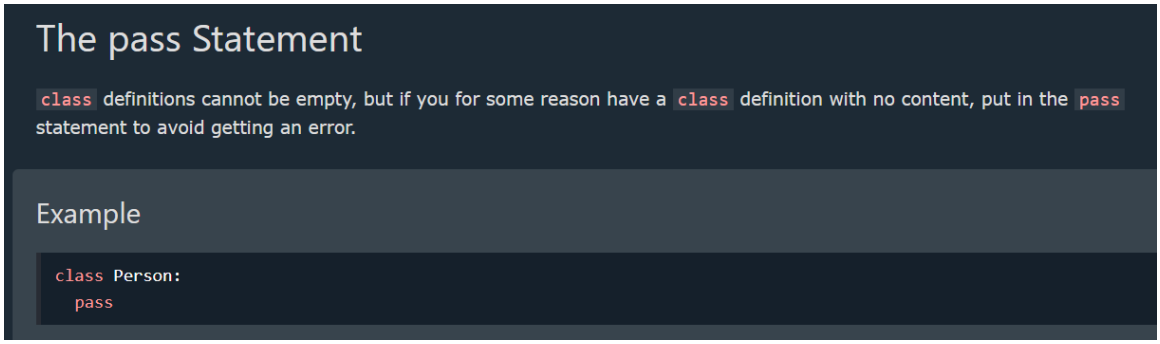
Feature	Instance Method (`self`)	Class Method (`cls`)	Static Method
Access instance attributes?	✅ Yes	❌ No	❌ No
Access class attributes?	✅ Yes	✅ Yes	❌ No
Requires `self` parameter?	✅ Yes	❌ No	❌ No
Requires `cls` parameter?	❌ No	✅ Yes	❌ No
Can modify instance variables?	✅ Yes	❌ No	❌ No
Can modify class variables?	✅ Yes	✅ Yes	❌ No
Used for?	Instance-specific operations	Class-level operations	Independent utility functions
Decorator needed?	❌ No	✅ `@classmethod`	✅ `@staticmethod`

When to Use Each Method?

Scenario	Method to Use
Need to modify or access instance attributes	**Instance Method**
Need to modify or access class attributes	**Class Method**
Need a function inside a class that does not interact with instance or class variables	**Static Method**







Inheritance

What is Inheritance?

Inheritance is a fundamental concept of Object-Oriented Programming (OOP) that allows one class (**child/subclass**) to inherit the attributes and methods of another class (**parent/base class**).

It enables **code reusability**, **hierarchical class organization**, and **extensibility** in programs.

Types of Inheritance in Python

Python supports five types of inheritance:

1. **Single Inheritance** – One child class inherits from one parent class.
2. **Multiple Inheritance** – A child class inherits from multiple parent classes.
3. **Multilevel Inheritance** – A child class inherits from another child class (grandparent → parent → child).
4. **Hierarchical Inheritance** – Multiple child classes inherit from a single parent class.
5. **Hybrid Inheritance** – A combination of two or more types of inheritance.

---

1. Single Inheritance

A child class inherits from a single parent class.

Example: Single Inheritance

```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("This animal makes a sound.")

# Child class inheriting from Animal
class Dog(Animal):
    def speak(self):  # Overriding parent method
        print(f"{self.name} barks.")
```

```
# Creating an object of the child class
dog = Dog("Buddy")
dog.speak()  # Output: Buddy barks.
```

✅ **Code reuse**: The Dog class inherits properties from Animal.
✅ **Method Overriding**: The speak() method in Dog replaces the one in Animal.

---

2. Multiple Inheritance

A child class inherits from **two or more parent classes**.

Example: Multiple Inheritance

```
# Parent class 1
class Father:
    def personality(self):
        print("Father is disciplined.")

# Parent class 2
class Mother:
    def nature(self):
        print("Mother is caring.")

# Child class inheriting from both
class Child(Father, Mother):
    def show_traits(self):
        self.personality()  # Inherited from Father
        self.nature()       # Inherited from Mother

# Creating object
child = Child()
child.show_traits()
# Output:
# Father is disciplined.
# Mother is caring.
```

✅ **Can inherit from multiple sources**.
❌ **May cause conflicts** if the same method exists in multiple parent classes.

🔹 **Note**: Python follows the **Method Resolution Order (MRO)** to decide which method to call when multiple parent classes have methods with the same name.

---

3. Multilevel Inheritance

A child class inherits from another child class, forming a **chain of inheritance** (grandparent → parent → child).

Example: Multilevel Inheritance

```
# Grandparent class
class Grandparent:
    def show_grandparent(self):
```

```
        print("I am the grandparent.")

# Parent class inheriting from Grandparent
class Parent(Grandparent):
    def show_parent(self):
        print("I am the parent.")

# Child class inheriting from Parent
class Child(Parent):
    def show_child(self):
        print("I am the child.")

# Creating an object of Child
child = Child()
child.show_grandparent()  # Output: I am the grandparent.
child.show_parent()       # Output: I am the parent.
child.show_child()        # Output: I am the child.
```

✅ **Creates a hierarchy of classes**.
✅ **Allows gradual specialization of behaviors**.
❌ **Too many inheritance levels** can make code complex and harder to manage.

---

4. Hierarchical Inheritance

Multiple child classes inherit from a **single parent class**.

Example: Hierarchical Inheritance

```
# Parent class
class Vehicle:
    def general_info(self):
        print("Vehicles are used for transportation.")

# Child class 1
class Car(Vehicle):
    def car_info(self):
        print("Cars have four wheels.")

# Child class 2
class Bike(Vehicle):
    def bike_info(self):
        print("Bikes have two wheels.")

# Creating objects
car = Car()
car.general_info()  # Output: Vehicles are used for transportation.
car.car_info()      # Output: Cars have four wheels.

bike = Bike()
bike.general_info()  # Output: Vehicles are used for transportation.
bike.bike_info()     # Output: Bikes have two wheels.
```

✅ **Reduces code duplication** since multiple child classes use the same parent class.
❌ **If changes are made in the parent class, all child classes may be affected**.

---

5. Hybrid Inheritance

A combination of multiple types of inheritance (e.g., Multiple + Multilevel).

Example: Hybrid Inheritance

```
# Parent class
class Engine:
    def engine_info(self):
        print("Engine is essential for a vehicle.")

# Parent class 2
class Vehicle:
    def vehicle_info(self):
        print("A vehicle is used for transportation.")

# Intermediate class (inherits from Engine and Vehicle)
class Car(Engine, Vehicle):
    def car_info(self):
        print("Cars usually have 4 wheels.")

# Final subclass
class SportsCar(Car):
    def sports_car_info(self):
        print("Sports cars are fast.")

# Creating an object of SportsCar
sports_car = SportsCar()
sports_car.engine_info()     # Output: Engine is essential for a vehicle.
sports_car.vehicle_info()    # Output: A vehicle is used for transportation.
sports_car.car_info()        # Output: Cars usually have 4 wheels.
sports_car.sports_car_info() # Output: Sports cars are fast.
```

✅ **Combines different types of inheritance for flexibility**.
❌ **Can be complex** due to multiple parent-child relationships.

Method Overriding in Inheritance

A **child class can override** a method from its parent class by defining a method with the same name.

Example: Overriding a Method

```
class Parent:
    def show_message(self):
        print("This is the parent class.")

class Child(Parent):
    def show_message(self):  # Overriding parent method
        print("This is the child class.")
```

```
child = Child()
child.show_message()  # Output: This is the child class.
```

✅ **Allows customization of behavior in child classes**.
✅ **Parent class remains unchanged**.

Using super() in Inheritance

The super() function is used to **call a method from the parent class** inside the child class.

Example: Using super()

```
class Parent:
    def show_message(self):
        print("This is the parent class.")

class Child(Parent):
    def show_message(self):
        super().show_message()  # Call parent method
        print("This is the child class.")

child = Child()
child.show_message()
# Output:
# This is the parent class.
# This is the child class.
```

✅ **Avoids redundant code by reusing the parent class method**.

Key Takeaways

Feature	Description
Single Inheritance	One child inherits from one parent.
Multiple Inheritance	One child inherits from multiple parents.
Multilevel Inheritance	Parent → Child → Grandchild.
Hierarchical Inheritance	One parent, multiple children.
Hybrid Inheritance	Combination of two or more inheritance types.
Method Overriding	Child class redefines a method from the parent.
super() Function	Calls a method from the parent class in the child class.

Polymorphism in Classes

- Polymorphism allows methods in **different classes to have the same name but different behaviors**.
- Polymorphism means **"many forms."** It allows the **same function or method to be used for different types of objects**.

Example of Polymorphism

```
class Animal:
    def make_sound(self):
        print("Some generic animal sound")

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")

animals = [Dog(), Cat()]
for animal in animals:
    animal.make_sound()  # Calls the respective method in each subclass
```

Types of Polymorphism in Python

1. **Method Overriding (Runtime Polymorphism)** – Child class redefines a method from the parent class.
2. **Method Overloading (Compile-time Polymorphism) – Not directly supported in Python** (handled using default arguments).
3. **Operator Overloading** – Using operators (+, *, ==, etc.) with user-defined classes.
4. **Polymorphism with Functions and Classes** – Functions and methods working with different objects.

1. Method Overriding (Runtime Polymorphism)

A **child class provides its own implementation** of a method defined in the **parent class**.

Example: Animals Making Sounds (Method Overriding)

```
class Animal:
    def make_sound(self):
        print("Animal makes a sound.")

class Dog(Animal):
    def make_sound(self):  # Overriding parent method
        print("Dog barks.")

class Cat(Animal):
    def make_sound(self):  # Overriding parent method
        print("Cat meows.")

# Creating objects
animals = [Dog(), Cat(), Animal()]

# Using polymorphism
for animal in animals:
```

```
animal.make_sound()
```

Output

```
Dog barks.
Cat meows.
Animal makes a sound.
```

✅ **Method Overriding allows different classes to have different behaviors for the same method.**

✅ **Achieves runtime polymorphism.**

---

2. Method Overloading (Using Default Arguments in Python)

Python does not **directly support** method overloading like Java or C++. However, we can **achieve it using default arguments or variable-length arguments**.

Example: Addition Function (Method Overloading using Default Arguments)

```
class MathOperations:
    def add(self, a, b=0, c=0):
        return a + b + c

math = MathOperations()
print(math.add(5))      # Output: 5
print(math.add(5, 10))   # Output: 15
print(math.add(5, 10, 20)) # Output: 35
```

✅ **Handles different numbers of arguments using default values.**

✅ **No need to define multiple functions with different parameters.**

---

3. Operator Overloading (Magic Methods in Python)

Python allows us to **override operators** like +, -, *, etc., by defining special methods like __add__(), __sub__(), etc.

Example: Adding Two Custom Objects

```
class Book:
    def __init__(self, pages):
        self.pages = pages

    def __add__(self, other):  # Overloading +
        return Book(self.pages + other.pages)

    def __str__(self):
        return f"Total pages: {self.pages}"

book1 = Book(100)
book2 = Book(200)
book3 = book1 + book2  # Uses __add__()

print(book3)  # Output: Total pages: 300
```

✅ **Allows arithmetic operations on user-defined objects.**

✅ **Enhances the readability and usability of custom classes.**

---

4. Polymorphism with Functions and Classes

A single function can operate on **different types of objects** without modification.

Example: A Common Function for Different Shapes

```
class Circle:
    def area(self, radius):
        return 3.14 * radius * radius

class Square:
    def area(self, side):
        return side * side

# Function using polymorphism
def print_area(shape, value):
    print(f"Area: {shape.area(value)}")

# Creating objects
circle = Circle()
square = Square()

# Calling function with different objects
print_area(circle, 5)  # Output: Area: 78.5
print_area(square, 4)  # Output: Area: 16
```

✅ **Function print_area() works for both Circle and Square.**
✅ **No need for separate functions for each shape.**

Encapsulation in Python 🔒

Encapsulation is one of the core principles of **Object-Oriented Programming (OOP)**. It refers to **hiding data (variables) and restricting direct access** to them. This helps in **data protection and better control** over the attributes of a class.

◇ 1. Why Use Encapsulation?

✅ **Protects Data** – Prevents accidental modification.
✅ **Restricts Direct Access** – Variables are accessed through methods.
✅ **Improves Code Maintainability** – Controlled access via getter & setter methods.

◇ 2. Encapsulation in Python (Using Private Variables)

In Python, we make variables **private** by using **double underscores (__)** before the variable name.

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private variable
```

```
    def get_balance(self):  # Getter method to access private variable
        return self.__balance

    def deposit(self, amount):  # Public method to modify private variable
        if amount > 0:
            self.__balance += amount
            return f"Deposited {amount}. New balance: {self.__balance}"
        else:
            return "Deposit amount must be positive"

# Creating an object
account = BankAccount(1000)

# Accessing private variable using getter method
print(account.get_balance())  # ✅ Output: 1000

# Depositing money using public method
print(account.deposit(500))  # ✅ Output: Deposited 500. New balance: 1500

# Trying to access private variable directly (❌ Will not work)
# print(account.__balance)  # ❌ AttributeError: 'BankAccount' object has no attribute '__balance'
```

◇ 3. Accessing Private Variables (Name Mangling): ⚠️ Not recommended but possible

Even though private variables cannot be accessed directly, Python allows access using **name mangling** (_ClassName__variable).

print(account._BankAccount__balance) # ⚠️ Not recommended but possible

This is a **workaround**, but it **defeats the purpose of encapsulation**.

◇ 4. Using Getter and Setter Methods

We can use **getter and setter methods** to safely access and modify private variables.

```
class Student:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def get_age(self):  # Getter method
        return self.__age

    def set_age(self, new_age):  # Setter method
        if new_age > 0:
            self.__age = new_age
        else:
            print("Age must be positive")

# Creating object
s1 = Student("Alice", 20)
```

```
print(s1.get_age())  # ✅ Output: 20
s1.set_age(25)  # ✅ Modifying private variable safely
print(s1.get_age())  # ✅ Output: 25
```

🔹 5. Using @property Decorator (More Pythonic Way)

Instead of defining get and set methods manually, Python provides **@property decorators**.

```
class Employee:
    def __init__(self, salary):
        self.__salary = salary  # Private variable

    @property
    def salary(self):  # Getter method
        return self.__salary

    @salary.setter
    def salary(self, amount):  # Setter method
        if amount > 0:
            self.__salary = amount
        else:
            print("Salary must be positive")

# Creating object
e1 = Employee(50000)

print(e1.salary)  # ✅ Output: 50000 (calls the getter method)

e1.salary = 60000  # ✅ Calls the setter method
print(e1.salary)  # ✅ Output: 60000

e1.salary = -1000  # ❌ Output: "Salary must be positive"
```

Abstraction in Python (OOPs) 🚀

Abstraction is one of the core principles of **Object-Oriented Programming (OOP)**. It is the process of **hiding implementation details** and **showing only the necessary features** of an object.

🔹 1. Why Use Abstraction?

✅ **Hides unnecessary details** – Users don't need to know how things work internally.
✅ **Simplifies complex systems** – Makes code easier to maintain and use.
✅ **Improves security** – Prevents direct modification of internal data.

🔹 2. Abstraction in Python (Using Abstract Classes & Methods)

Python provides **ABC (Abstract Base Class)** from the abc module to implement abstraction.

```
from abc import ABC, abstractmethod

class Animal(ABC):  # Abstract Class
    @abstractmethod
    def make_sound(self):  # Abstract Method (No implementation)
        pass

class Dog(Animal):  # Concrete Class
    def make_sound(self):
        return "Woof! Woof!"

class Cat(Animal):  # Concrete Class
    def make_sound(self):
        return "Meow! Meow!"

# Creating objects
dog = Dog()
cat = Cat()

print(dog.make_sound())  # ✅ Output: Woof! Woof!
print(cat.make_sound())  # ✅ Output: Meow! Meow!

# animal = Animal()  ❌ Error: Cannot instantiate abstract class
```

🔹 3. Explanation

- ABC (Abstract Base Class) is used to create an **abstract class**.
- @abstractmethod forces **child classes** to implement the method.
- We **cannot create an object of an abstract class** (Animal in this case).
- Only **child classes (Dog, Cat) provide implementation** of make_sound().

🔹 4. Real-World Example: Payment System

```
from abc import ABC, abstractmethod

class Payment(ABC):  # Abstract Class
    @abstractmethod
    def process_payment(self, amount):
        pass

class CreditCardPayment(Payment):  # Concrete Class
    def process_payment(self, amount):
        return f"Processing credit card payment of ${amount}"
```

```
class PayPalPayment(Payment):  # Concrete Class
    def process_payment(self, amount):
        return f"Processing PayPal payment of ${amount}"

# Creating objects
payment1 = CreditCardPayment()
payment2 = PayPalPayment()

print(payment1.process_payment(100))  # ✅ Output: Processing credit card payment of $100
print(payment2.process_payment(200))  # ✅ Output: Processing PayPal payment of $200
```

🔹 5. Key Differences Between Abstraction & Encapsulation

Feature	Abstraction	Encapsulation
Definition	Hides **implementation details**	Hides **data** using private variables
How?	Using **abstract classes & methods**	Using **private variables & getter/setter methods**
Purpose	Focus on **what an object does**	Focus on **how an object's data is protected**
Example	`@abstractmethod` forces child classes to implement methods	`__private_variable` restricts direct access

---

🚀 Summary

✅ **Abstraction hides implementation details** using **abstract classes & methods**.
✅ ABC (Abstract Base Class) & @abstractmethod **enforce method implementation** in child classes.
✅ **Cannot create objects of abstract classes**.
✅ **Real-world applications**: Payment systems, databases, frameworks, etc.

OS Module in Python 🖥️

Python has a built-in os module with methods for interacting with the operating system, like creating files and directories, management of files and directories, input, output, environment variables, process management, etc.

---

1. Importing the OS Module

import os

---

2. Working with the File System

2.1. Get the Current Working Directory

print(os.getcwd()) # Output: Current directory path

2.2. Change the Current Directory

os.chdir("/path/to/directory") # Change working directory
print(os.getcwd()) # Verify the change

2.3. List Files and Directories

print(os.listdir()) # Lists files & folders in current directory
print(os.listdir("/path/to/directory")) # List files in a specific directory

2.4. Create a Directory

os.mkdir("new_folder") # Creates a folder
Create multiple directories (Nested Folders)
os.makedirs("parent_folder/child_folder")

2.5. Remove a Directory

os.rmdir("new_folder") # Removes a folder (only if empty)
Remove multiple directories
os.removedirs("parent_folder/child_folder")

2.6. Rename a File or Directory

os.rename("old_name.txt", "new_name.txt") # Renames a file
os.rename("old_folder", "new_folder") # Renames a directory

---

3. Working with Files

3.1. Check if a File Exists

print(os.path.exists("example.txt")) # Output: True/False

3.2. Check if a Path is a File or Directory

print(os.path.isfile("example.txt")) # True if it's a file
print(os.path.isdir("my_folder")) # True if it's a directory

3.3. Get File Size

```
print(os.path.getsize("example.txt"))  # Output: Size in bytes
```

3.4. Get Absolute Path

```
print(os.path.abspath("example.txt"))
```

---

4. Environment Variables

4.1. Get an Environment Variable

```
print(os.environ.get("HOME"))  # Output: Home directory (Linux/Mac)
print(os.environ.get("USERNAME"))  # Output: Username (Windows)
```

4.2. Set an Environment Variable

```
os.environ["MY_VARIABLE"] = "Hello, World!"
print(os.environ["MY_VARIABLE"])  # Output: Hello, World!
```

---

5. Executing System Commands

5.1. Run a Shell Command

```
os.system("ls")  # Linux/Mac - List files
os.system("dir")  # Windows - List files
```

5.2. Open a File Using Default Application

```
os.startfile("example.txt")  # Windows
os.system("open example.txt")  # Mac
os.system("xdg-open example.txt")  # Linux
```

---

6. Process Management

6.1. Get Process ID

```
print(os.getpid())  # Output: Current process ID
```

6.2. Get Parent Process ID

```
print(os.getppid())  # Output: Parent process ID
```

6.3. Create a Child Process

```
pid = os.fork()  # Only on Linux/macOS
if pid == 0:
    print("Child process")
else:
    print("Parent process")
```

---

7. Path Operations with os.path

The os.path module helps in handling **file paths**.

7.1. Join Paths

```
path = os.path.join("folder", "file.txt")
print(path)  # Output: folder/file.txt (Linux/Mac) or folder\file.txt (Windows)
```

7.2. Get File Extension

```
print(os.path.splitext("example.txt"))  # Output: ('example', '.txt')
```

7.3. Get Directory Name

```
print(os.path.dirname("/path/to/file.txt"))  # Output: /path/to
```

7.4. Get Base File Name

```
print(os.path.basename("/path/to/file.txt"))  # Output: file.txt
```

---

8. Summary of OS Module Functions

Function	Description
os.getcwd()	Get current directory
os.chdir(path)	Change directory
os.listdir(path)	List files & directories
os.mkdir(name)	Create a directory
os.rmdir(name)	Remove an empty directory
os.rename(old, new)	Rename file/directory
os.path.exists(path)	Check if path exists
os.path.isfile(path)	Check if it’s a file
os.path.isdir(path)	Check if it’s a directory
os.environ.get(var)	Get environment variable
os.system(command)	Run shell command
os.getpid()	Get process ID

Conclusion

The os module is powerful for file handling, directory management, environment variables, and system commands. Would you like practice exercises or a mini-project using the os module? 🚀

Python os Module: Complete Reference

The Python `os` module provides a way of using operating system-dependent functionality such as interacting with files and directories, accessing environment variables, and managing processes.

Working with Directories and Files

- os.mkdir(path, mode=0o777): Creates a new directory.
- os.remove(path): Deletes a file.
- os.listdir(path='.') : Lists all files and directories in the specified path.

File and Directory Manipulation

- os.path: Provides utilities for manipulating file paths.
- os.rename(src, dst): Renames a file or directory.
- os.remove(path): Deletes a file.

Path Methods (os.path)

- os.path.join(*paths): Combines path components into a single path.
- os.path.abspath(path): Returns the absolute version of the path.
- os.path.normpath(path): Normalizes a path (e.g., removing redundant separators).
- os.path.exists(path): Checks if a path exists.
- os.path.isfile(path): Checks if a path is a file.
- os.path.isdir(path): Checks if a path is a directory.
- os.path.basename(path): Returns the last component of a path.
- os.path.dirname(path): Returns the directory component of a path.
- os.path.split(path): Splits the path into (head, tail) components.
- os.path.splitext(path): Splits the file path into (root, extension).
- os.path.isabs(path): Checks if a path is absolute.
- os.path.relpath(path, start): Returns the relative path from 'start' to 'path'.
- os.path.getsize(path): Returns the size of the file in bytes.
- os.path.getmtime(path): Returns the last modification time of the file.

Environment Variables and System Information

- os.getenv(key, default=None): Gets the value of an environment variable.
- os.environ: A mapping object representing environment variables.

Other Commonly Used Methods in os Module

- os.getcwd(): Returns the current working directory.
- os.chdir(path): Changes the current working directory.
- os.rmdir(path): Removes an empty directory.
- os.makedirs(path, exist_ok=False): Creates directories recursively.
- os.walk(top, topdown=True, onerror=None, followlinks=False): Generates file names in a directory tree.
- os.system(command): Executes a system command.
- os.getpid(): Returns the current process ID.
- os.urandom(n): Returns a string of `n` random bytes.
- os.access(path, mode): Checks the accessibility of a path.

date.py

```python
import os


# combined=os.path.join("c:/mydir","./sample.txt")
# print(combined)

# print(os.path.abspath("./sample.txt"))
# print(os.path.abspath("./dates.py"))
# print(os.path.normpath("d:\Batch\\Python_classes\\Day_30\dates.py"))
# print(os.path.exists("./sample.txt"))

# print(os.path.isfile("./sample.txt"))
# print(os.path.isdir("../DAY_30"))
# print(os.path.basename("d:\Batch\Python_classes\Day_30\sample.txt"))

# print(os.path.dirname("d:\Batch\Python_classes\Day_30\date.py")+"\sample.txt")

# print(os.path.split("d:\Batch\Python_classes\Day_30\date\date.py\sample.txt"))
```

```python

# print(os.path.split("d:\Batch\Python_classes\Day_30\date\date.py\sample.txt"))

# print(os.path.splitext("d:\Batch\Python_classes\Day_30\sample.txt")[1]==".png")
# print(os.path.isabs("/date.py"))
# print(os.path.relpath("d:\Batch\Python_classes\Day_30\date.py"))
# print(os.path.getsize("d:\Batch\Python_classes\Day_30\date.py"))
# print(os.path.getmtime("d:\Batch\Python_classes\Day_30\date.py"))
```

datetime Module in Python

The datetime module in Python provides classes and functions to work with **dates, times, and time intervals**.

The datetime module in Python is used to **work with dates and times**. It allows you to get the current date/time, format it, perform date arithmetic, and more.

---

1. Importing the Module

import datetime

---

2. Getting Current Date and Time

now = datetime.datetime.now()
print(now) # Example: 2025-04-04 21:00:30.123456

To get only the **date** or **time**:
print(now.date())   # 2025-04-04
print(now.time())   # 21:00:30.123456

---

3. Get Today's Date

today = datetime.date.today()
print(today) # Example: 2025-04-04

---

4. Create a Specific Date or Time

my_date = datetime.date(2025, 12, 25)
print(my_date) # 2025-12-25

my_time = datetime.time(14, 30, 0)
print(my_time) # 14:30:00

---

5. Date Arithmetic (Add/Subtract Days)

Use timedelta for date calculations:
from datetime import timedelta

today = datetime.date.today()
tomorrow = today + timedelta(days=1)
yesterday = today - timedelta(days=1)

print("Today:", today)
print("Tomorrow:", tomorrow)
print("Yesterday:", yesterday)

Date Arithmetic (Timedelta)

➤ **Creating a timedelta**

```
from datetime import timedelta

delta = timedelta(days=5, hours=3)
print(delta)  # 5 days, 3:00:00
```

➤ **Adding/Subtracting dates**

```
now = datetime.now()
future = now + timedelta(days=7)
past = now - timedelta(days=30)

print(future)  # 7 days from now
print(past)    # 30 days ago
```

---

6. Formatting Date and Time (strftime)

Convert a datetime object to a string format:

```
from datetime import timedelta


now = datetime.datetime.now()
formatted = now.strftime("%Y-%m-%d %H:%M:%S")
print(formatted)  # 2025-04-04 21:03:00


print(now.strftime("%Y-%m-%d"))     # 2025-04-04
print(now.strftime("%d/%m/%Y"))     # 04/04/2025
print(now.strftime("%I:%M %p"))     # 02:23 PM
print(now.strftime("%A, %B %d"))    # Friday, April 04
```

Format Code	Meaning	Example
%Y	Year (4 digits)	2025
%m	Month (01 to 12)	04
%d	Day of the month	04
%H	Hour (24-hour)	14
%I	Hour (12-hour)	02
%p	AM or PM	PM
%A	Full weekday name	Friday
%B	Full month name	April

---

7. Parsing Date String (strptime)

Convert a string into a datetime object:

```
from datetime import datetime

date_string = "25/12/2023 10:30 AM"
parsed_date = datetime.strptime(date_string, "%d/%m/%Y %I:%M %p")
print(parsed_date)  # Output: 2023-12-25 10:30:00
```

---

8. Get Day, Month, Year, etc.

```
now = datetime.datetime.now()
print(now.year)    # 2025
print(now.month)   # 4
print(now.day)     # 4
print(now.hour)    # 21
print(now.minute)  # 0
```

---

9. Working with date and time classes

➤ **date object (only date)**

```
from datetime import date

today = date.today()
print(today)          # Output: 2025-04-04
print(today.year)     # 2025
```

➤ **time object (only time)**

```
from datetime import time

t = time(10, 45, 30)
print(t)              # Output: 10:45:30
print(t.hour)         # 10
```

---

10. Get Day of the Week

```
today = datetime.today()
print(today.strftime("%A"))  # Output: Friday

# Using weekday() method
print(today.weekday())  # Output: 4 (Monday=0, Sunday=6)
```

```python
from datetime import datetime
import pytz

# Current local time
print(datetime.now())

# Specifying time zone
tz = pytz.timezone("US/Pacific")
print(datetime(2025, 12, 13, tzinfo=tz))

    #or

print(datetime.now(tz))  # Current time in US/Pacific

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
aming Language/.venv/Scripts/python.exe" "c:/Users/abhin/OneDrive/Desktop/10 k coders/7. Python Pro
2025-04-05 02:36:13.109832
2025-12-13 00:00:00-07:53
2025-04-04 14:06:15.482158-07:00
```

☑ Summary Table

Function / Class	Description
datetime.now()	Current date and time
datetime.today()	Today's date
datetime.strptime()	Parse string to datetime
datetime.strftime()	Format datetime to string
datetime(year, m, d, h, m)	Create custom datetime object
date.today()	Returns current date
timedelta(days=, hours=)	Create time difference
datetime + timedelta	Future date
datetime - timedelta	Past date

💡 Real-World Example: Countdown Timer

```
from datetime import datetime, timedelta

event_date = datetime(2025, 12, 31, 23, 59)
now = datetime.now()
remaining = event_date - now

print(f"Time remaining for New Year: {remaining.days} days and {remaining.seconds // 3600} hours")
```

Would you like a **mini-project using datetime** like a digital clock, birthday countdown, or age calculator? ⏳

Database connection