Tutorial 2-Basics Of Python

This code consists of two print statements in Python. The first line outputs "Greetings, Earth" to the console, and the second line outputs "Glad to have you on my platform." Essentially, it displays two messages to the user.

```
print("Greetings, Earth")
print("Glad to have you on my platform")
```

```
Greetings, Earth
Glad to have you on my platform
```

The code includes a multi-line comment that thanks visitors to a YouTube platform, followed by a print statement that outputs "Hi there" to the console. The comment is ignored during execution.

```
# One-line and multi-line comments
'''
Hello and thanks for visiting my
YouTube platform
'''
print("Hi there")
```

```
Hi there
```

The code defines a variable num with a value of 10 and checks its data type, which will be <class 'int'>. The type check does not produce output unless explicitly printed.

```
# Defining a variable
num = 10
type(num)
```

```
int
```

prints the value of the variable num to the console.

```
print(num)
```

```
10
```

The code defines three variables:

- `value1` stores `10`.
- `value2` stores `40`.
- `name` stores `"Krish"`, a string.

```
value1 = 10
value2 = 40
name = "PRASHANT"
```

The code defines `c` with a value of `10` and then checks its data type using `type(c)`.

```
# Define the variable 'c' before using it.
c = 10  # or any other value you want to assign to c
type(c)
```

```
int
```

The code assigns the value `10` to the variable `a1`.

```
a1=10
```

The code assigns `"Krish"` to `user_name` and prints it.

```
user_name = "PRASHANT"
print(user_name)
```

```
Krish
```

The code assigns `10` to `num` and checks its data type, which is `int` (integer).

```
## Whole numbers
num = 10
type(num)
```

```
int
```

The code assigns `10` to `a` and then adds `10` to it, resulting in `20`.

```
a = 10  # Assign a value to 'a' before using it.
a + 10
```

```
20
```

Multiplies `a` by `10`. If `a = 10`, the result is `100`.

```
a*10
```

```
100
```

Divides `a` by `10`. If `a = 10`, the result is `1.0`.

```
a/10
```

```
1.0
```

Finds the remainder when `a` is divided by `10`. If `a = 10`, the result is `0`.

```
a%10
```

```
0
```

Prints the data type of `a`.

```
print(type(a))
```

```
<class 'int'>
```

This Python code initializes two variables, `a` and `b`, with values 10 and 20, respectively. The `print(a + b)` statement outputs their sum, which is **30**.

```
a=10
b=20
print(a+b)
```

```
30
```

This Python code assigns the floating-point value `190.5` to variable `a`. When `a` is called, it outputs **190.5**.

```
#floating
a=190.5
```

```
a
```

```
190.5
```

This code checks if `a` is an integer using `isinstance()`. It returns **`True`** if `a` is an `int`, otherwise **`False`**.

```
## Data type conversion
print(isinstance(a, int))
```

```
False
```

This converts `b` into a floating-point number.

```
float(b)
```

```
20.0
```

This code defines two Boolean variables: `flag1` as **True** and `flag2` as **False**.

```
## Logical Values
flag1 = True
flag2 = False
```

This returns the data type of `flag1`, which is **`bool`**.

```
type(flag1)
```

```
bool
```

This returns **`True`** because `or` returns `True` if at least one value is `True`.

```
flag1 or flag2
```

```
True
```

This code assigns `"PRASHANT"` to `user_name`, prints it, and displays its data type, which is **`str`** **(string)**.

```
# Text Data
user_name = "PRASHANT"

print(user_name)
print(type(user_name))
```

```
PRASHANT
<class 'str'>
```

This code assigns `"PRASHANT"` to `user_name`, prints it, and shows its type (`str`). Then, `name1` is defined as `"PRASHANT"`, and `"GUPTA"` is concatenated to it, resulting in `"PRASHANTGUPTA"`.

```
# Text Data
user_name = "PRASHANT"

print(user_name)
print(type(user_name))

# %%
# Define name1 before using it.
name1 = "PRASHANT" # Assuming you want to use the same value as user_name. You can change this to any string value.
name1 + "GUPTA"
```

```
PRASHANT
<class 'str'>
'PRASHANTGUPTA'
```

This concatenates `name1` (a string) with `1` converted to a string, resulting in `"PRASHANT1"`.

```
name1 + str(1)
```

```
'PRASHANT1'
```

This defines `num` as a **complex number** (`1.0 - 2.3j`), where `j` represents the imaginary unit.

```
## Imaginary numbers
num = 1.0 - 2.3j
num
```

```
(1-2.3j)
```

This prints the **real** (`1.0`) and **imaginary** (`-2.3`) parts of the complex number `num`.

```
print(num.real, num.imag)
```

```
1.0 -2.3
```

This shows **Python's dynamic typing**, where `var` is first assigned an `int` (`10`), then reassigned a `str` (`"PRASHANT"`).

```
# Flexible Typing
var = 10
var = "PRASHANT"
```

This assigns the string `"Krish"` to the variable `name`, demonstrating **strict typing** where `name` remains a string.

```
# Strict Typing
name = ""
```

Concatenates a string and a number

```
name + str(1)
```

```
'PRASHANT1'
```

print("The value is:", num) → Prints a message with a variable

num = 100 → Stores 100 in num print() → Outputs text to the console "," → Adds a space automatically between "The value is:" and num

```
# Text Formatting
```

```
num = 100
print("The value is:", num)
```

```
The value is: 100
```

format(x=fname, y=lname) → Formats and inserts variables into a string

fname = "PRASHANT" → First name lname = "GUPTA" → Last name .format(x=fname, y=lname) → Replaces {x} with fname and {y} with lname

```
# Name Formatting
```

```
fname = "PRASHANT"
lname = "GUPTA"
print("First name: {x}, Last name: {y}".format(x=fname, y=lname))
```

```
First name: PRASHANT, Last name: GUPTA
```

Takes user input, converts to integers, and adds them.

input() → Gets input as a string int(num1), int(num2) → Converts input to integers

- → Adds the two numbers print() → Displays the result

```
## User Input
num1 = input("Enter value for num1: ")
num2 = input("Enter value for num2: ")
print(int(num1) + int(num2))
```

```
Enter value for num1: 1
Enter value for num2: 6
7
```

Tutorial 3- Python Control Flows

Checks if a number is even.

- `input()` → Takes user input as a string
- `float(num)` → Converts input to a floating-point number
- `if num_float % 2 == 0:` → Checks divisibility by 2
- `print("The number is even")` → Displays result if condition is true

🔷 **Example Input & Output:**

```
Enter a number: 4
The number is even
```

```
# Conditional Statement

num = input("Enter a number: ")
num_float = float(num)
if num_float % 2 == 0:
    print("The number is even")
```

```
Enter a number: 3
```

`10 % 2 != 0` → **Checks if 10 is not odd**

- `10 % 2` → Remainder when 10 is divided by 2 (Result: `0`)
- `!= 0` → Checks if the remainder is **not** `0`
- **False**, because `10` is even.

🔷 **Expression Output:**

```
False
```

```
10%2!=0
```

```
False
```

Checks if a number is even or odd.

- `input()` → Takes user input as a string
- `float(num)` → Converts input to a floating-point number
- `if num_float % 2 == 0:` → Checks if divisible by 2

- ✅ **True** → Prints `"This is an even number"`
- ❌ **False** → Prints `"This is an odd number"`

🔷 **Example Input & Output:**

```
Enter a number: 7
This is an odd number
```

```
# If-Else Condition

num = input("Enter a number: ")
num_float = float(num)

if num_float % 2 == 0:
    print("This is an even number")
else:
    print("This is an odd number")
```

```
Enter a number: 4
This is an even number
```

Classifies age into different groups.

input() → Takes user age float(user_age) → Converts input to float Conditions: < 18 → "Minor" 18 - 45 → "Mid Age group" 46 - 50 → "Senior Mid Age group"

> 50 → "Senior Citizen" 🔷 Example Input & Output:

vbnet Copy Edit Enter your age: 30
You belong to the Mid Age group

```
## Age Classification
## Nested If-Else Condition

user_age = float(input("Enter your age: "))

if user_age < 18:
    print("You are a Minor")
elif 18 <= user_age <= 45:
    print("You belong to the Mid Age group")
elif 45 < user_age <= 50:
    print("You are in the Senior Mid Age group")
else:
    print("You are a Senior Citizen")
```

```
Enter your age: 45
You belong to the Mid Age group
```

Classifies a person based on age with nested conditions.

- `input()` → Takes user age
- `float(user_age)` → Converts input to float
- **Conditions:**
  - `< 18` → `"Minor"`
    - `< 15` → `"School"`
    - `15 - 17` → `"College"`
  - `18 - 45` → `"Mid Age group"`
  - `46 - 50` → `"Senior Mid Age group"`
  - `> 50` → `"Senior Citizen"`

🔷 **Example Input & Output:**

```
Enter your age: 14
You are a Minor
```

```
You are in School
```

```
# Nested If-Else

user_age = float(input("Enter your age: "))

if user_age < 18:
    print("You are a Minor")
    if user_age < 15:
        print("You are in School")
    else:
        print("You are in College")
elif 18 <= user_age <= 45:
    print("You belong to the Mid Age group")
elif 45 < user_age <= 50:
    print("You are in the Senior Mid Age group")
else:
    print("You are a Senior Citizen")
```

```
Enter your age: 67
You are a Senior Citizen
```

Prints the square of each number in a list.

numbers = [1, 2, 3, 4, 5, 6, 7] → List of numbers for num in numbers: → Loops through each number print(num ** 2) → Prints square of num

🔷 Output:

Copy Edit 1
4
9
16
25
36
49

Loops through a list and prints squares of numbers.

- `numbers = [1, 2, 3, 4, 5, 6, 7]` → List of numbers
- `for num in numbers:` → Iterates over each number
- `print(num ** 2)` → Prints square of each number

🔷 **Output:**

```
1
4
9
16
25
36
49
```

```
## Looping Statements
## For Loop and While Loop

numbers = [1, 2, 3, 4, 5, 6, 7]
for num in numbers:
    print(num ** 2)
```

```
1
4
9
16
25
36
49
```

Calculates the sum of all numbers in a list.

- `numbers = [1, 2, 3, 4, 5, 6, 7]` → List of numbers
- `total = 0` → Initializes sum
- `for num in numbers:` → Iterates through list
- `total += num` → Adds each number to `total`
- `print(total)` → Prints the sum

🔷 **Output:**

```
28
```

```
## Calculate the sum of all elements in the list

numbers = [1, 2, 3, 4, 5, 6, 7]
total = 0

for num in numbers:
    total += num

print(total)
```

```
28
```

Calculates sum of even and odd numbers separately.

- `sum_even = 0, sum_odd = 0` → Initialize sums
- Loops through `numbers` list
- `if num % 2 == 0:` → Adds even numbers to `sum_even`
- `else:` → Adds odd numbers to `sum_odd`
- Prints both sums

🔷 **Output:**

```
Sum of even numbers: 12
Sum of odd numbers: 16
```

```
## Calculate the sum of even and odd numbers

numbers = [1, 2, 3, 4, 5, 6, 7]
sum_even = 0
sum_odd = 0

for num in numbers:
    if num % 2 == 0:
        sum_even += num
    else:
        sum_odd += num

print("Sum of even numbers: {}".format(sum_even))
print("Sum of odd numbers: {}".format(sum_odd))
```

```
Sum of even numbers: 12
Sum of odd numbers: 16
```

Calculates the sum of even numbers up to 10 using a while loop.

- `num = 0` → Starts from 0
- `while num <= 10:` → Loops until 10
- `if num % 2 == 0:` → Checks even numbers
- `sum_even += num` → Adds even numbers to `sum_even`
- **Issue:** `sum_odd` is incomplete (missing `+= num`)

🔷 **Fix:** Add `sum_odd += num` inside `else:`.

```
## While Loop Condition

num = 0
sum_even = 0
sum_odd = 0

while num <= 10:
    if num % 2 == 0:
        sum_even += num
    else:
        sum_odd
```

Loops from 1 to 6 but stops at 4 using `break`.

- `while num < 7:` → Loops while `num` is less than 7
- `if num == 4:` → Stops loop when `num` is 4
- `print(num)` → Prints numbers before 4
- `num += 1` → Increments `num`

🔷 **Output:**

```
1
2
3
```

Loops from 1 to 6 but stops at 4 using `break`.

- `while num < 7:` → Runs while `num` is less than 7
- `if num == 4:` → Stops loop when `num` is 4
- `print(num)` → Prints numbers before 4
- `num += 1` → Increments `num`

🔷 **Output:**

```
1
2
3
```

```
## Break Statement

num = 1
while num < 7:
    if num == 4:
        break
    print(num)
    num += 1
```

```
1
2
3
```

```
## Continue Statement

num = 0
while num < 7:
    num += 1
    if num == 4:
        continue
    print(num)
```

```
1
2
3
5
6
```

```
7
```

Here's the text rewritten with different wording while keeping the same meaning:

Tutorial 4 - Python Operators

Types of Python Operators:

- Logical
- Equality
- Comparison
- Arithmetic

4.1.1 Logical Operators

In Python, the following keywords are used for Boolean operations:

Keyword	Description
`not`	Unary negation
`and`	Conditional AND
`or`	Conditional OR

Examples:

Let me know if you need more modifications! 😊

type(False) → Returns the data type of False.

🔹 Output:

javascript Copy Edit <class 'bool'> Explanation: False is a Boolean (bool) type in Python.

```
type(False)
```

```
bool
```

`bool(1)` → **Converts `1` to a boolean value.**

🔹 **Output:**

```
True
```

Explanation: In Python, any nonzero number is considered `True`.

```
bool(1)
```

```
True
```

`bool(1)` → **Converts `1` to a boolean value.**

🔹 **Output:**

```
True
```

Explanation: In Python, any nonzero number is considered `True`.

```
a=True
b=False
```

`True and False` → **Evaluates a logical AND operation.**

🔹 **Output:**

```
False
```

Explanation: `and` returns `True` only if both values are `True`; otherwise, it returns `False`.

```
True and False
```

```
False
```

```
True or False
```

```
True
```

```
not False
```

```
True
```

```
age=int(input("Enter the age"))
if age<18 or age>=35:
    print("Successful execution")
```

```
Enter the age19
```

Equality Operators

Python provides the following operators to check for equality:

Operator	Description
`is`	Returns `True` if variables `a` and `b` refer to the same object.
`is not`	Returns `True` if variables `a` and `b` refer to different objects.
`==`	Returns `True` if `a` and `b` have the same value.
`!=`	Returns `True` if `a` and `b` have different values.

Let me know if you need further modifications! 😊

```
x = "Krish"
y = "Krish1"

x == y
```

```
False
```

Checks if the user's age is 18 and prints a message twice.

int(input("Enter your age")) → Takes age as an integer if years == 18: → Checks if age is 18 Issue: The same if condition is repeated, so the message prints twice. 🔷 Fix: Remove the duplicate if statement.

🔷 Example Output:

sql Copy Edit Enter your age: 18
You are in your teenage years
You are in your teenage years

```
years = int(input("Enter your age"))

if years == 18:
    print("You are in your teenage years")
if years == 18:
    print("You are in your teenage years")
```

```
Enter your age18
 You are in your teenage years
 You are in your teenage years
```

Prints the memory addresses of two string variables.

- `id(name1)` → Returns the memory location of `"PRASHANT"`
- `id(name2)` → Returns the memory location of `"SHIVAM"`

🔹 **Example Output (addresses will vary):**

```
140123456789456
140123456789512
```

Each string has a unique memory address.

```
name1 = "PRASHANT"
name2 = "SHIVAM"

print(id(name1))
print(id(name2))
```

```
138798761550576
138798761655088
```

`a is b` → **Checks if `a` and `b` refer to the same memory location.**

🔹 **Returns:**

- `True` → If both variables point to the same object
- `False` → If they are different objects

✅ **Example:**

```
a = [1, 2, 3]
b = a
print(a is b)  # True (same object)
```

`a is b` → **Checks if `a` and `b` refer to the same memory location.**

🔹 **Returns:**

- `True` → If both variables point to the same object
- `False` → If they are different objects

✅ **Example:**

```
a = [1, 2, 3]
b = a
print(a is b)  # True (same object)
```

```
a is b
```

```
False
```

Prints memory addresses of two lists.

- `list_a = [1, 2, 3]` → Creates a list
- `list_b = [1, 2, 3]` → Creates another list with the same values
- `id(list_a) != id(list_b)` → Different memory locations since lists are mutable

🔹 **Example Output (addresses will vary):**

```
140123456789456
140123456789512
```

Even though `list_a` and `list_b` have the same values, they are different objects in memory.

```
list_a = [1, 2, 3]
list_b = [1, 2, 3]

print(id(list_a))
print(id(list_b))
```

```
139630616976320
139630616978880
```

`list_a is list_b` → **Checks if both lists refer to the same memory location.**

- `list_a = [1, 2, 3]` → Creates a new list
- `list_b = [1, 2, 3]` → Creates another new list with the same values
- `list_a is list_b` → **Returns** `False` because lists are mutable and stored separately in memory

✅ **Example Output:**

```
False
```

```
list_a = [1, 2, 3]  # Assign a list to list_a
list_b = [1, 2, 3]  # Assign a list to list_b

list_a is list_b  # Now this expression will work correctly
```

```
False
```

`lst is not lst1` → **Checks if** `lst` **and** `lst1` **refer to different memory locations.**

- `lst = [1, 2, 3]` → Creates a list
- `lst1 = [4, 5, 6]` → Creates another different list
- `lst is not lst1` → **Returns** `True` because both are separate objects

✅ **Example Output:**

```
True
```

`lst is not lst1` → **Checks if** `lst` **and** `lst1` **are different objects in memory.**

- `lst = [1, 2, 3]` → Creates a list
- `lst1 = [4, 5, 6]` → Creates another list
- `lst is not lst1` → **Returns** `True` because they have different memory locations

🔷 **Output:**

```
True
```

```
lst = [1, 2, 3]  # Assign a list to lst
lst1 = [4, 5, 6] # Assign a different list to lst1

lst is not lst1  # Now this expression will work correctly
```

```
True
```

`"PRASHANT" != "PRASHANT"` → **Compares two identical strings.**

- `!=` → Checks if values are **not equal**
- Since both strings are the same, **returns** `False`

🔷 **Output:**

```
False
```

```
"PRASHANT"!="PRASHANT"
```

```
False
```

Comparison Operators

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

```
score = float(input("Enter the score"))

if score >= 35:
    print("Qualified")
    if 50 <= score <= 70:
        print("First Division")
elif score < 35:
    print("Not Qualified")
```

```
Enter the score10
Not Qualified
```

Mathematical Operators

Operation	Description
+	Adds two values
-	Subtracts one value from another
*	Multiplies two values
/	Performs true division (returns a float)
//	Performs floor division (returns an integer)
%	Returns the remainder of a division operation

`24 + 24` → **Adds two numbers.**

🔹 **Output:**

```
48
```

```
24+24
```

```
48
```

48 - 24 → Subtracts 24 from 48.

🔹 Output:

Copy Edit 24

```
48-24
```

```
24
```

`48 * 24` → **Multiplies 48 by 24.**

🔹 **Output:**

```
1152
```

48 * 24 → Multiplication operation.

🔹 Output:

yaml Copy Edit 1152

```
48*24
```

```
1152
```

```
48/4
```

```
12.0
```

```
48//5
```

```
9
```

```
48%5
```

```
3
```

Tutorial 5- Python Number Methods

```
1+2
```

```
3
```

abs() Function

`abs(x)` returns the absolute value of a given number `x`. The input `x` can be an integer, a floating-point number, or a complex number.

```
abs(10)
```

```
10
```

`abs(-20)` → **Returns the absolute value of `-20`.**

🔹 **Output:**

```
20
```

abs(-20) → Returns the positive value of -20.

🔹 Output:

Copy Edit 20

```
abs(-20)
```

```
20
```

abs(34.20) → Returns the absolute value of 34.20.

🔹 Output:

Copy Edit 34.2

```
abs(34.20)
```

```
34.2
```

`abs(-55.02)` → **Returns the absolute value of `-55.02`.**

🔹 **Output:**

```
55.02
```

```
abs(-55.02)
```

```
55.02
```

`ceil()` Function

`ceil(x)` returns the smallest integer that is greater than or equal to `x`. In other words, it rounds `x` up to the nearest whole number.

Note: This function is not available directly as `ceil()`. It requires importing the `math` module to use it.

```
import math
```

`math.ceil(43.67)` returns the smallest integer greater than or equal to `43.67`, which is `44`.

```
math.ceil(43.3)
```

```
44
```

`math.ceil(42.1)` → **Rounds `42.1` up to the nearest integer.**

🔹 **Output:**

```
43
```

🔹 **Requires:** `import math` before using.

```
math.ceil(42.1)
```

```
43
```

`math.ceil(-44.5)` → **Rounds `-44.5` up to the nearest integer.**

🔹 **Output:**

```
-44
```

🔹 **Requires:** `import math` before using.

```
math.ceil(-44.5)
```

```
-44
```

floor

`math.floor(43.1)` → **Rounds `43.1` down to the nearest integer.**

🔹 **Output:**

```
43
```

🔹 **Requires:** `import math` before using.

```
math.floor(43.1)
```

```
43
```

```
math.floor(43.9)
```

```
43
```

```
math.floor(-56.9)
```

```
-57
```

`exp(x)` calculates the exponential value of `x`, which is `e^x`, where `e` is Euler's number (approximately 2.718).

Note: This function is part of the `math` module and must be accessed using `math.exp(x)`.

```
math.exp(10)
```

```
22026.465794806718
```

`math.exp(-7)` → **Calculates `e⁻⁷` (exponential of `-7`).**

🔹 **Output:**

```
0.0009118819655545162
```

🔹 **Requires:** `import math` before using.

```
math.exp(-7)
```

```
0.0009118819655545162
```

fabs()

math.fabs(10.53) → Returns the absolute value of 10.53.

🔹 Output:

Copy Edit 10.53 🔹 Requires: import math before using.

```
math.fabs(10.53)
```

```
10.53
```

`math.fabs(-10)` → **Returns the absolute value of `-10`.**

🔹 **Output:**

```
10.0
```

🔹 **Requires:** `import math` before using.

```
math.fabs(-10)
```

```
10.0
```

log(x)

`math.log(10)` → **Returns the natural logarithm (ln) of `10`.**

🔹 **Output:**

```
2.302585092994046
```

🔹 **Requires:** `import math` before using.

```
math.log(10)
```

```
2.302585092994046
```

`math.log(65.5)` → **Returns the natural logarithm (ln) of `65.5`.**

🔹 **Output:**

```
4.18180943772277
```

🔷 **Requires:** `import math` before using.

```
math.log(65.5)
```

```
4.182050142641207
```

`math.log10(40)` → **Returns the base-10 logarithm of** `40`.

🔷 **Output:**

```
1.6020599913279625
```

🔷 **Requires:** `import math` before using.

`math.log10(40)` → **Computes log base 10 of** `40`.

🔷 **Output:**

```
1.6020599913279625
```

🔷 **Requires:** `import math` before using.

```
math.log10(40)
```

```
1.6020599913279625
```

Max()

`max(10,12,5,76,100)` → **Returns the largest number.**

🔷 **Output:**

```
100
```

`max(10,12,5,76,100)` → **Finds the maximum value.**

🔷 **Output:**

```
100
```

```
max(10,12,5,76,100)
```

```
100
```

`max(-55, -44, -33)` → **Finds the highest value among negatives.**

🔷 **Output:**

```
-33
```

`max(-55, -44, -33)` → **Returns the largest number.**

🔷 **Output:**

```
-33
```

```
max(-55,-44,-33)
```

```
-33
```

min()

`min(0,100,4,5,6,3)` → **Finds the smallest number.**

🔷 **Output:**

```
0
```

```
min(0,100,4,5,6,3)
```

```
0
```

```
min(-1,0)
```

```
-1
```

pow()

```
import math
math.pow(20,5)
```

```
3200000.0
```

```
import math
math.pow(-5,-5)
```

```
-0.00032
```

sqrt()

`math.sqrt(16)` → **Returns the square root of** `16.`

🔷 **Output:**

```
4.0
```

🔷 **Requires:** `import math` before using.

```
math.sqrt(16)
```

```
4.0
```

`math.sqrt(9)` → **Returns the square root of** `9.`

🔷 **Output:**

```
3.0
```

🔷 **Requires:** `import math` before using.

```
math.sqrt(9)
```

```
3.0
```

`math.sqrt(101)` → **Returns the square root of** `101.`

🔷 **Output:**

```
10.04987562112089
```

🔷 **Requires:** `import math` before using.

```
math.sqrt(101)
```

```
10.04987562112089
```

Triggnometric functions

`math.sin(0)` → **Returns the sine of `0` radians.**

🔹 **Output:**

```
0.0
```

🔹 **Requires:** `import math` before using.

```
import math
math.sin(0)
```

```
0.0
```

```
math.cos(90)
```

```
-0.4480736161291701
```

```
math.cos(0)
```

```
1.0
```

```
math.tan(45)
```

```
1.6197751905438615
```

```
math.tan(90)
```

```
-1.995200412208242
```

```
## hypot()
```

```
math.hypot(2,3)
```

```
3.605551275463989
```

`math.modf(3.14159)` → **Splits a number into fractional and integer parts.**

- **Fractional part:** `0.14159`
- **Integer part:** `3.0`

🔹 **Output:**

```
Fractional part: 0.14159
Integer part: 3.0
```

🔹 **Requires:** `import math` before using.

```
import math

# Call modf with a floating-point number as an argument.
fractional_part, integer_part = math.modf(3.14159)

# Print the results.
print("Fractional part:", fractional_part)
print("Integer part:", integer_part)
```

```
Fractional part: 0.14158999999999988
Integer part: 3.0
```

6-Python List,Dictionary,Sets etc.ipynb

Python Data Structures and Boolean

Boolean

- **Boolean and Logical Operators**
- **Lists**
- **Comparison Operators**
- **Dictionaries**
- **Tuples**
- **Sets**

Boolean Variables

Boolean values consist of two constant objects: **False** and **True**.

- They are used to indicate truth values (other values can also evaluate as true or false).
- In numeric contexts, they behave like integers **0** and **1**, respectively.
- The built-in function **bool()** can be used to convert any value to a Boolean, based on its truthy or falsy nature.
- Boolean values are represented as **False** and **True** in Python.

```
False
```

```
False
```

`print(True, False)` → **Displays boolean values `True` and `False`.**

🔹 **Output:**

```
True False
```

```
print(True,False)
```

```
True False
```

`type(True)` → **Returns the data type of `True`.**

🔹 **Output:**

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

`type(True)` → **Checks the data type of `True`.**

🔹 **Output:**

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

🔹 **Another way to say it:**
Returns that `True` belongs to the Boolean (`bool`) type.

```
type(True)
```

```
bool
```

`type(False)` → **Determines the data type of `False`.**

🔹 **Output:**

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

🔹 **Another way to say it:**
Indicates that `False` is of Boolean (`bool`) type.

```
type(False)
```

```
bool
```

`my_str = 'PRASHANT123'` → **Stores** `"PRASHANT123"` **in** `my_str.`

```
my_str='PRASHANT123'
```

```
my_str.istitle()
```

```
True
```

```
print(my_str.isalnum()) #check if all char are numbers
print(my_str.isalpha()) #check if all char in the string are alphabetic
print(my_str.isdigit()) #test if string contains digits
print(my_str.istitle()) #test if string contains title words
print(my_str.isupper()) #test if string contains upper case
print(my_str.islower()) #test if string contains lower case
print(my_str.isspace()) #test if string contains spaces
print(my_str.endswith('k')) #test if string endswith a d
print(my_str.startswith('K')) #test if string startswith H
```

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

Boolean and Logical Operators

```
True and True
```

```
True
```

```
True and False
```

```
False
```

```
True or False
```

```
True
```

```
True or True
```

```
True
```

```
str_example='Hello World'
my_str='Krish'
```

```
my_str.isalpha() or str_example.isnum()
```

```
True
```

Lists

A **list** is a built-in data structure in Python that allows storing multiple elements in an **ordered** and **modifiable** sequence. Each value inside a list is referred to as an **item**.

- Lists are **mutable**, meaning their elements can be changed after creation.
- They are defined using **square brackets [ ]**, with elements separated by commas.
- Just like strings are enclosed in quotes, lists contain values enclosed within brackets.

```
##mutable vs immutable
str1="Krish"
print(str1)
type(str1)
```

```
Krish
str
```

`str1 = "PRASHANT"` → **Stores** `"PRASHANT"` **in** `str1`.
`print(str1)` → **Displays** `"PRASHANT"`.

🔷 **Output:**

```
PRASHANT
```

```
str1="PRASHANT"
print(str1)
```

```
PRASHANT
```

Modifies a string by creating a new one.

- `str1 = "GUPTA"` → Stores `"GUPTA"`
- `str1[:2] + "jk" + str1[3:]` → Replaces the third character with `"jk"`
- `print(str1)` → Displays modified string

🔷 **Output:**

```
GUjkTA
```

Creates a modified string.

- `str1 = "GUPTA"` → Stores `"GUPTA"`
- `str1[:2] + "jk" + str1[3:]` → Replaces the third character with `"jk"`
- `print(str1)` → Prints updated string

🔷 **Output:**

```
GUjkTA
```

```
str1 = "GUPTA"
# To change the string, you need to create a new string
str1 = str1[:2] + "jk" + str1[3:]  # Replace the third character with "jk"
print(str1)
```

```
GUjkTA
```

Converts a tuple to a list and prints it.

- `list((1,2,3,4,5))` → Converts tuple `(1,2,3,4,5)` into a list
- `type(lst)` → Returns `<class 'list'>` (not printed)
- `print(lst)` → Displays the list

🔷 **Output:**

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

Converts a tuple to a list and prints it.

- `list((1,2,3,4,5))` → Converts tuple to list
- `type(lst)` → Checks type (not printed)
- `print(lst)` → Displays list

🔷 **Output:**

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

```
lst=list((1,2,3,4,5))
type(lst)
print(lst)
```

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

Creates a list with numbers 1 to 5.

🔷 **List:** `[1, 2, 3, 4, 5]`

```
lst=[1,2,3,4,5]
```

This code loops through each element in the list `lst` and prints the square (`i**2`) of each element.

```
for i in lst:
    print(i**2)
```

```
1
4
9
16
25
```

This code returns the smallest element in the list `lst`.

```
min(lst)
```

```
1
```

This code returns the type of an empty list, which is `<class 'list'>`.

```
type([])
```

```
list
```

This creates an empty list named `lst_example`.

```
lst_example=[]
```

```
type(lst_example)
```

```
list
```

The code `lst=list()` creates an empty list in Python, which can be used to store multiple items in an ordered sequence.

```
lst=list()
```

The code `type(lst)` returns the type of the object `lst`, which in this case is `<class 'list'>`, indicating that `lst` is a list.

```
type(lst)
```

```
list
```

The code `lst=['Mathematics', 'chemistry', 100, 200, 300, 204]` creates a list named `lst` containing both string and integer elements.

```
lst=['Mathematics', 'chemistry', 100, 200, 300, 204]
```

The code `len(lst)` returns the number of elements in the list `lst`, which is 6.

```
len(lst)
```

```
6
```

The code `type(lst)` returns the type of `lst`, which is `<class 'list'>`, indicating it's a list.

```
type(lst)
```

```
list
```

Append

The code `lst` simply displays the contents of the list, which is `['Mathematics', 'chemistry', 100, 200, 300, 204]`.

```
lst
```

```
['Mathematics', 'chemistry', 100, 200, 300, 204]
```

The code `lst.append("Krish")` adds the element "Krish" to the end of the list `lst`. After execution, `lst` becomes `['Mathematics', 'chemistry', 100, 200, 300, 204, 'Krish']`.

```
#.append is used to add elements in the list
lst.append("Krish")
lst
```

```
['Mathematics', 'chemistry', 100, 200, 300, 204, 'Krish']
```

The code `lst.append(["John", "Bala"])` adds a sublist `["John", "Bala"]` to the end of `lst`. After execution, `lst` becomes `['Mathematics', 'chemistry', 100, 200, 300, 204, 'Krish', ['John', 'Bala']]`.

```
lst.append(["John","Bala"])
lst
```

```
['Mathematics', 'chemistry', 100, 200, 300, 204, 'Krish', ['John', 'Bala']]
```

The code `lst[2:6]` retrieves a slice of the list `lst` from index 2 to 5 (excluding index 6). It returns `[100, 200, 300, 204]`.

```
lst[2:6]
```

```
[100, 200, 300, 204]
```

The code `lst` displays the current contents of the list, which is `['Mathematics', 'chemistry', 100, 200, 300, 204, 'Krish', ['John', 'Bala']]`.

```
lst
```

```
['Mathematics', 'chemistry', 100, 200, 300, 204, 'Krish', ['John', 'Bala']]
```

The code `lst[6]` retrieves the element at index 6 of the list, which is `'PRASHANT'`.

```
##Indexing in List
lst[6]
```

```
'Krish'
```

```
lst[1:6]
```

```
['chemistry', 100, 200, 300, 204]
```

```
lst[1:6]
```

```
['chemistry', 100, 200, 300, 204]
```

Insert

```
lst
```

```
['Mathematics', 'chemistry', 100, 200, 300, 204, 'Krish', ['John', 'Bala']]
```

```
## insert in a specific order
```

```
lst.insert(2,"Naik")
```

```
lst
```

```
['Mathematics',
 'chemistry',
 'Naik',
 100,
 200,
 300,
 204,
 'Krish',
 ['John', 'Bala']]
```

```
lst.append(["Hello","World"])
```

```
lst
```

```
['Mathematics',
 'chemistry',
 'Naik',
 100,
 200,
 300,
 204,
 'Krish',
 ['John', 'Bala'],
 ['Hello', 'World']]
```

```
lst=[1,2,3]
```

```
lst.append([4,5])
```

```
lst
```

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

Extend Method

```
lst=[1,2,3,4,5,6]
```

```
lst.append([8,9])
lst
```

```
[1, 2, 3, 4, 5, 6, [8, 9]]
```

```
lst.extend([8,9])
```

```
lst
```

```
[1, 2, 3, 4, 5, 6, [8, 9], 8, 9]
```

Various Operations that we can perform in List

```
lst=[1,2,3,4,5]
```

```
sum(lst)
```

```
sum(lst)
```

```
15
```

```
lst*5
```

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

```
for i in lst:
    print(i/5)
```

```
0.2
0.4
0.6
0.8
1.0
```

```
lst*5
```

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

```
lst
```

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

Pop() Method

```
lst.pop()
```

```
5
```

```
lst
```

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

```
lst.pop(2)
```

```
3
```

```
lst
```

```
[1, 2, 4]
```

count():Calculates total occurrence of given element of List

```
lst=[1,1,2,3,4,5]
lst.count(1)
```

```
2
```

```
#length:Calculates total length of List
len(lst)
```

```
6
```

```
# index(): Returns the index of first occurrence. Start and End index are not necessary parameters
lst.index(1,1,4)
```

```
1
```

```
##Min and Max
min(lst)
```

```
1
```

```
max(lst)
```

```
5
```

SETS

A **Set** is an unordered collection of elements that is **iterable, mutable**, and does not allow **duplicate values**. In Python, the **set class** is used to implement this concept, which is derived from the mathematical definition of a set. It is internally based on a **hash table** data structure.

```
## Defining an empy set

set_var= set()
print(set_var)
print(type(set_var))
```

```
set()
<class 'set'>
```

The code `set_var={1,2,3,4,3}` creates a set named `set_var` containing unique elements, so the duplicate `3` is removed. The set becomes `{1, 2, 3, 4}`.

```
set_var={1,2,3,4,3}
```

The code `set_var` displays the contents of the set, which is `{1, 2, 3, 4}`.

```
set_var
```

```
{1, 2, 3, 4}
```

The code creates a set `set_var` with elements `{"Avengers", "IronMan", 'Hitman'}` and prints it. `type(set_var)` returns `<class 'set'>`, indicating it's a set.

```
set_var={"Avengers","IronMan",'Hitman'}
print(set_var)
type(set_var)
```

```
{'IronMan', 'Avengers', 'Hitman'}
set
```

The code `set_var.add("Hulk")` adds the element "Hulk" to the set `set_var`.

```
## Inbuilt function in sets

set_var.add("Hulk")
```

The code `print(set_var)` displays the current contents of the set, which is `{"Avengers", "IronMan", 'Hitman', "Hulk"}`.

```
print(set_var)
```

```
{'IronMan', 'Avengers', 'Hulk', 'Hitman'}
```

The code creates two sets: `set1` with `{"Avengers", "IronMan", 'Hitman'}` and `set2` with `{"Avengers", "IronMan", 'Hitman', 'Hulk2'}`.

```
set1={"Avengers","IronMan",'Hitman'}
set2={"Avengers","IronMan",'Hitman','Hulk2'}
```

The code `set2.intersection_update(set1)` updates `set2` to only contain elements common with `set1`. After execution, `set2` becomes `{"Avengers", "IronMan", 'Hitman'}`.

```
set2.intersection_update(set1)
```

```
set2
```

```
{'Avengers', 'Hitman', 'IronMan'}
```

The code `set2.difference(set1)` returns the elements in `set2` that are not in `set1`. It returns `{'Hulk2'}`.

```
##Difference
set2.difference(set1)
```

```
set()
```

The code `set2` displays the current contents of `set2`, which is `{'Hulk2'}`.

```
set2
```

```
{'Avengers', 'Hitman', 'IronMan'}
```

The code `set2.difference_update(set1)` removes elements from `set2` that are present in `set1`. After execution, `set2` becomes `{'Hulk2'}`.

```
## Difference update
```

```
set2.difference_update(set1)
```

The code `print(set2)` displays the current contents of `set2`, which is `{'Hulk2'}`.

```
print(set2)
```

```
set()
```

Dictionaries

A **dictionary** is a data structure that stores key-value pairs in an **unordered, mutable, and indexed** format. In Python, dictionaries are defined using **curly braces** `{}`, where each element consists of a **key** mapped to a **value**.

```
dic={}
```

The code `type(dic)` returns the type of `dic`, which is `<class 'dict'>`, indicating it's a dictionary.

```
type(dic)
```

```
dict
```

The code `type(dict())` returns `<class 'dict'>`, indicating that `dict()` creates an empty dictionary.

```
type(dict())
```

```
dict
```

The code `set_ex={1,2,3,4,5}` creates a set named `set_ex` with the elements `{1, 2, 3, 4, 5}`.

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

The code `type(set_ex)` returns `<class 'set'>`, indicating that `set_ex` is a set.

```
type(set_ex)
```

```
set
```

The code creates a dictionary `my_dict` with keys ("Car1", "Car2", "Car3") and their corresponding values ("Audi", "BMW", "Mercedes Benz").

```
## Let create a dictionary
```

```
my_dict={"Car1": "Audi", "Car2":"BMW","Car3":"Mercidies Benz"}
```

The code `type(my_dict)` returns `<class 'dict'>`, indicating that `my_dict` is a dictionary.

```
type(my_dict)
```

```
dict
```

The code `my_dict['Car1']` accesses the value associated with the key `'Car1'`, which is `"Audi"`.

```
##Access the item values based on keys

my_dict['Car1']
```

```
'Audi'
```

The code loops through the keys of `my_dict` and prints each key. The output will be:

```
Car1
Car2
Car3
```

```
# We can even loop throught the dictionaries keys

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

```
Car1
Car2
Car3
```

The code loops through the values of `my_dict` and prints each value. The output will be:

```
Audi
BMW
Mercedes Benz
```

The code loops through the values of `my_dict` and prints each value, which are `"Audi"`, `"BMW"`, and `"Mercedes Benz"`.

```
# We can even loop throught the dictionaries values

for x in my_dict.values():
    print(x)
```

```
Audi
BMW
Mercidies Benz
```

The code loops through both keys and values of `my_dict` and prints each key-value pair. The output will be:

```
('Car1', 'Audi')
('Car2', 'BMW')
('Car3', 'Mercedes Benz')
```

The code prints key-value pairs from `my_dict`.

```
# We can also check both keys and values
for x in my_dict.items():
    print(x)
```

```
('Car1', 'Audi')
('Car2', 'BMW')
('Car3', 'Mercidies Benz')
```

The code adds a new key-value pair `'car4': 'Audi 2.0'` to `my_dict`.

```
## Adding items in Dictionaries

my_dict['car4']='Audi 2.0'
```

The code `my_dict` displays the updated dictionary.

```
my_dict
```

```
{'Car1': 'Audi', 'Car2': 'BMW', 'Car3': 'Mercidies Benz', 'car4': 'Audi 2.0'}
```

The code updates the value of `'Car1'` in `my_dict` to `'Maruti'`.

```
my_dict['Car1']='MAruti'
```

The code `my_dict` displays the updated dictionary, which is `{'Car1': 'Maruti', 'Car2': 'BMW', 'Car3': 'Mercedes Benz', 'car4': 'Audi 2.0'}`.

```
my_dict
```

```
{'Car1': 'MAruti', 'Car2': 'BMW', 'Car3': 'Mercidies Benz', 'car4': 'Audi 2.0'}
```

Nested Dictionary

The code creates three dictionaries (`car1_model`, `car2_model`, `car3_model`) and then nests them into a larger dictionary `car_type` with keys `'car1'`, `'car2'`, and `'car3'`.

```
car1_model={'Mercedes':1960}
car2_model={'Audi':1970}
car3_model={'Ambassador':1980}

car_type={'car1':car1_model,'car2':car2_model,'car3':car3_model}
```

The code `print(car_type)` displays the nested dictionary:

```
 {'car1': {'Mercedes': 1960}, 'car2': {'Audi': 1970}, 'car3': {'Ambassador': 1980}}


print(car_type)
```

```
{'car1': {'Mercedes': 1960}, 'car2': {'Audi': 1970}, 'car3': {'Ambassador': 1980}}
```

The code `print(car_type['car1'])` accesses and displays the value associated with the key `'car1'`, which is `{'Mercedes': 1960}`.

```
## Accessing the items in the dictionary

print(car_type['car1'])
```

```
{'Mercedes': 1960}
```

The code `print(car_type['car1']['Mercedes'])` accesses the year `1960` from the nested dictionary `car_type['car1']['Mercedes']`.

```
print(car_type['car1']['Mercedes'])
```

```
1960
```

Tuples

The code creates an empty tuple named `my_tuple`.

```
## create an empty Tuples

my_tuple=tuple()
```

The code `type(my_tuple)` returns `<class 'tuple'>`, indicating `my_tuple` is a tuple.

```
type(my_tuple)
```

The code creates an empty tuple named `my_tuple`.

```
my_tuple=()
```

The code `type(my_tuple)` returns `<class 'tuple'>`, indicating `my_tuple` is a tuple.

```
type(my_tuple)
```

```
tuple
```

The code creates a tuple `my_tuple` with the elements `"PRASHANT"`, `"Ankur"`, and `"John"`.

```
my_tuple=("PRASHANT","Ankur","John")
```

The code creates a tuple `my_tuple` with the elements `'Hello'` and `'World'`.

```
my_tuple=('Hello','World')
```

The code prints the type of `my_tuple` (which is `<class 'tuple'>`) and the contents of `my_tuple` (`('Hello', 'World')`).

```
print(type(my_tuple))
print(my_tuple)
```

```
<class 'tuple'>
('Hello', 'World')
```

The code `type(my_tuple)` returns `<class 'tuple'>`, indicating `my_tuple` is a tuple.

```
type(my_tuple)
```

```
tuple
```

The code `my_tuple.count('PRASHANT')` counts how many times `'PRASHANT'` appears in `my_tuple`.

```
## Inbuilt function
my_tuple.count('PRASHANT')
```

```
0
```

The code creates a tuple `my_tuple` and prints the index of `'Ankur'`, which is `2`.

```
my_tuple = ('Hello', 'World', 'Ankur')  # Add 'Ankur' to the tuple
print(my_tuple.index('Ankur'))  # Now this will print the index of 'Ankur' (which is 2)
```

```
2
```

Tutorial 8 - Python NumPy Arrays

NumPy Overview

NumPy is a powerful library for array-based computing in Python. It offers a **high-performance multidimensional array** structure and various tools to manipulate and perform operations on these arrays. It serves as the **core library for scientific computing** in Python.

Understanding Arrays

An **array** is a structured data type used to store multiple values of the **same data type** in an organized manner. Unlike **Python lists**, which can hold elements of different data types, **NumPy arrays** are designed to store **only homogeneous data** (elements of the same type).

```
## import the library
import numpy as np
```

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

```
type(arr)
```

```
numpy.ndarray
```

```
arr.shape
```

```
(4,)
```

```
lst1=[1,2,3,4,5]
lst2=[2,3,4,5,6]
lst3=[3,4,5,6,7]

arr1=np.array([lst1,lst2,lst3])
```

```
arr1
```

```
array([[1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])
```

```
arr1.shape
```

```
(3, 5)
```

```
#indexing
arr[3]
```

```
4
```

```
arr[3]=5
```

```
arr
```

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

```
arr[-1]
```

```
5
```

```
arr[:-1]
```

```
array([1, 2, 3])
```

```
arr[::-3]
```

```
array([5, 1])
```

```
arr1
```

```
array([[1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])
```

```
arr1[:,3:].shape
```

```
(3, 2)
```

```
arr1[:,1]
```

```
array([2, 3, 4])
```

```
arr1[1:,1:3]
```

```
array([[3, 4],
       [4, 5]])
```

```
arr1[1:,3:]
```

```
array([[5, 6],
       [6, 7]])
```

```
##EDA
arr
```

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

```
arr[arr<2]
```

```
array([1])
```

```
arr1
```

```
array([[1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7]])
```

```
arr1.reshape(5,3)
```

```
array([[1, 2, 3],
       [4, 5, 2],
       [3, 4, 5],
       [6, 3, 4],
       [5, 6, 7]])
```

```
##mechanism to create an array
np.arange(1,20,2).reshape(2,5)
```

```
array([[ 1,  3,  5,  7,  9],
       [11, 13, 15, 17, 19]])
```

```
np.arange(1,20,2).reshape(2,5,1)
```

```
array([[[ 1],
        [ 3],
        [ 5],
        [ 7],
        [ 9]],

       [[11],
        [13],
        [15],
        [17],
        [19]]])
```

```
arr *arr
```

```
array([ 1,  4,  9, 25])
```

```
arr1 * arr1
```

```
array([[ 1,  4,  9, 16, 25],
       [ 4,  9, 16, 25, 36],
       [ 9, 16, 25, 36, 49]])
```

```
np.ones((5,3))
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

```
np.zeros((4,5))
```

```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

```
np.random.randint(10,50,4).reshape(2,2)
```

```
array([[14, 44],
       [45, 30]])
```

```
np.random.randn(5,6)
```

```
array([[ 1.364159  ,  0.64853629, -1.79003519, -0.4831596 , -1.07067143,
        -0.46606484],
       [-0.68413157, -0.61619057, -0.63602444, -0.51150599, -0.23464932,
        -0.5086206 ],
       [ 0.36276398,  0.12099929,  0.76586084, -0.96676113,  0.97641236,
         0.84720967],
       [-1.26299974, -0.31111896,  0.09873799, -0.87457118, -1.69271566,
         0.85191542],
       [ 1.146852  , -0.20709935,  0.57034504, -0.18403732, -1.56637059,
         0.28745848]])
```

```
np.random.random_sample((4,7))
```

```
array([[0.78613926, 0.64108037, 0.44236694, 0.98923293, 0.95385383,
        0.30022414, 0.80727272],
       [0.76966352, 0.97379596, 0.29432639, 0.06805791, 0.74021576,
        0.49085209, 0.02950275],
       [0.13947663, 0.30506695, 0.05597039, 0.72881369, 0.20157191,
        0.44112961, 0.32856359],
       [0.69845964, 0.06657119, 0.91062996, 0.15267499, 0.7252863 ,
        0.6363192 , 0.41150356]])
```

Tutorial 9 - Python Pandas Guide (Part 1)

In this section, we will explore the fundamentals of **Pandas**, a powerful data analysis library in Python. The topics covered include:

- **Pandas DataFrame** – A tabular data structure similar to an Excel spreadsheet or SQL table.
- **Pandas Series** – A one-dimensional labeled array capable of holding any data type.
- **Basic Pandas Operations** – Key functions to manipulate and analyze data efficiently.

```
!pip install pandas
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.11/dist-packages (2.2.2)
Requirement already satisfied: numpy>=1.23.2 in /usr/local/lib/python3.11/dist-packages (from pandas) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas) (2025.1)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas) (2025.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2->pandas) (1.17.0)
```

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

The code creates a 5x4 NumPy array with values from 0 to 19 using `np.arange(0,20)` and reshapes it into a 5x4 matrix.

The code creates a 5x4 matrix with values from 0 to 19 using `np.arange(0,20)` and reshapes it.

```
np.arange(0,20).reshape(5,4)
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
```

```
           [ 8,  9, 10, 11],
           [12, 13, 14, 15],
           [16, 17, 18, 19]])
```

The code creates a DataFrame `df` with values from 0 to 19, reshaped into a 5x4 matrix, and labeled with custom row and column names.

```
## Create Dataframe
df=pd.DataFrame(data=np.arange(0,20).reshape(5,4),index=["Row1",
                                                        "Row2","Row3",
                                                        "Row4","Row5"],columns=["Column1",
                                                                                "Column2",
                                                                                "Column3",
                                                                                "Column4"])
```

The code `df.head()` displays the first 5 rows of the DataFrame `df`.

```
df.head()
```

	Column1	Column2	Column3	Column4
Row1	0	1	2	3
Row2	4	5	6	7
Row3	8	9	10	11
Row4	12	13	14	15
Row5	16	17	18	19

The code `df.tail()` displays the last 5 rows of the DataFrame `df`.

```
df.tail()
```

	Column1	Column2	Column3	Column4
Row1	0	1	2	3
Row2	4	5	6	7
Row3	8	9	10	11
Row4	12	13	14	15
Row5	16	17	18	19

```
type(df)
```

```
pandas.core.frame.DataFrame
def __init__(data=None, index: Axes | None=None, columns: Axes | None=None, dtype: Dtype |
None=None, copy: bool | None=None) -> None
```

```
Two-dimensional, size-mutable, potentially heterogeneous tabular data.

Data structure also contains labeled axes (rows and columns).
Arithmetic operations align on both row and column labels. Can be
thought of as a dict-like container for Series objects. The primary
pandas data structure.
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 5 entries, Row1 to Row5
Data columns (total 4 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   Column1  5 non-null      int64
 1   Column2  5 non-null      int64
 2   Column3  5 non-null      int64
 3   Column4  5 non-null      int64
dtypes: int64(4)
memory usage: 372.0+ bytes
```

The code `df.describe()` provides a summary of statistical measures (like mean, min, max, etc.) for numerical columns in the DataFrame `df`.

```
df.describe()
```

	Column1	Column2	Column3	Column4
count	5.000000	5.000000	5.000000	5.000000
mean	8.000000	9.000000	10.000000	11.000000
std	6.324555	6.324555	6.324555	6.324555
min	0.000000	1.000000	2.000000	3.000000
25%	4.000000	5.000000	6.000000	7.000000
50%	8.000000	9.000000	10.000000	11.000000
75%	12.000000	13.000000	14.000000	15.000000
max	16.000000	17.000000	18.000000	19.000000

The code `df.head()` shows the first 5 rows of the DataFrame `df`. It refers to indexing in the context of columns and rows.

```
##Indexing
## columnname,rowindex[loc],rowindex columnindex number[.iloc]
df.head()
```

	Column1	Column2	Column3	Column4
Row1	0	1	2	3
Row2	4	5	6	7
Row3	8	9	10	11
Row4	12	13	14	15
Row5	16	17	18	19

The code `type(df['Column1'])` returns the data type of the values in the `Column1` of the DataFrame `df`.

```
##columnname
type(df['Column1'])
```

```
pandas.core.series.Series
def __init__(data=None, index=None, dtype: Dtype | None=None, name=None, copy: bool | None=None,
fastpath: bool | lib.NoDefault=lib.no_default) -> None
```

```
One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be a hashable type. The object
supports both integer- and label-based indexing and provides a host of
methods for performing operations involving the index. Statistical
methods from ndarray have been overridden to automatically exclude
```

```
df[['Column1','Column2','Column3']]
```

	Column1	Column2	Column3
Row1	0	1	2
Row2	4	5	6
Row3	8	9	10
Row4	12	13	14
Row5	16	17	18

The code `df.loc[['Row3', 'Row4']]` selects the rows labeled 'Row3' and 'Row4' from the DataFrame `df`.

```
##using row index name loc
df.loc[['Row3','Row4']]
```

	Column1	Column2	Column3	Column4
Row3	8	9	10	11
Row4	12	13	14	15

The code `df.head()` displays the first 5 rows of the DataFrame `df`.

```
df.head()
```

	Column1	Column2	Column3	Column4
Row1	0	1	2	3
Row2	4	5	6	7
Row3	8	9	10	11
Row4	12	13	14	15
Row5	16	17	18	19

The code `df.iloc[2:4, 0:2]` selects rows 2 to 3 and columns 0 to 1 from the DataFrame `df`.

```
df.iloc[2:4,0:2]
```

	Column1	Column2
Row3	8	9
Row4	12	13

```
df.iloc[2:,1:]
```

	Column2	Column3	Column4
Row3	9	10	11
Row4	13	14	15
Row5	17	18	19

The code `df.iloc[:, 1:].values` converts all rows and columns from index 1 onwards of the DataFrame `df` into a NumPy array.

```
##convert dataframe into arrays
df.iloc[:,1:].values
```

```
array([[ 1,  2,  3],
       [ 5,  6,  7],
       [ 9, 10, 11],
       [13, 14, 15],
       [17, 18, 19]])
```

The code `df.isnull().sum()` counts the number of missing (null) values in each column of the DataFrame `df`.

```
## Basic operations
df.isnull().sum()
```

	0
Column1	0
Column2	0
Column3	0
Column4	0

dtype: int64

Creates a DataFrame with missing (NaN) values.

```
df=pd.DataFrame(data=[[1,np.nan,2],[1,3,4]],index=["Row1",
                                                   "Row2"],columns=["Column1",
                                                                    "Column2",
                                                                    "Column3",
                                                                    ])
```

Displays the DataFrame.

```
df
```

	Column1	Column2	Column3
Row1	1	NaN	2
Row2	1	3.0	4

Counts missing values in each column.

```
df.isnull().sum()
```

	0
Column1	0
Column2	1
Column3	0

dtype: int64

Checks if each column has zero missing values.

```
df.isnull().sum()==0
```

	0
Column1	True
Column2	False
Column3	True

dtype: bool

Displays the DataFrame.

```
df
```

	Column1	Column2	Column3
Row1	1	NaN	2
Row2	1	3.0	4

Counts unique values in **Column3** of the DataFrame.

```
df['Column3'].value_counts()
```

Column3	count
2	1
4	1

dtype: int64

Displays the DataFrame.

```
df
```

	Column1	Column2	Column3
Row1	1	NaN	2
Row2	1	3.0	4

Returns unique values from "Column2" in the DataFrame.

```
df['Column2'].unique()
```

```
array([nan,  3.])
```

Filters and returns rows where values in "Column2" are greater than 2.

```
df[df['Column2']>2]
```

	Column1	Column2	Column3
Row2	1	3.0	4

Tutorial 14- Python Pickling And Unpickling The pickle module implements binary protocols for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as “serialization”, “marshalling,” 1 or “flattening”; however, to avoid confusion, the terms used here are “pickling” and “unpickling”.

Pickle in Python is primarily used in serializing and deserializing a Python object structure. In other words, it's the process of converting a Python object into a byte stream to store it in a file/database, maintain program state across sessions, or transport data over the network

Imports the Seaborn library for data visualization.

```
import seaborn as sns
```

Loads the "tips" dataset from Seaborn into the DataFrame `df`.

```
df=sns.load_dataset('tips')
```

Displays the first 5 rows of `df`.

```
df.head()
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Imports the `pickle` module for serialization.

```
import pickle
```

Assigns the string `'file.pkl'` to the variable `filename`.

```
filename='file.pkl'
```

Saves the DataFrame `df` to `'file.pkl'` in binary format using pickle.

```
##serialize process
pickle.dump(df,open(filename,'wb'))

##unsereliaze
df=pickle.load(open(filename,'rb'))
```

Displays the first 5 rows of the DataFrame.

```
df.head()
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Creates a dictionary with first and last name.

Creates a dictionary with keys `'first_name'` and `'last_name'`.

```
dic_example={'first_name':'PRASHNAT','last_name':'GUPTA'}

pickle.dump(dic_example,open('test.pkl','wb'))
```

Loads data from the `test.pkl` file in binary mode.

```
pickle.load(open('test.pkl','rb'))
```

```
{'first_name': 'Krish', 'last_name': 'Naik'}
```

Python List Comprehension

```
type([1,2,3,4,5,5])
```

```
list
```

Creates a list of numbers from 0 to 7.

```
lst=[]
for i in range(0,8):
    lst.append(i)

print(lst)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

Squares each number and stores them in a list.

```
numbers = [1, 2, 3, 4, 5, 6]
squared_numbers = []

for num in numbers:
    squared_numbers.append(num ** 2)

print(squared_numbers)
```

```
[1, 4, 9, 16, 25, 36]
```

Squares each number in the list.

```
values = [1, 2, 3, 4, 5]
squared_values = [num ** 2 for num in values]
print(squared_values)
```

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

Extracts even numbers from a list.

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_values = [num for num in values if num % 2 == 0]
print(even_values)
```

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

Converts a nested list into a single list.

```
nested_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened_list = [element for group in nested_lists for element in group]
print(flattened_list)
```

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

Flattens a nested list into a single list.

```
nested_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flattened_list = [element for group in nested_lists for element in group]
print(flattened_list) # Changed the variable from flattend_list to flattened_list
```

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

Converts a list of strings into a list of integers.

```
## Generating a list of the squares of even numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[n**2 for n in numbers if n%2==0]
```

```
[4, 16, 36, 64, 100]
```

```
## Converting a list of strings to a list of integers
strings = ['1', '2', '3', '4', '5']
[int(s) for s in strings]
```

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

Generates Fibonacci numbers by summing the two previous values in the `fib` list.

```
## Generating a list of the Fibonacci sequence using a list comprehension
fib = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]

[fib[i-1]+ fib[i-2] for i in range(2,len(fib))]
```

```
[1,
 2,
 3,
 5,
 8,
 13,
 21,
 34,
 55,
 89,
 144,
 233,
 377,
 610,
 987,
 1597,
 2584,
 4181,
 6765]
```

```
##Generating a list of all the divisors of a number:

number =36
[i for i in range(1,number+1) if number%i==0]
```

```
[1, 2, 3, 4, 6, 9, 12, 18, 36]
```

Anonymous Functions in Python

In Python, an anonymous function, also known as a **lambda function**, is a concise function that can be defined in a single line without a formal name. It is particularly useful when we need a short function without explicitly using the `def` keyword.

Syntax of a Lambda Function:

```
lambda parameters: expression
```

- **Parameters** represent the inputs to the function.
- **Expression** is a single operation that gets evaluated and returned as output.
- Unlike regular functions, a lambda function **automatically returns** the result of the expression without requiring an explicit `return` statement.

```
f=lambda x,y:x+y
f
```

```
<function __main__.<lambda>(x, y)>
```

`f(5,6)` calls function `f` with arguments `5` and `6`. The result depends on how `f` is defined.

```
f(5,6)
```

```
11
```

Returns the length of `"PRASHANT GUPTA"`, which is `14`.

```
get_length = lambda text: len(text)
get_length("PRASHANT GUPTA")
```

```
13
```

Squares each number in `nums` and prints `[1, 4, 9, 16, 25, 36]`.

```
nums = [1, 2, 3, 4, 5, 6]
squared_values = list(map(lambda n: n**2, nums))
print(squared_values)
```

```
[1, 4, 9, 16, 25, 36]
```

Filters even numbers from `nums` and prints `[2, 4, 6]`.

```
nums = [1, 2, 3, 4, 5, 6]
even_nums = list(filter(lambda n: n % 2 == 0, nums))
print(even_nums)
```

```
[2, 4, 6]
```

The code defines a lambda function `f` that checks if a number is even. `f(3)` returns `False` because 3 is not even.

```
f=lambda x:x%2==0
f(3)
```

```
False
```

The code sorts the `fruits` list first by the length of each fruit and then alphabetically for fruits with the same length. The result is `['date', 'apple', 'banana', 'cherry', 'elderberry']`.

```
fruits = ['apple', 'banana', 'cherry', 'date', 'elderberry']
sorted_fruits = sorted(fruits, key=lambda fruit: (len(fruit), fruit))
print(sorted_fruits)
```

```
['date', 'apple', 'banana', 'cherry', 'elderberry']
```

The code `sorted(fruits, key=lambda x: len(x))` sorts the list `fruits` based on the length of each element in ascending order.

```
sorted(fruits,key=lambda x:len(x))
```

```
['date', 'apple', 'banana', 'cherry', 'elderberry']
```

Advanced Examples

Arranging a List of Dictionaries by a Particular Key

This defines a list of dictionaries, each representing an individual with `full_name`, `years`, and `profession` keys.

```
individuals = [
    {'full_name': 'Alice', 'years': 25, 'profession': 'Engineer'},
    {'full_name': 'Bob', 'years': 30, 'profession': 'Manager'},
    {'full_name': 'Charlie', 'years': 22, 'profession': 'Intern'},
    {'full_name': 'Dave', 'years': 27, 'profession': 'Designer'},
]
```

This sorts the `individuals` list based on the `years` value.

```
sorted(individuals, key=lambda x: x['years'])
```

```
[{'full_name': 'Charlie', 'years': 22, 'profession': 'Intern'},
 {'full_name': 'Alice', 'years': 25, 'profession': 'Engineer'},
 {'full_name': 'Dave', 'years': 27, 'profession': 'Designer'},
 {'full_name': 'Bob', 'years': 30, 'profession': 'Manager'}]
```

This finds the key in the data dictionary with the highest value.

```
## Finding the maximum value in a dictionary
data = {'a': 10, 'b': 20, 'c': 5, 'd': 15}
max(data,key=lambda x:data[x])
```

```
'b'
```

```
## Grouping a list of strings based on their first letter
```

This groups the `items` list by the first letter of each word and prints the groups.

```
from itertools import groupby

items = ['apple', 'banana', 'cherry', 'date', 'elderberry', 'fig']

categorized = groupby(sorted(items), key=lambda word: word[0])

for letter, category in categorized:
    print(letter, list(category))
```

```
a ['apple']
b ['banana']
c ['cherry']
d ['date']
e ['elderberry']
f ['fig']
```

Data Hiding in Python

Data hiding is a core concept in object-oriented programming that involves grouping related data and functions into a unified structure called a class. This technique helps regulate access to data and functions, ensuring security and efficient code organization.

```
class Individual:
    ## Constructor
    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def show_details(self):
        print(f"The individual's name is {self.__name} and their age is {self.__age}")


individual = Individual("Krish", 32)

individual.show_details()
```

```
The individual's name is Krish and their age is 32
```

```
### Access Control ---> Encapsulation
## Protected

class Individual:
    ## Constructor
    def __init__(self, name, age):
        self._name = name
        self._age = age
```

This creates an instance of the `Individual` class with the name "Krish" and age 32.

```
individual=Individual("Krish",32) # Changed 'Person' to 'Individual'
```

This defines a class `Individual` with a constructor to set `name` and `age`. The `Learner` class inherits from `Individual`, calls the superclass constructor, and adds a method to display the details.

```
class Individual:
    ## Constructor
    def __init__(self, name, age):
        self._name = name
        self._age = age
```

Got it! Here are some essential **DSA codes in Python** with explanations.

---

1 Reverse an Array

```
def reverse_array(arr):
    return arr[::-1]

arr = [1, 2, 3, 4, 5]
print(reverse_array(arr))  # Output: [5, 4, 3, 2, 1]
```

---

2 Check if a String is a Palindrome

```
def is_palindrome(s):
    return s == s[::-1]

print(is_palindrome("radar"))  # Output: True
print(is_palindrome("hello"))  # Output: False
```

---

3 Find the Maximum Element in an Array

```
def find_max(arr):
    return max(arr)

arr = [10, 20, 30, 5, 15]
print(find_max(arr))  # Output: 30
```

---

4 Binary Search (Efficient Searching in Sorted Array)

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

arr = [1, 3, 5, 7, 9, 11]
print(binary_search(arr, 7))  # Output: 3
```

---

5 Bubble Sort (Sorting Algorithm)

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

arr = [64, 34, 25, 12, 22, 11, 90]
print(bubble_sort(arr))  # Output: [11, 12, 22, 25, 34, 64, 90]
```

---

6 Find Fibonacci Number using Recursion

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(6))  # Output: 8
```

---

7 Find Factorial using Recursion

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)

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

---

8 Graph Traversal - BFS (Breadth-First Search)

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            queue.extend(graph[node])

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
```

```
bfs(graph, 'A')  # Output: A B C D E F
```

Let me know if you need **more DSA codes** or explanations for any of these! □□