ENGINEERING COLLEGE HUB

FOCUS & WIN

"You can be discouraged by failure or you can learn from it. So go ahead and make mistakes. Make all you can because that's where you will find success, on the other side of failure."

Page No.:

Date : / /

Python Programming

Unit -1 One Shot + PYQs.

To read and write simple python programs.

→ Basic Concept of Python.

→ Python Variables

→ Data types

→ Python basic Operator + Code

→ python blocks

→ Declaring and Using Numeric Data types : int, float, etc.

SUBSCribe + JOIN TELEGRAM

Python ✓

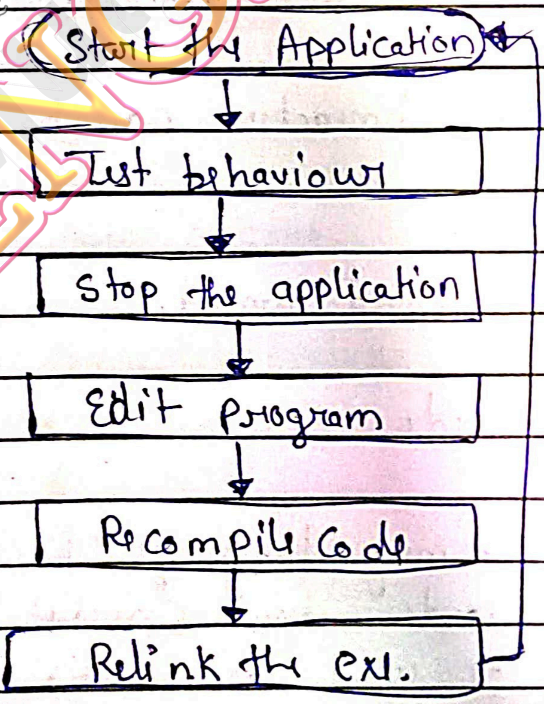
→ Python is a general-purpose interpreted, interactiv object-oriented and high-level programming language.

→ It was created by Guido van Rossum during 1985-90.

Features of Python

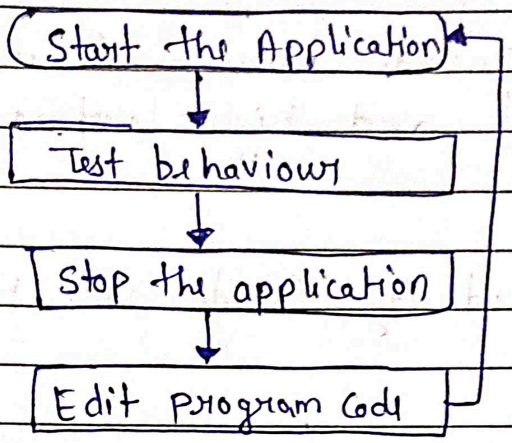
- Easy to learn
- Easy to read
- Easy to maintain
- Scalable
- GUI Programming.

1. Traditional Development Cycle



Page No.:
Date: / /

2. Python's Development Cycle.



Identifier

→ An identifier is a name given to a variable, function, class, module or other entities in a program.

Rules for Naming

→ (a-z, A-Z) or (_) underscore. ← Starting

→ The remaining characters can be letters, number or underscore.

→ case sensitive

→ Cannot be a reserved word.

Reserved Keywords.

→ In a programming language Reserved keywords are that words have special meaning and cannot be used as identifiers

e.g → (False, class, global, with).

Page No..
Date / /

Lines and Indentation ✓

→ Python does not use braces {} to indicates blocks of code for class and function.

→ Blocks of code are denoted by line indentation

e.g →
```
if True :
        print ("True")
else
        print ("False")
```

Comments in Python.

→ Single line comment

''' ''' → multi line Comment.

Python Variables

→ Variables are nothing but reserved memory locations to store values. It means that when you create a variable, you reserve some space in the memory.

→ Based on the data type of a variable, the interpreter allocates memory.

→ you can store integers, decimals or char.

Page No.:
Date : / /

Standard Data Types.

→ python has various standard data types that are used to define the operations possible on them.

→ python has five standard data types –

1) Numbers → (int, float, complex).

2) String → (set of char).

3) List → (collection of elements)

4) Tuple → (similiar to list but immutable).

5) Dictionary → (Key value pairs).

Understanding Python Blocks

→ In python, a block is a group of statements, that are indented together.

→ Blocks are used to define the scope of variables and to control the flow of execution.

e.g →
```
if condition:
    # - - - -
    # - - - -
    |
    |
    |
    |
```

Page No..
Date / /

→ Python basic Operators.

→ Declaring and Using Numeric data types : int , float , etc.
↓ Complex.

Subscribe

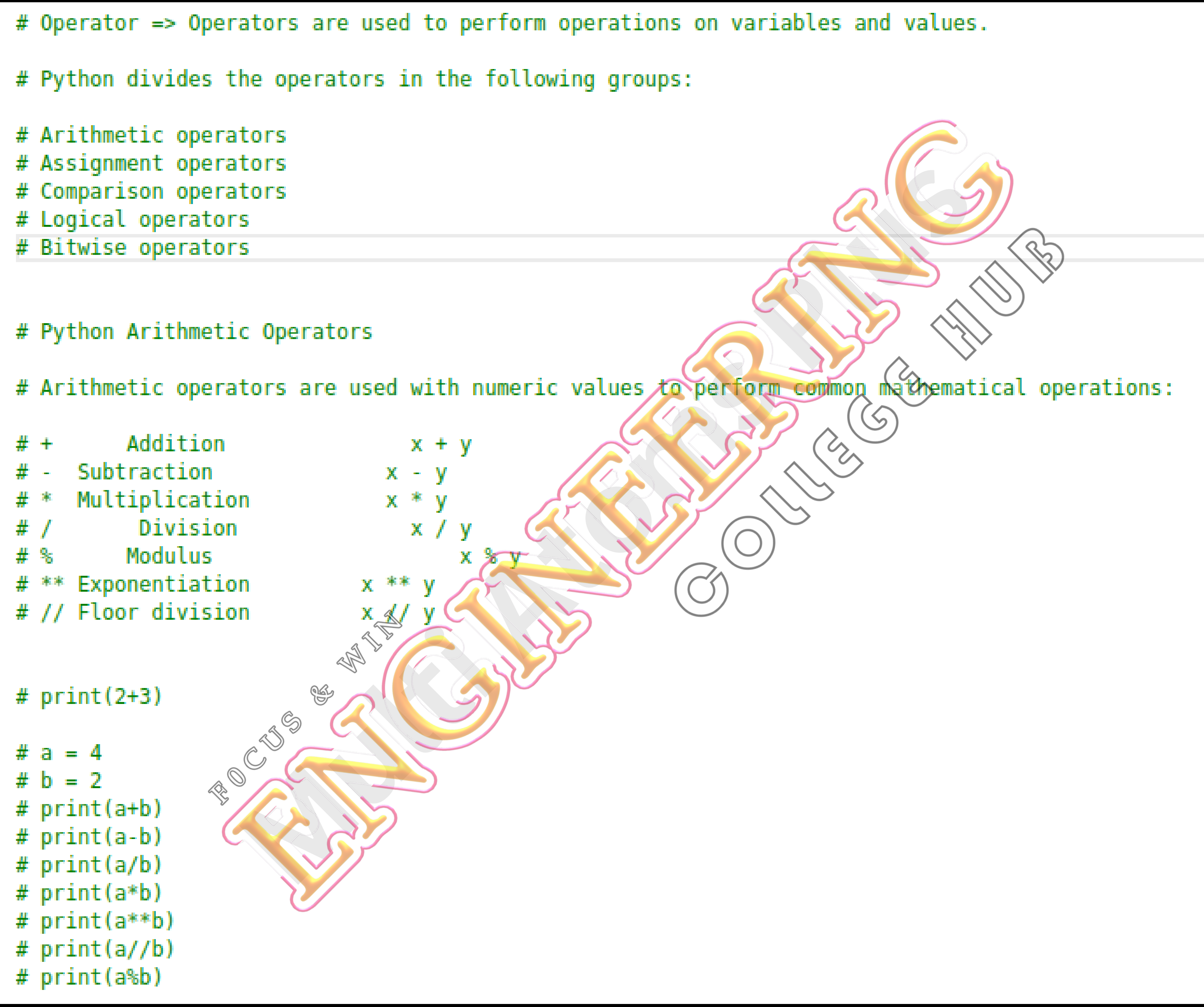
+

Join Telegram

Github

Link: Github Code

```
# Python Numbers

# There are three numeric types in Python:
# 1. int
# 2. float
# 3. complex

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

# x = 1    # int
# y = 2.8  # float
# z = 1j   # complex

# print(type(x))
# print(type(y))
# print(type(z))

# Int
# Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

# Example
# Integers:

# x = 1
# y = 35656222554887711
# z = -3255522

# print(type(x))
# print(type(y))
# print(type(z))

# Float
# Float, or "floating point number" is a number, positive or negative, containing one or more decimals.
# Example
# Floats:

# x = 1.10
# y = 1.0
# z = -35.59

# print(type(x))
# print(type(y))
# print(type(z))
```



```
# Operator => Operators are used to perform operations on variables and values.

# Python divides the operators in the following groups:

# Arithmetic operators
# Assignment operators
# Comparison operators
# Logical operators
# Bitwise operators



# Python Arithmetic Operators

# Arithmetic operators are used with numeric values to perform common mathematical operations:

# +      Addition               x + y
# -  Subtraction              x - y
# *  Multiplication           x * y
# /       Division              x / y
# %      Modulus                    x % y
# ** Exponentiation         x ** y
# // Floor division         x // y


# print(2+3)

# a = 4
# b = 2
# print(a+b)
# print(a-b)
# print(a/b)
# print(a*b)
# print(a**b)
# print(a//b)
# print(a%b)
# Python Assignment Operators

# Assignment operators are used to assign values to variables:

# =                  x = 5                 x = 5
# +=                x += 3                 x = x + 3
# -=                x -= 3                 x = x - 3
# *=                x *= 3                 x = x * 3
# /=                x /= 3                 x = x / 3
# %=                x %= 3                 x = x % 3
# //=                    x //= 3                x = x // 3



# Python Comparison Operators

# Comparison operators are used to compare two values:


# ==            Equal                             x == y
# !=            Not equal                          x != y
# >                 Greater than                       x > y
# <                 Less than                           x < y
# >=            Greater than or equal to          x >= y
# <=            Less than or equal to                 x <= y


# a = 4
# b = 2

# print(a==b)
# print(a!=b)
# print(a>b)
# print(a<b)
# print(a>=b)
# print(a<=b)
# Ques 1: Programs to add two numbers


# a = int(input("Enter first Number "))

# b = int(input("Enter second Number "))

# c = a + b

# print("Sum = ",c)


# Ques 2: Program to calculate area of Circle

radius = int(input("Enter the Radius of Circle--> "))

area = 3.14 * radius * radius

print("Area of Circle --> ",area)
```

```python
# Python Logical Operators

# Logical operators are used to combine conditional statements:



# and          Returns True if both statements are true                          x < 5 and  x < 10
# or       Returns True if one of the statements is true                     x < 5 or x < 4

# not          Reverse the result, returns False if the result is true          not(x < 5 and x < 10)

# a = 2

# print(a<1 and a<10)
# print(not(a<1 or a<10))


# Python Bitwise Operators
# Bitwise operators are used to compare (binary) numbers:


'''&      AND
        Sets each bit to 1 if both bits are 1                                   x & y

   | OR
        Sets each bit to 1 if one of two bits is 1                              x | y

   ^ XOR
       Sets each bit to 1 if only one of two bits is 1                           x ^ y

   ~ NOT
       Inverts all the bits                                                   ~x

   <<     Zero fill left shift
       Shift left by pushing zeros in from the right                           x << 2
       and let the leftmost bits fall off
   >>     Signed right shift
       Shift right by pushing copies of the leftmost
       bit in from the left, and let the rightmost bits
       fall off                                                                x >> 2'''

# 5 -> 1 0 1
# 7 -> 1 1 1

a = 5
b = 7
c = 3

print(a&b)
print(a|b)
print(a^b)
print(~c)
```

Python Programming (BCC302 / BCC402/ BCC302H / BCC402H)

●●●

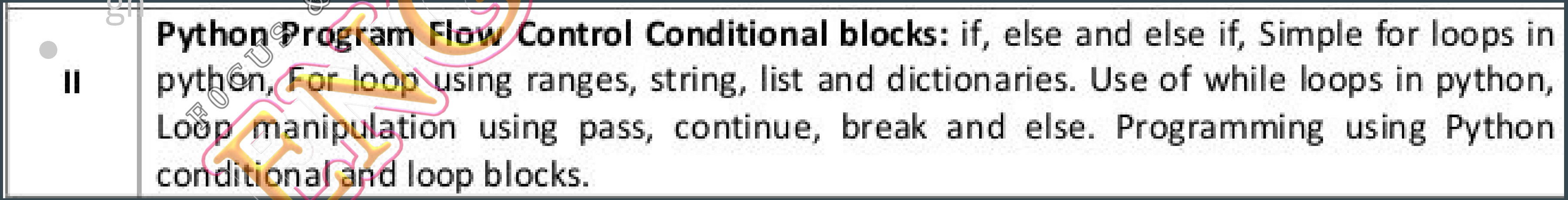
Unit-2 ONE SHOT

Python Program Flow Control Conditional Blocks

PYQS + IMP QUESTIONS

Syllabus

gh



II	**Python Program Flow Control Conditional blocks:** if, else and else if, Simple for loops in python, For loop using ranges, string, list and dictionaries. Use of while loops in python, Loop manipulation using pass, continue, break and else. Programming using Python conditional and loop blocks.

- Conditional Statements - if else
- Loops - For, While
- Pass, Continue and break - (2022-23)
- If-else and Loops Programs - PYQs

Decision Making :

- Decision-making is the anticipation of conditions occurring during the execution of a program and specified actions taken according to the conditions.
- Decision structures evaluate multiple expressions, which produce TRUE or FALSE as the outcome. You need to determine which action to take and which statements to execute if the outcome is TRUE or FALSE otherwise.
- Python programming language assumes any non-zero and non-null values as TRUE, and any zero or null values as FALSE value.

Python programming language provides the following types of decision-making statements.

if statements

An if statement consists of a boolean expression followed by one or

more statements.

if...else statements

An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE.

nested if statements

You can use one if or else if statement inside another if or else if statements..

Let’s Code Decision Making Statements

```python
# 1. if Statement

# x = 1
# if x > 5:
#   print("x is greater than 5")



# 2. if-else Statement

# x=31
# if x > 5:
#   print("x is greater than 5")
# else:
#   print("x is not greater than 5")

# 3. Nested if-Else Statement

# score = 95
# if score >= 90:
#   grade = 'A'
# elif score >= 80:
#   grade = 'B'
# elif score >= 70:
#   grade = 'C'
# else:
#   grade = 'F'
# print(f"Your grade is {grade}")
```

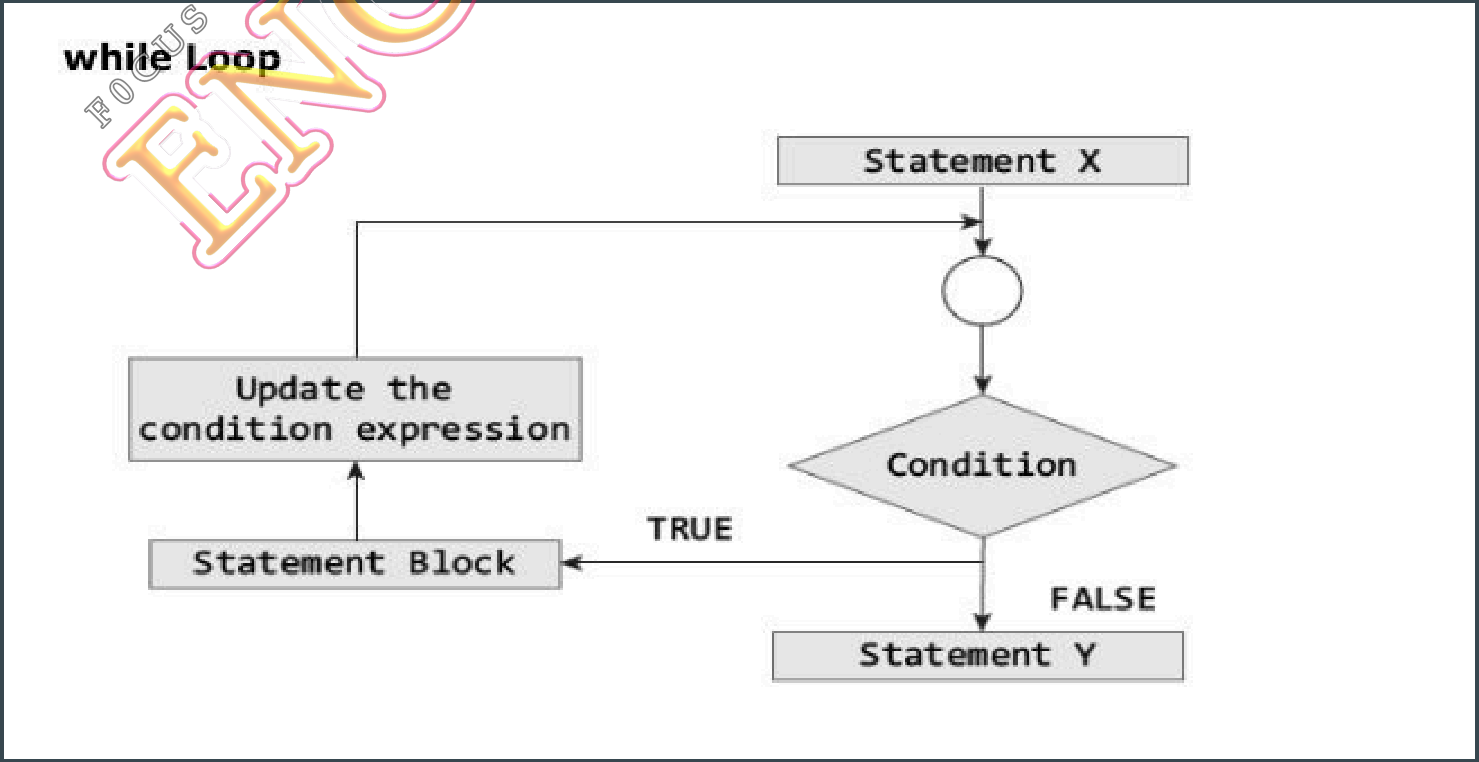
Loops in Python

A loop statement allows us to execute a statement or group of statements multiple times.

types of loops:

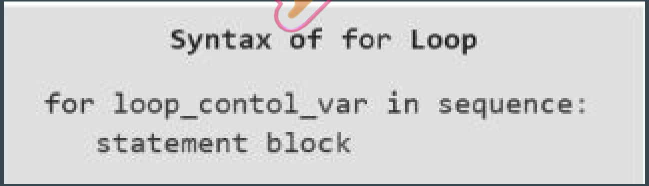
- While Loop
- For Loop
- Nested Loop

With the while loop we can execute a set of statements as long as a condition is true.



For Loop

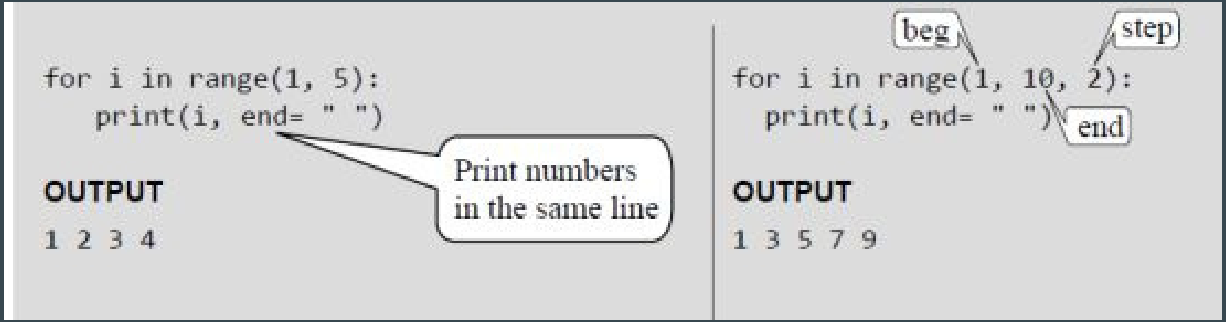
For loop provides a mechanism to repeat a task until a particular condition is True. It is usually known as a determinate or definite loop because the programmer knows exactly how many times the loop will repeat. The for...in statement is a looping statement used in Python to iterate over a sequence of objects.


Syntax of for Loop

```
for loop_contol_var in sequence:
    statement block
```

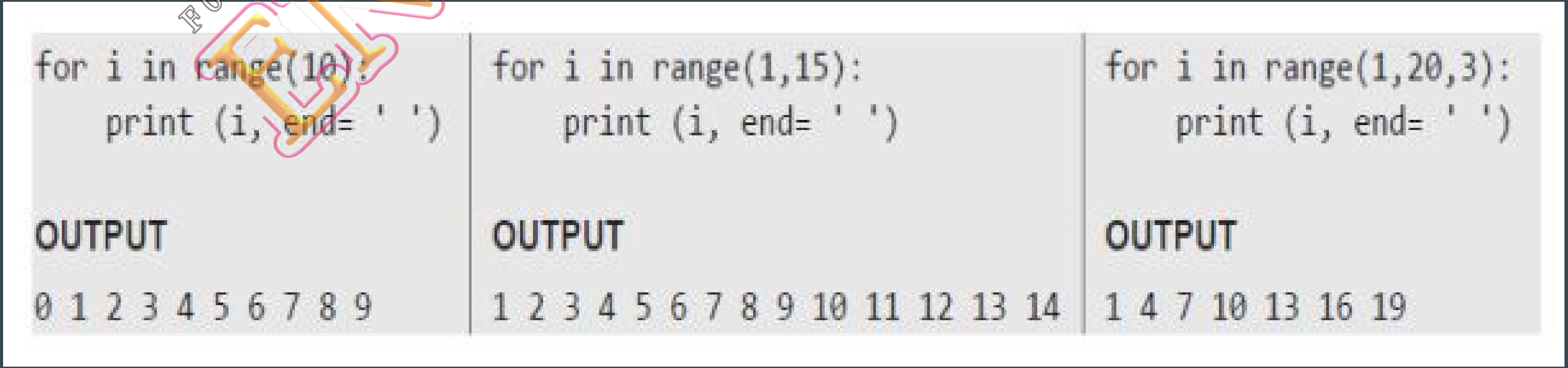
For Loop and Range() Function

The range() function is a built-in function in Python that is used to iterate over a sequence of numbers. The syntax of range() is range(beg, end, [step])
The range() produces a sequence of numbers starting with beg (inclusive) and ending with one less than the number end. The step argument is option (that is why it is placed in brackets). By default, every number in the range is incremented by 1 but we can specify a different increment using step. It can be both negative and positive, but not zero.



```
for i in range(1, 5):
    print(i, end= " ")

OUTPUT
1 2 3 4
```

```
for i in range(1, 10, 2):
  print(i, end= " ")

OUTPUT
1 3 5 7 9
```



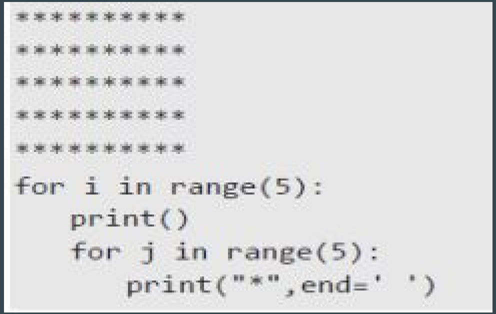
```
for i in range(10):
    print (i, end= ' ')

OUTPUT
0 1 2 3 4 5 6 7 8 9
```

```
for i in range(1,15):
    print (i, end= ' ')

OUTPUT
1 2 3 4 5 6 7 8 9 10 11 12 13 14
```

```
for i in range(1,20,3):
    print (i, end= ' ')

OUTPUT
1 4 7 10 13 16 19
```

Nested Loops

- Python allows its users to have nested loops, that is, loops that can be placed inside other loops. Although this feature will work with any loop like while loop as well as for loop.
- A for loop can be used to control the number of times a particular set of statements will be executed. Another outer loop could be used to control the number of times that a whole loop is repeated.
- Loops should be properly indented to identify which statements are contained within each for statement.



```
**********
**********
**********
**********
**********
for i in range(5):
    print()
    for j in range(5):
        print("*",end=' ')
```

```
# 1. Write a program to sum the digits of a number.

# n=int(input("Enter a number:"))

# tot=0

# while(n>0):
#   dig = n%10
#   tot = tot+dig
#   n = n//10

# print("The total sum of digits is:",tot)



# Write a program to calculate factorial of a given number.

# n=int(input("Enter number:"))
# fact=1
# while(n>0):
#   fact=fact*n
#   n=n-1
# print("Factorial of the number is: ")
# print(fact)


# Write a program to print Fibonacci series.

# a=0
# b=1
# n=int(input("Enter the number of terms needed "))
# print(a,b,end=" ")
# while(n-2):
#   c=a+b
#   a=b
#   b=c
#   print(c,end=" ")
#   n=n-1
```

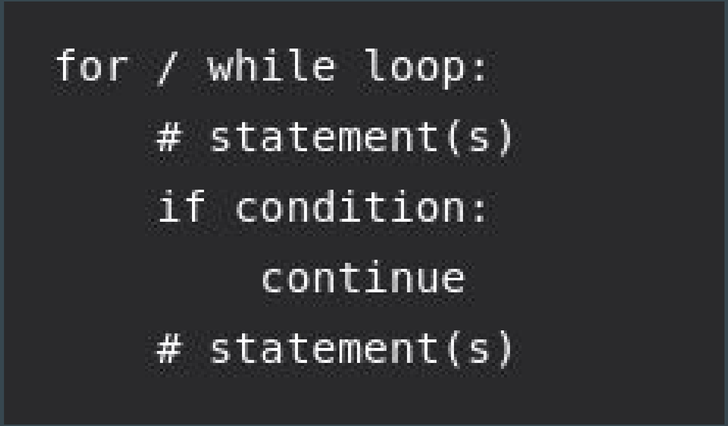
Break Statement in Python

The break statement in Python is used to terminate the loop or statement in which it is present.

```
for / while loop:
    # statement(s)
    if condition:
        break
    # statement(s)
# loop end
```

Continue Statement in Python

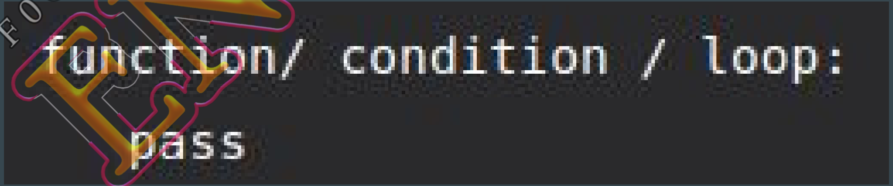
Continue is also a loop control statement just like the break statement. continue statement is opposite to that of the break statement, instead of terminating the loop, it forces to execute the next iteration of the loop. As the name suggests the continue statement forces the loop to continue or execute the next iteration.



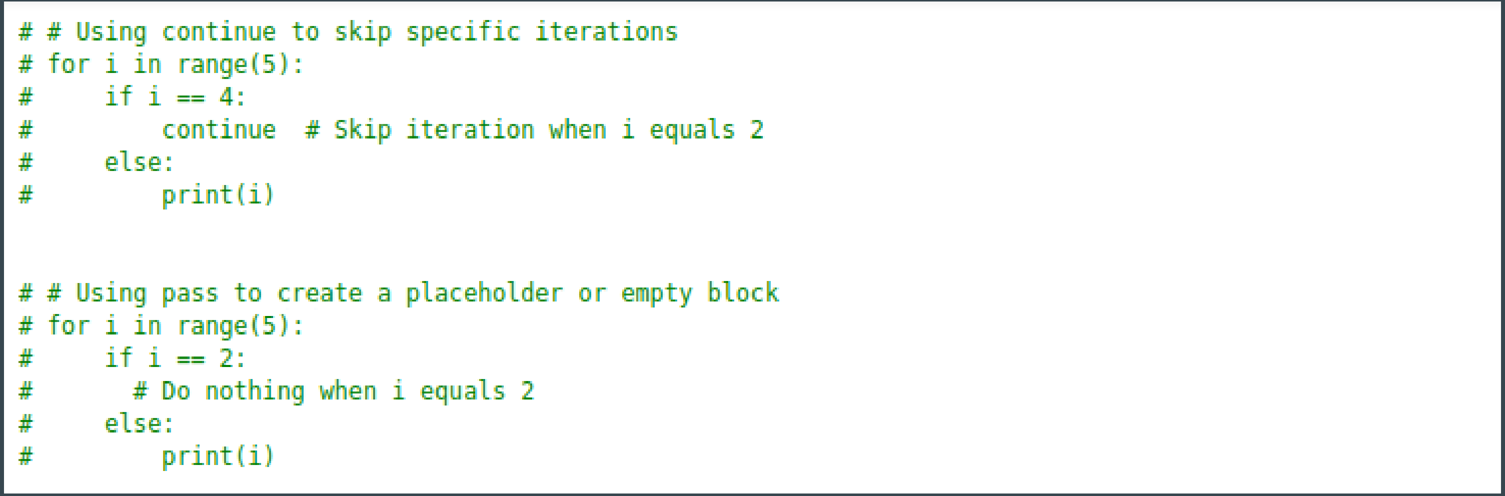
```
for / while loop:
    # statement(s)
    if condition:
        continue
    # statement(s)
```

Pass Statement in Python

As the name suggests pass statement simply does nothing. The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. It is like a null operation, as nothing will happen if it is executed.



```
function/ condition / loop:
    pass
```



```
# # Using continue to skip specific iterations
# for i in range(5):
#     if i == 4:
#         continue  # Skip iteration when i equals 2
#     else:
#         print(i)


# # Using pass to create a placeholder or empty block
# for i in range(5):
#     if i == 2:
#        # Do nothing when i equals 2
#     else:
#         print(i)
```

Thank You so much for watching

Subscribe For More Videos

Join Telegram Channel for Notes

Python Programming (BCC302 / BCC402/ BCC302H / BCC402H)

Python Complex data types

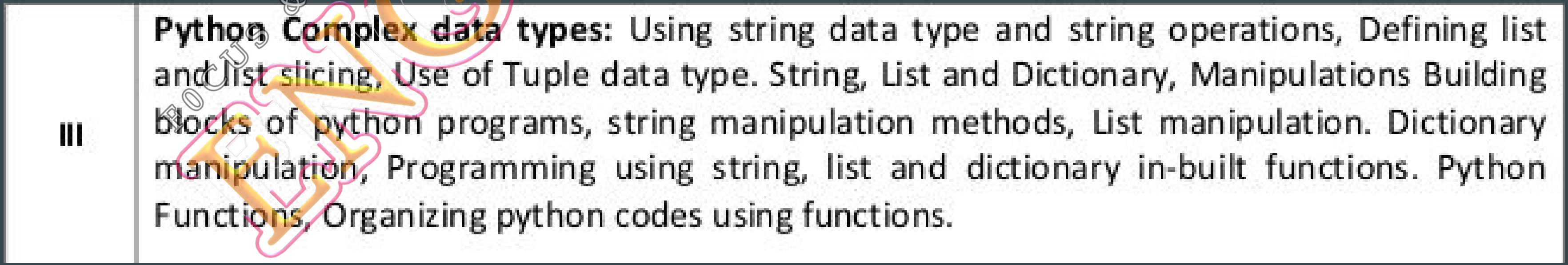
●●●

Unit-3 ONE SHOT

PYQS + IMP QUESTIONS

Multi Atoms Plus

Syllabus



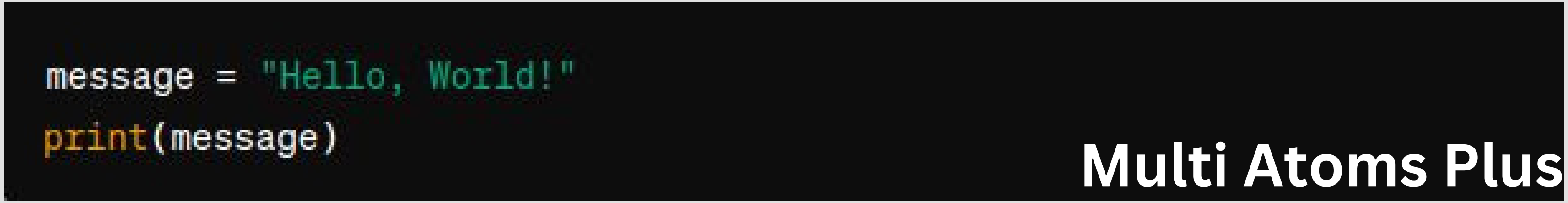
III	**Python Complex data types:** Using string data type and string operations, Defining list and list slicing, Use of Tuple data type. String, List and Dictionary, Manipulations Building blocks of python programs, string manipulation methods, List manipulation. Dictionary manipulation, Programming using string, list and dictionary in-built functions. Python Functions, Organizing python codes using functions.

- String and its Operations - 2021-22
- List and its Operations - 2022-23
- Tuple and Dictionary - 2021-22(2)
- Functions - 2022-23

Multi Atoms Plus

Introduction to Python Complex Data Types:

- Python offers various data types to store and manipulate data.
- Complex data types include strings, lists, tuples, and dictionaries.
- These data types serve different purposes and have unique characteristics.



```
message = "Hello, World!"
print(message)
```

Python String:

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows either pair of single or double quotes.

Operations:

1. Concatenation: Concatenation involves combining two or more strings into a single string.
2. Indexing: Indexing allows accessing individual characters within a string using their positions (indices).

- Python uses zero-based indexing, where the first character is at index 0.

3. Length: The ‘len()’ function returns the length of a string, i.e., the number of characters it contains.

4. Case Conversion: String methods like upper(), lower(), capitalize(), and title() can change the case of characters in a string.

5. String Splitting and Joining:

- The split() method splits a string into a list of substrings based on a delimiter.
- The join() method concatenates elements of a list into a single string with a specified

Lets Code

```
# 1.Concatenation

# greeting = "Hello"
# name = "Alice"
# message = greeting + ", " + name + "!"
# print(message)  # Output: Hello, Alice!

# 2.  Indexing:

# message = "Hello"
# print(message[0])  # Output: H
# print(message[4])  # Output: o


# 3. Length:

# message = "Hello, World!"
# print(len(message))  # Output: 13

# 4. Case Conversion:

# message = "Hello, World!"
# print(message.upper())  # Output: HELLO, WORLD!
# print(message.lower())  # Output: hello, world!
# print(message.capitalize())  # Output: Hello, world!
# print(message.title())  # Output: Hello, World!


# 5. String Splitting and Joining:

# sentence = "Hello, how are you?"
# words = sentence.split()  # Split by whitespace
# print(words)  # Output: ['Hello,', 'how', 'are', 'you?']

# words = ["Hello", "how", "are", "you?"]
# sentence = " ".join(words)  # Join with a whitespace separator
# print(sentence)  # Output: Hello how are you?
```

String Slicing

- string slicing is a technique used to extract a portion of a string by specifying a range of indices.
- Syntax : string[start:stop:step]

1. Basic
2. Omitting Start or End
3. Negative Indices
4. Step Parameter
5. Reversing a String

```
# # 1. Basic

# message = "Hello, World!"
# print(message[2:6])  # Output: "llo,"


# # 2. Omitting Start or End

# # If you omit the start parameter, the slice starts from the beginning of the string.
# # If you omit the end parameter, the slice extends to the end of the string.

# message = "Hello, World!"
# print(message[:5])  # Output: "Hello"
# print(message[7:])  # Output: "World!"


# 3. Negative Indices
# # Negative indices count from the end of the string.
# # Slicing with negative indices allows you to extract substrings from the end of the string.

# message = "Hello, World!"
# print(message[-6:-1])  # Output: "World"


# 4. Step Parameter
# # The step parameter specifies how many characters to skip between each character included in the slice.

# message = "Hello, World!"
# print(message[::2])  # Output: "HloWrd"


# 5. Reversing a String

# message = "Hello, World!"
# print(message[::-1])  # Output: "!dlroW ,olleH"
```

Lists in Python - 2022-23

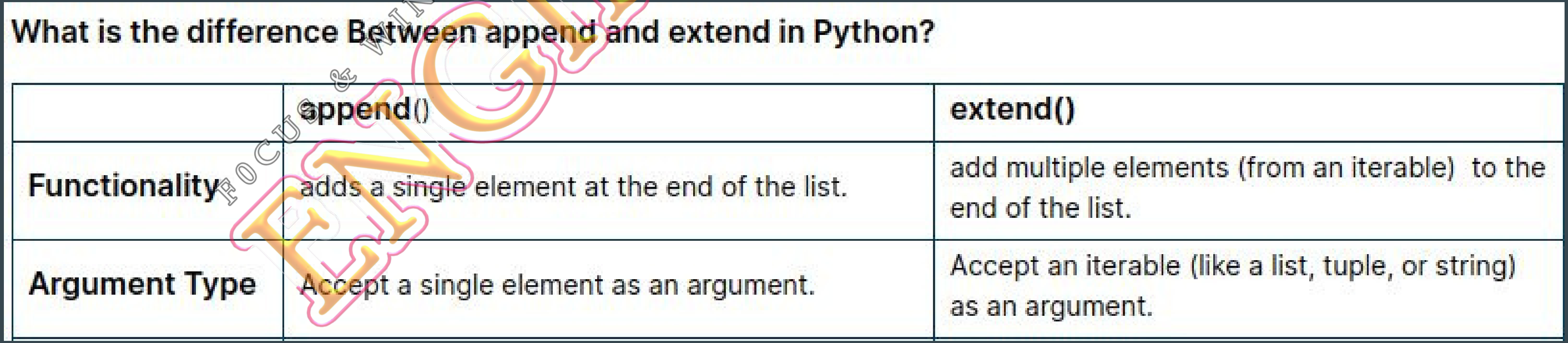
- Lists are ordered collections of items in Python.
- They are versatile and can contain elements of different data types, including integers, floats, strings, and even other lists.

```python
numbers = [1, 2, 3, 4, 5]
fruits = ['apple', 'banana', 'orange']
mixed_list = [1, 'apple', 3.14, True]
```

1. Appending Elements
2. Removing Elements
3. Slicing Lists
4. Concatenating Lists
5. Reversing a String

```
# a. Appending Elements:
# You can add elements to the end of a list using the append() method.

# fruits = ['apple', 'banana', 'orange']
# fruits.append('grape')
# print(fruits)  # Output: ['apple', 'banana', 'orange', 'grape']


# b. Removing Elements:
# Elements can be removed from a list using methods like remove() or pop().


# fruits = ['apple', 'banana', 'orange']
# fruits.remove('banana')
# print(fruits)  # Output: ['apple', 'orange']


# c. Slicing Lists:
# Slicing allows you to extract a portion of a list.


# numbers = [1, 2, 3, 4, 5]
# subset = numbers[1:4]
# print(subset)  # Output: [2, 3, 4]


# d. Concatenating Lists:
# Lists can be concatenated using the + operator.

# list1 = [1, 2, 3]
# list2 = [4, 5, 6]
# combined_list = list1 + list2
# print(combined_list)  # Output: [1, 2, 3, 4, 5, 6]

# -> Lists are mutable, meaning their elements can be changed after creation.
# -> They offer flexibility and are widely used for storing and manipulating collections of data in Python.
```

2022-23- 2marks



What is the difference Between append and extend in Python?

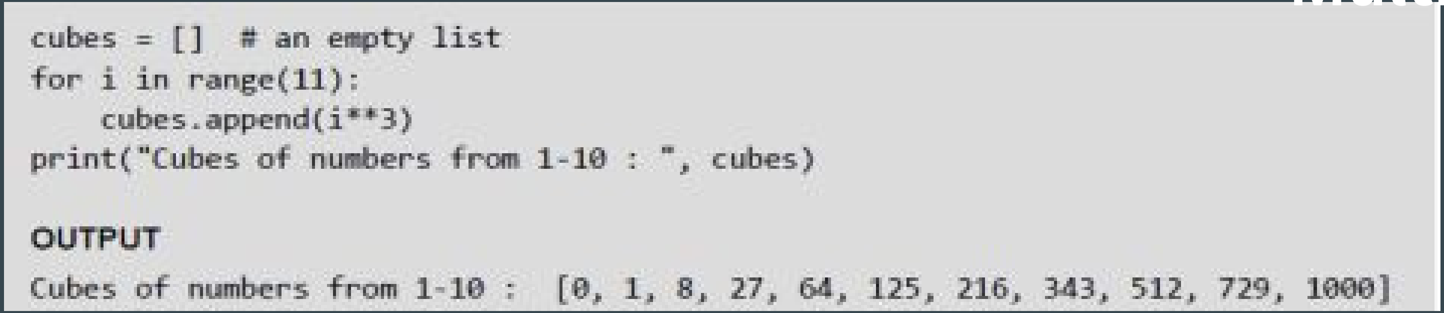
	append()	extend()
Functionality	adds a single element at the end of the list.	add multiple elements (from an iterable) to the end of the list.
Argument Type	Accept a single element as an argument.	Accept an iterable (like a list, tuple, or string) as an argument.

Lists Comprehension - 2022-23

Python also supports computed lists called list comprehensions having the following syntax. List = [expression for variable in sequence]

Where, the expression is evaluated once, for every item in the sequence.

List comprehensions help programmers to create lists in a concise way. This is mainly beneficial to make new lists where each element is the obtained by applying some operations to each member of another sequence or iterable. List comprehension is also used to create a subsequence of those elements that satisfy a certain condition.



```
cubes = []  # an empty list
for i in range(11):
    cubes.append(i**3)
print("Cubes of numbers from 1-10 : ", cubes)

OUTPUT
Cubes of numbers from 1-10 :  [0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Tuples in Python

- Tuples are ordered collections of elements in Python.
- They are similar to lists but are immutable, meaning their elements cannot be changed after creation.

```
coordinates = (10, 20)
fruits = ('apple', 'banana', 'orange')
mixed_tuple = (1, 'apple', 3.14, True)
```

1. Immutable Nature
2. Tuple Unpacking
3. Length and Membership Test

```
# a. Immutable Nature:
# Once a tuple is created, its elements cannot be modified, added, or removed.

# coordinates = (10, 20)
# coordinates[0] = 5  # This will raise a TypeError

# b. Tuple Unpacking:

# Tuple unpacking allows you to assign the elements of a tuple to separate variables.

# coordinates = (10, 20)
# x, y = coordinates
# print(x, y)  # Output: 10 20

# c. Length and Membership Test:

# The len() function returns the length of a tuple, i.e., the number of elements it contains.
# The in and not in operators are used to test for membership in a tuple.

# fruits = ('apple', 'banana', 'orange')
# print(len(fruits))  # Output: 3
# print('banana' in fruits)  # Output: True

# Key Points:

# Tuples provide a lightweight data structure for storing collections of items.
# They are commonly used for representing fixed collections of data, such as coordinates, database records, or function return values.
# While tuples are immutable, they can contain mutable objects such as lists.

# Ques. Write a Python Program to add an item in a tuple (2021-22)

# tuplex = (2,3,4,5);
# # Convert the tuple to a list.
# listx = list(tuplex)
# # Use different methods to add items to the list.
# listx.append(30)
# # Convert the modified list back to a tuple to obtain 'tuplex' with the added element.
# tuplex = tuple(listx)
# # Print the final 'tuplex' tuple with the added element
# print(tuplex)
```

Dictionaries in Python - 2022-23

- Dictionaries are unordered collections of key-value pairs in Python.
- They provide a flexible way to store and retrieve data, where each value is associated with a unique key.

```
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

1. Adding Items
2. Removing Items
3. Dictionary Methods - keys() , values() , items().

```
# a. Adding Items:

# New key-value pairs can be added to a dictionary using assignment.

# person = {"name": "Alice", "age": 30}

# person["city"] = "New York"
# print(person)

# b. Removing Items:

# Items can be removed from a dictionary using the del keyword or the pop() method.

# person = {"name": "Alice", "age": 30, "city": "New York"}
# # del person["age"]
# # print(person)
# person.pop("city")
# print(person)

# Dictionary Methods:

# a. keys():

# The keys() method returns a view of all keys in the dictionary.

# person = {"name": "Alice", "age": 30, "city": "New York"}
# print(person.keys())  # Output: dict_keys(['name', 'age', 'city'])

# b. values():

# The values() method returns a view of all values in the dictionary.

# person = {"name": "Alice", "age": 30, "city": "New York"}
# print(person.values())  # Output: dict_values(['Alice', 30, 'New York'])

# c. items():
```

```
# Dictionary Methods:

# a. keys():

# The keys() method returns a view of all keys in the dictionary.

# person = {"name": "Alice", "age": 30, "city": "New York"}
# print(person.keys())  # Output: dict_keys(['name', 'age', 'city'])

# b. values():

# The values() method returns a view of all values in the dictionary.

# person = {"name": "Alice", "age": 30, "city": "New York"}
# print(person.values())  # Output: dict_values(['Alice', 30, 'New York'])

# c. items():

# The items() method returns a view of all key-value pairs in the dictionary as tuples.

# person = {"name": "Alice", "age": 30, "city": "New York"}
# print(person.items())  # Output: dict_items([('name', 'Alice'), ('age', 30), ('city', 'New York')])


#  Key Points:

# Dictionaries are versatile data structures used to store key-value pairs.
# They offer fast and efficient lookup capabilities, making them ideal for various programming tasks in Python.
```

Functions

- Functions are reusable blocks of code that perform specific tasks.
- They help organize code into manageable chunks, promote code reuse, and enhance readability.

Lets Code

```
# 1. Introduction:

# Functions are reusable blocks of code that perform specific tasks.
# They help organize code into manageable chunks, promote code reuse, and enhance readability.


# 2. Defining Functions:

# Functions are defined using the def keyword followed by the function name and parameters enclosed in parentheses.

# def greet(name):
#     print("Hello, " + name + "!")


# 3. Calling Functions:

# Functions are called by using their name followed by parentheses containing any required arguments.


# greet("Alice")  # Output: Hello, Alice!


# 4. Parameters and Arguments:

# Parameters are placeholders for data that the function expects to receive.
# Arguments are the actual values passed to the function when it is called.


# def add(x, y):
#     return x + y

# result = add(3, 5)
# print(result)  # Output: 8
# 5. Return Statement:

# The return statement allows a function to send data back to the caller.
# Functions can return one or more values.

# def square(x):
#     return x ** 2

# result = square(4)
# print(result)  # Output: 16


# 6. Default Parameters:

# Default parameters have predefined values and are used when no argument is provided.

# def greet(name="Guest"):
#     print("Hello, " + name + "!")

# greet()  # Output: Hello, Guest!
# greet("Alice")  # Output: Hello, Alice!



# 7. Lambda Functions:

# Lambda functions, also known as anonymous functions, are small, single-expression functions.
# They are defined using the lambda keyword and are often used for short, simple operations.


# double = lambda x: x * 2
# print(double(5))  # Output: 10
# 8. Recursion:

# Recursion is a technique in which a function calls itself to solve smaller instances of the same problem.


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

# result = factorial(5)
# print(result)  # Output: 120



# 9. Key Points:

# Functions are essential building blocks of Python programs.
# They encapsulate logic, promote code reuse, and improve code organization.
# Understanding how to define, call, and work with functions is crucial for effective Python programming.
```

Thank You so much for watching

Subscribe For More Videos

Join Telegram Channel for Notes

Multi Atoms Plus

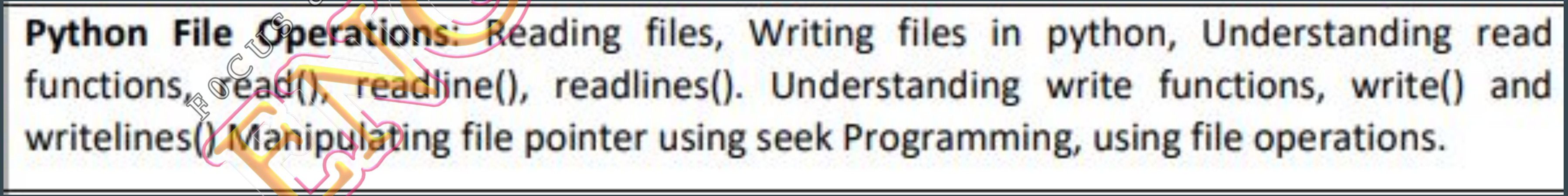
Python Programming

...

Unit-4 One Shot (BCC-302 & BCC-402)

Python File Operations

Unit-4 Syllabus



Python File Operations: Reading files, Writing files in python, Understanding read functions, read(), readline(), readlines(). Understanding write functions, write() and writelines() Manipulating file pointer using seek Programming, using file operations.

Theory + Coding Part + Important Ques

Introduction to Python File Operations

File operations in Python allow you to perform various tasks on files such as reading, writing, appending, and manipulating the file pointer. Python provides built-in functions and methods to work with files efficiently.

File: A file is a digital container used to store data on a computer. It has a name, often with an extension indicating its type (e.g., .txt for text files), and can contain various types of information such as text, images, or programs. Files are essential for data allowing information to be saved and retrieved later.

Types of files:

1. **Text Files** : .txt, .log etc.
2. **Binary Files** : .mp4, .mov, .png etc.

File operations in Python involve several key tasks. You can open a file in various modes such as read ('r'), write ('w'), and append ('a'). Reading operations include reading the entire file content at once, reading line by line, or reading all lines into a list. Writing operations allow you to write a single string or multiple lines to a file.

Python can be used to perform operations on a file. (read & write data)

Open the File	→	Perform Operation	→	Close the File
open()		read() or write()		close()

File Opening in Python

File opening in Python involves using the open() function, which allows you to access and interact with files.

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

filename: Specifies the name of the file you want to open, including its path if it's not in the current directory.

mode: Specifies the mode in which the file is opened.

```
# Opening a file in read mode
file = open('example.txt', 'r')
```

Different Modes

- **'r': Read mode.** Opens a file for reading. (default mode)
- **'w': Write mode** . Opens a file for writing. If the file exists, it truncates the file to zero length. If the file does not exist, it creates a new file.
- **'a': Append mode.** Opens a file for appending data. The file pointer is at the end of the file if the file exists. If the file does not exist, it creates a new file for writing.
- **'b': Binary mode.** This can be added to any of the above modes (e.g., 'rb', 'wb', 'ab') to work with binary files.
- **'+':** Open a file for updating (reading and writing). (eg. r+, w+)

It's recommended to use the with statement when opening files. This ensures that the file is properly closed after its suite finishes, even if an exception is raised.

```
with open('example.txt', 'r') as file:
```

Reading files in Python

It involves several methods to retrieve data stored in files. The main methods for reading files are:

- **read()**
- **readline()**
- **readlines()**

Reading the Entire File: The read() method reads the entire content of the file at once and returns it as a single string. This is useful for small files but can be inefficient for large files as it loads all data into memory.

```
file = open('example.txt', 'r')
content = file.read()
print(content)
file.close()
```

Reading Line by Line: The readline() method reads one line at a time from the file. This is useful for processing large files line by line, as it doesn't load the entire file into memory at once.

```
file = open('example.txt', 'r')

line = file.readline()

print(line)

file.close()
```

.strip() removes the newline character

```
print(line.strip())
```

Reading All Lines: The readlines() method reads all lines of the file and returns them as a list of strings, where each string is a line from the file. This method can be convenient for iterating over lines but may be inefficient for very large files.

```
file = open('example.txt', 'r')

lines = file.readlines()

print(lines)

file.close()

o/p = ['fsfs\n', 'sfsfsfsf\n', 'sfsf']
```

.strip() removes the newline character

```
print(lines.strip())
```

Let's Code..

Writing to files in Python

It is an essential operation for data storage, logging, configuration files, and more. Here are the basic methods for writing to files, including writing a single string, writing multiple lines, and appending to a file.

1. Writing a Single String

The write() method allows you to write a single string to a file. This is useful for simple text output. If the file does not exist, it will be created. If it does exist, the file will be truncated (emptied) before writing the new content.

```
file = open('example.txt', 'w')

file.write('Hello, World!\n')

file.close()
```

2. Writing Multiple Lines

The **writelines()** method allows you to write a list of strings to a file. Each string in the list is written to the file sequentially. This method does not add new lines automatically, so each string should end with a newline character if needed.

```
lines = ['First line\n', 'Second line\n', 'Third line\n']

file = open('example.txt', 'w')

file.writelines(lines)

file.close()
```

3. Appending to a File

To append data to an existing file without truncating it, you can open the file in append mode ('a'). The new data will be added at the end of the file.

```
file = open('example.txt', 'a')

file.write('This line will be appended.\n')

file.close()
```

By using these methods, you can efficiently write data to files in Python, ensuring proper file management and data persistence. Always remember to close the file after writing to ensure that all data is flushed from the buffer and saved to the disk, and to release system resources.

Multi Atoms Plus

“r" Open text file for reading. The stream is positioned at the beginning of the file.

“r+" Open for reading + writing. The stream is positioned at the beginning of the file.

“w" Truncate file to zero length or create text file for writing. The stream →beginning.

“w+" Open for reading + writing. The file is created if it does not exist, otherwise it is truncated. The stream → beginning of the file.

“a" Open for writing. The file is created if it does not exist. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the then current end of file.

“a+" Open for reading + writing. The file is created if it does not exist. The stream is positioned at the end of the file.

Let’s Code..

Deleting the file in Python

To delete a file in Python, you can use the os.remove() function from the os module. Here's how you can delete a file:

```
import os

file_path = 'example.txt'

if os.path.exists(file_path):

    os.remove(file_path)

    print(f"{file_path} successfully deleted.")

else:

    print(f"{file_path} does not exist.")
```

Q. WAPP to write the no. of letters and digits in the given input String in a file object.

```
input_string = input("Enter a string: ")

letters_count = 0

digits_count = 0


for char in input_string:

    if char.isalpha():

        letters_count += 1

    elif char.isdigit():

        digits_count += 1
```

```
# Write counts to a file

file_path = 'letter_digit_counts.txt'

file = open(file_path, 'w')

file.write(f"Number of letters: {letters_count}\n")

file.write(f"Number of digits: {digits_count}\n")

file.close()
```

Let's Code..

File Handler in Python

File Handler Basics:

- A file handler, or file object, is an interface to interact with files in Python programs.
- It is created when a file is opened using the open() function.

Operations:

- Reading: Use methods like read(), readline(), or iterate over lines with a loop.
- Writing: Employ write() to add content to a file, or writelines() to write multiple lines at once.
- Moving the Pointer: Adjust the file pointer with seek() to navigate through the file.
- Querying Position: Determine the current position with tell().

Multi Atoms Plus

Modes: as previous

Closing Files:

- Call close( ) on the file handler to free up resources once operations are done.

Error Handling:

- Handle potential errors like FileNotFoundError or IOError when opening or manipulating files.

Best Practices:

- Always close files after use to prevent resource leaks.
- Utilize context managers (with statement) for cleaner and safer file handling.
- Handle exceptions to gracefully manage file-related errors.

seek() Function in Python

It is used to change the current position (or offset) of the file pointer within a file. This function is particularly useful when you need to navigate to a specific location in a file to read or write data.

```
file_object.seek(offset, whence)
```

offset: It specifies the number of bytes to move the file pointer.

whence: It specifies the reference point from where the offset is calculated. It can take one of the following values:

- 0 (default): Start of the file
- 1: Current position of the file pointer
- 2: End of the file

Moving the File Pointer:

- When you open a file, the file pointer starts at the beginning (0 offset).
- seek(offset, 0) moves the pointer offset bytes from the beginning of the file.
- seek(offset, 1) moves the pointer offset bytes from the current position.
- seek(offset, 2) moves the pointer offset bytes from the end of the file (a negative offset is usually used in this case).

```
# Open a file
file = open('example.txt', 'r')
# Move pointer to the 10th byte from the start
file.seek(10, 0)
print(file.read(5))  # Reads 5 bytes from the current position (10th byte)
# Move 5 bytes forward from the current position
file.seek(5, 1)
print(file.read(10))  # Reads 10 bytes from the current position (15th byte)
# Move to the 10 bytes before the end of the file
file.seek(-10, 2)
print(file.read(5))  # Reads 5 bytes from this position (10 bytes before the end)
# Close the file
file.close()
```

tell() function in Python

The tell() function returns an integer that represents the current position of the file pointer in bytes from the beginning of the file.'

```
file_object.tell()
```

Understanding and utilizing tell() allows precise control over file operations, especially when dealing with large files or when needing to track and manage file positions dynamically during file processing in Python.

Let's Code..

Thanks for Watching

Please Subscribe and Share Multi Atoms Plus

Join Telegram for Notes

Python Programming Unit-5 One Shot

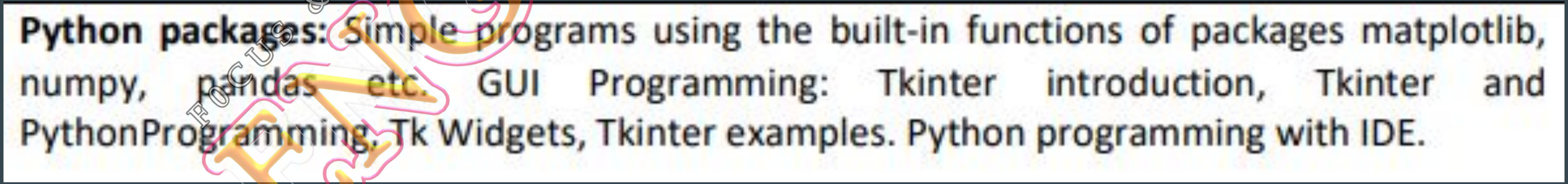
...

(BCC302 / BCC402/ BCC302H / BCC402H)

Python Packages

Multi Atoms Plus

Unit-5 Aktu Updated Syllabus



Python packages: Simple programs using the built-in functions of packages matplotlib, numpy, pandas etc. GUI Programming: Tkinter introduction, Tkinter and PythonProgramming, Tk Widgets, Tkinter examples. Python programming with IDE.

Python Packages

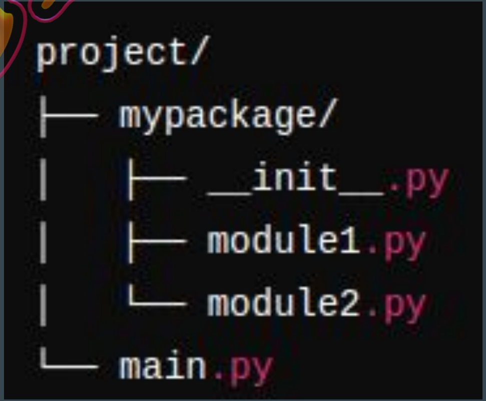
A Python package is a collection of modules bundled together. These modules can include functions, classes, and variables that can be used in your Python programs. Packages help to organize and structure code, making it more modular, reusable, and maintainable.

Key Concepts:

1. Module: A single file containing Python code (functions, classes, variables, etc.) with a .py extension.
2. Package: A directory containing multiple modules and an __init__.py file, which makes it a package. The __init__.py file can be empty or execute initialization code for the package.

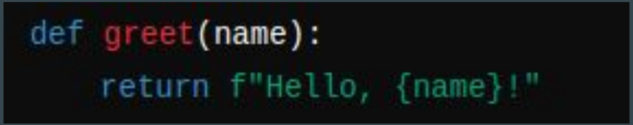
Creating a Package

1. Directory Structure:



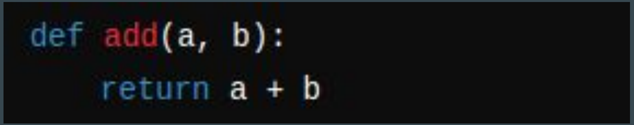
```
project/
├── mypackage/
│   ├── __init__.py
│   ├── module1.py
│   └── module2.py
└── main.py
```

2. module1.py:



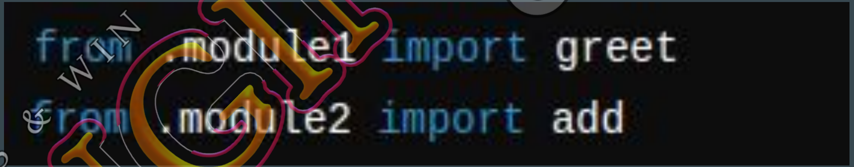
```python
def greet(name):
    return f"Hello, {name}!"
```

3. module2.py:



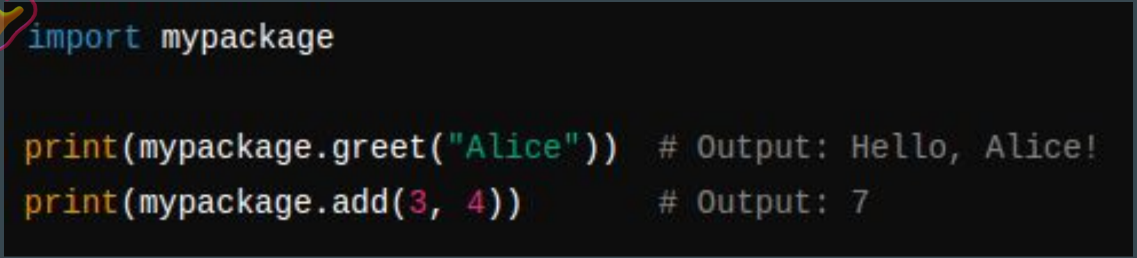
```python
def add(a, b):
    return a + b
```

4. __init__.py:



```python
from .module1 import greet
from .module2 import add
```

Using the Package



```python
import mypackage

print(mypackage.greet("Alice"))  # Output: Hello, Alice!
print(mypackage.add(3, 4))       # Output: 7
```

Benefits of Using Packages

1. Modularity: Break down large programs into smaller, manageable, and reusable modules.
2. Namespace Management: Avoid name conflicts by organizing code into separate namespaces.
3. Reusability: Easily reuse code across different projects.
4. Maintainability: Easier to manage and maintain code.

Popular Python Packages

Matplotlib: For creating static, animated, and interactive visualizations.

Numpy: For numerical computations and operations on large arrays and matrices.

Pandas: For data manipulation and analysis.

Requests: For making HTTP requests.

Installing Packages

Use pip, the Python package installer, to install packages from the Python Package Index (PyPI).

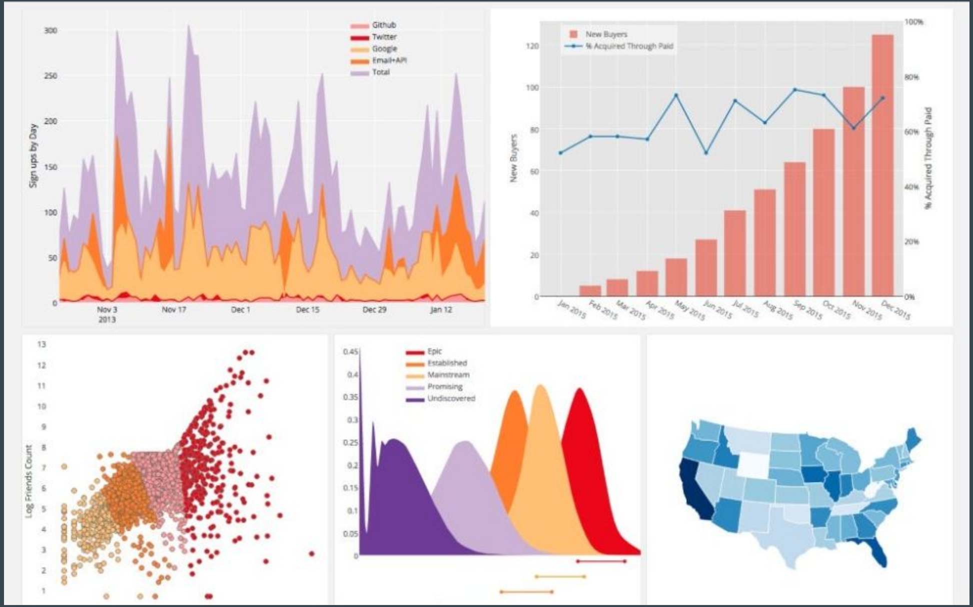
```
pip install package_name
```

Introduction to Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It is widely used for data visualization in scientific computing, data science, and machine learning.

Installation

```
pip install matplotlib
```



Types of Matplotlib

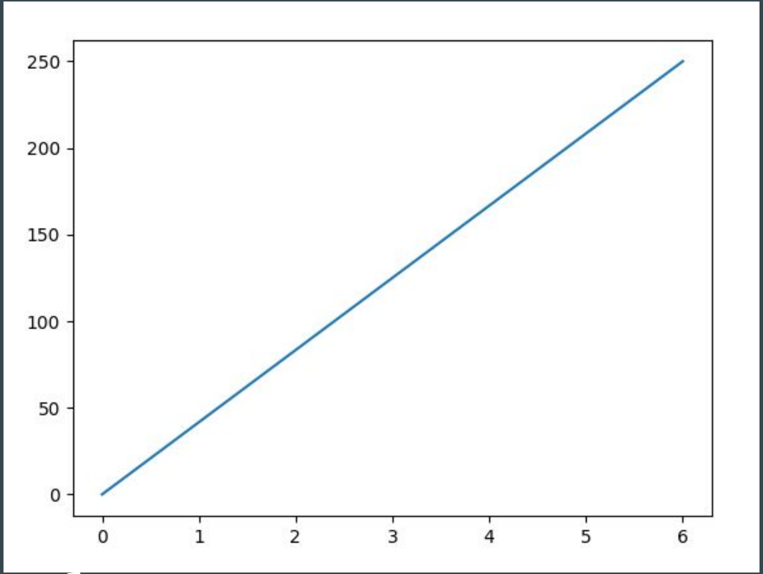
1. **Line Plot** - Displays data trends over time.
2. **Scatter Plot** - Shows relationships between variables.
3. **Bar Chart** - Compares categorical data values.
4. **Histogram** - Represents data distribution frequencies.
5. **Pie Chart** - Illustrates proportions of a whole.
6. **Box Plot** - Summarizes data distribution quartiles.
7. **Violin Plot** - Displays data distribution and density.
8. **Heatmap** - Visualizes matrix-like data with color.
9. **Area Plot** - Fills the area under a curve.
10. **3D Plot** - Represents three-dimensional data visually.
11. **Subplots** - Multiple plots in one figure.

Matplotlib Pyplot

Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the plt alias: `import matplotlib.pyplot as plt`

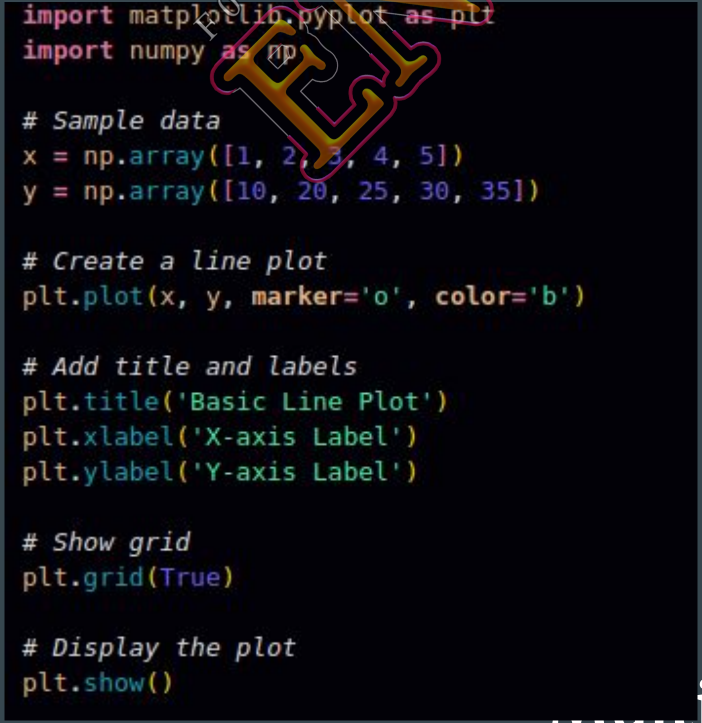
Now the Pyplot package can be referred to as plt.

```
import matplotlib.pyplot as plt

import numpy as np

xpoints = np.array([0, 6])

ypoints = np.array([0, 250])

plt.plot(xpoints, ypoints)

plt.show()
```

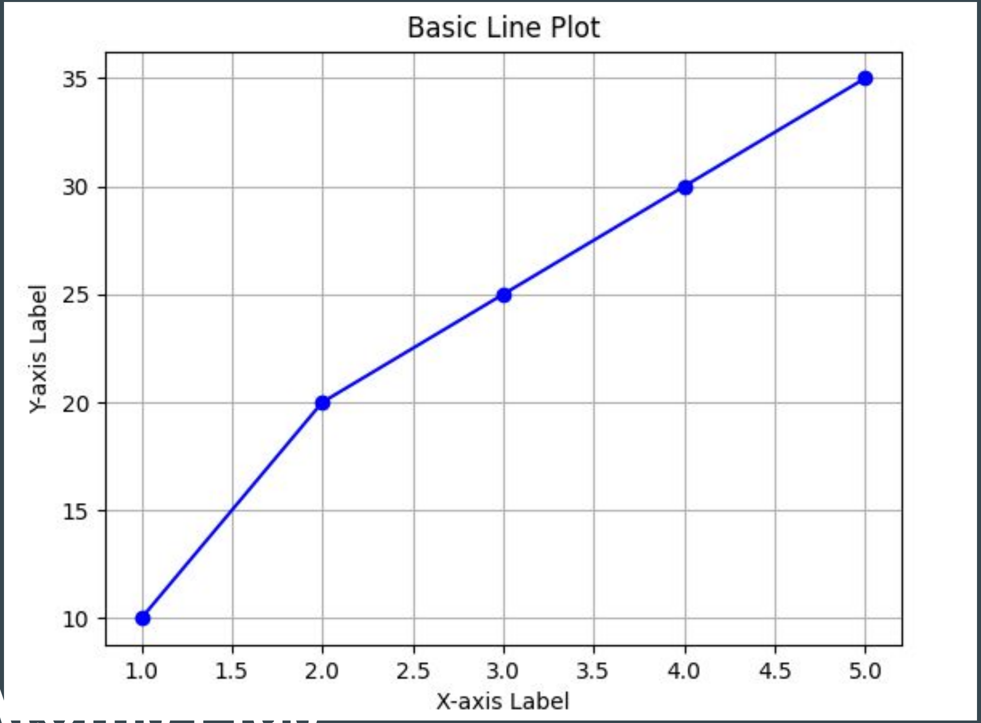


Line plot with Matplotlib:

A line plot in Matplotlib is used to display data points connected by straight lines. It's particularly useful for showing trends over time or continuous data.



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

# Sample data
x = np.array([1, 2, 3, 4, 5])
y = np.array([10, 20, 25, 30, 35])

# Create a line plot
plt.plot(x, y, marker='o', color='b')

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

# Show grid
plt.grid(True)

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



Customizations

Line Color:

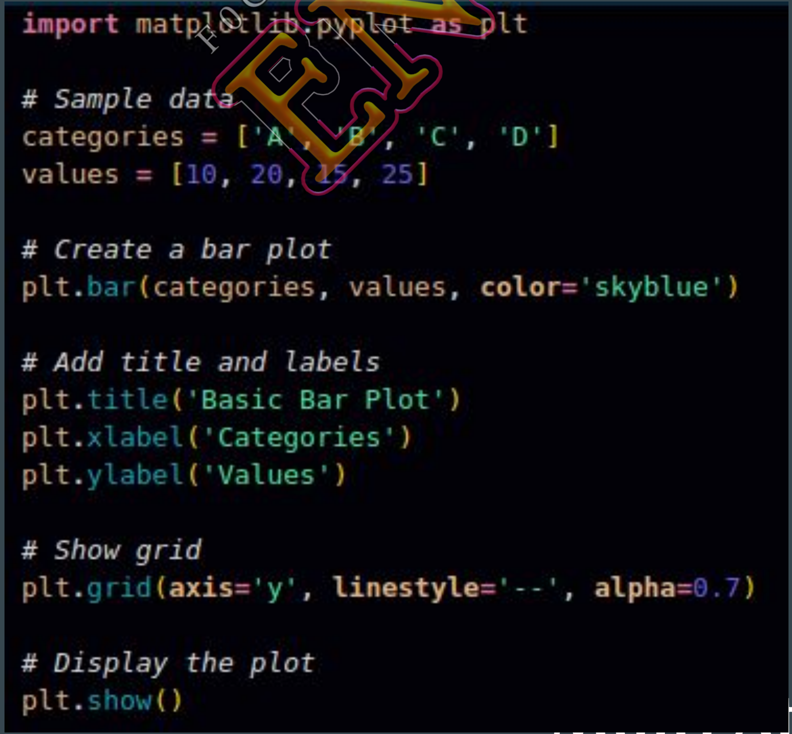
Use color names (e.g., 'blue'), hex codes (e.g., '#FF5733'), or RGB tuples (e.g., (0.1, 0.2, 0.5)).

Markers:

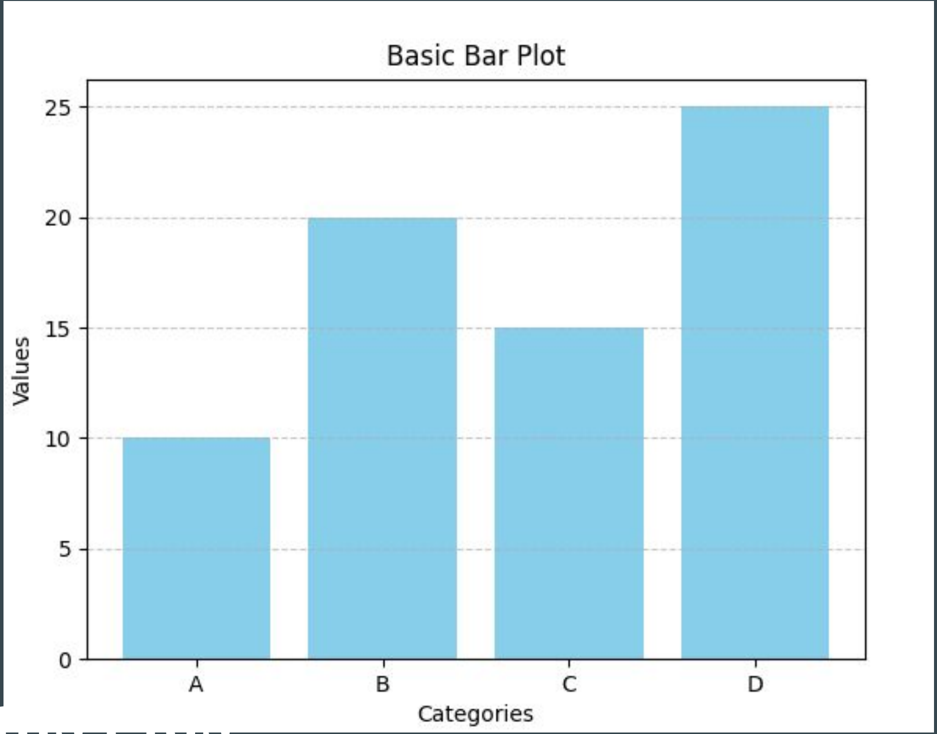
Types include '.' (point), 'o' (circle), 's' (square), '^' (triangle), etc.

Bar plot with Matplotlib

A bar plot (or bar chart) in Matplotlib is used to display categorical data with rectangular bars. Each bar's length represents the value of the category it represents.

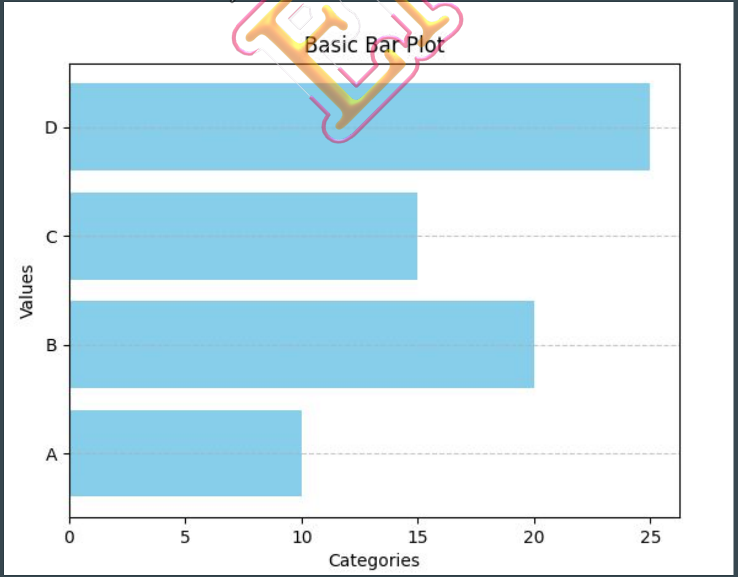


```
import matplotlib.pyplot as plt

# Sample data
categories = ['A', 'B', 'C', 'D']
values = [10, 20, 15, 25]

# Create a bar plot
plt.bar(categories, values, color='skyblue')

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

# Show grid
plt.grid(axis='y', linestyle='--', alpha=0.7)

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



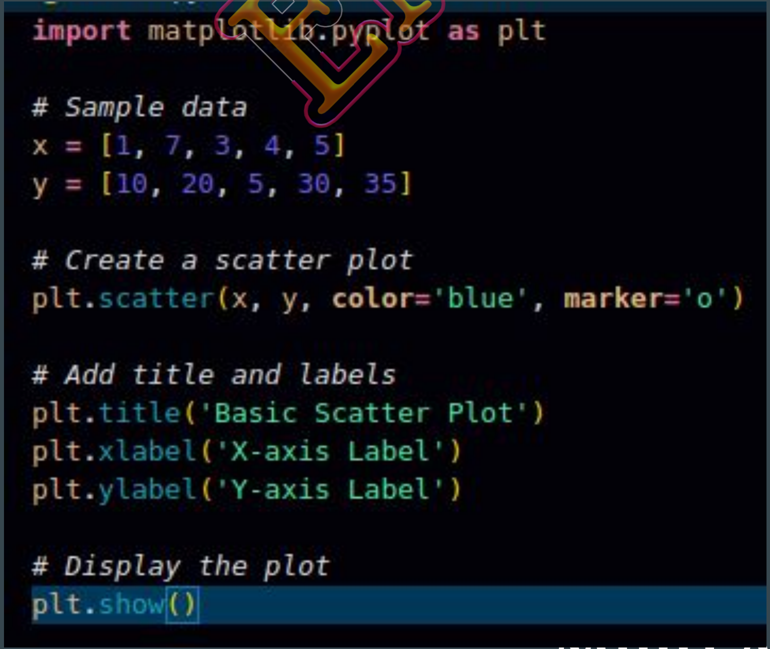
Horizontal Bar Plot:

```
plt.barh(categories, values, color='skyblue')
```

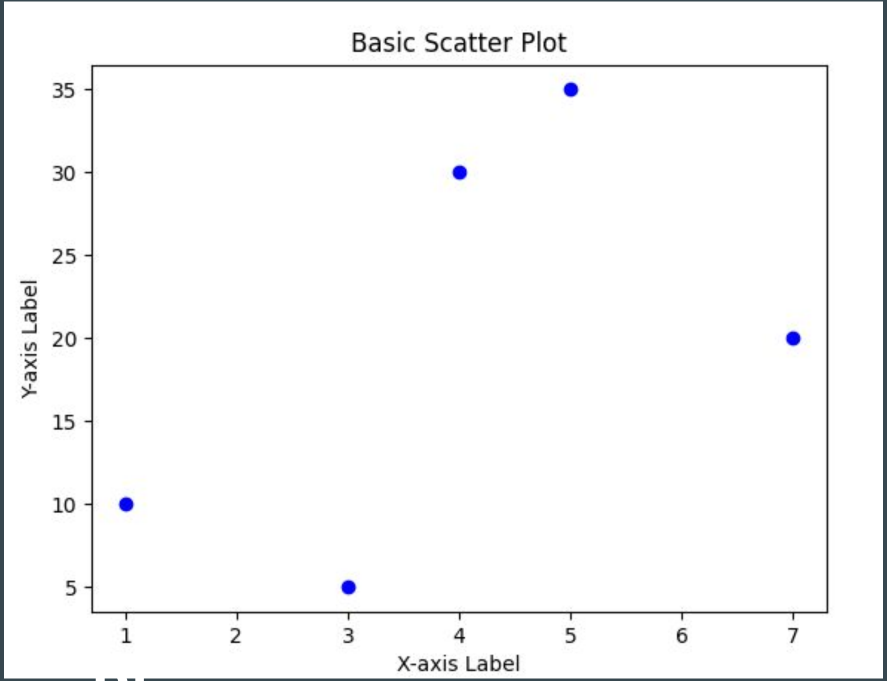


Scatter plot with Matplotlib

A scatter plot in Matplotlib is used to display the relationship between two variables by plotting data points on a Cartesian plane. Each point represents a pair of values from two datasets.

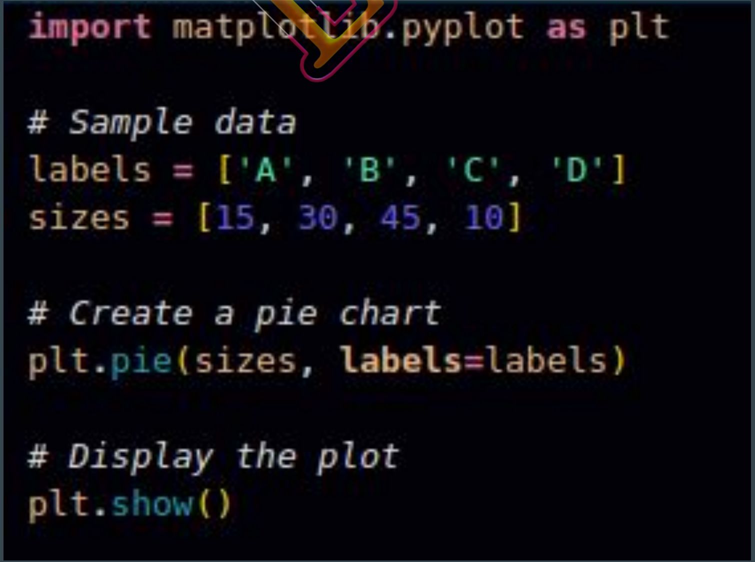


```
import matplotlib.pyplot as plt

# Sample data
x = [1, 7, 3, 4, 5]
y = [10, 20, 5, 30, 35]

# Create a scatter plot
plt.scatter(x, y, color='blue', marker='o')

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

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

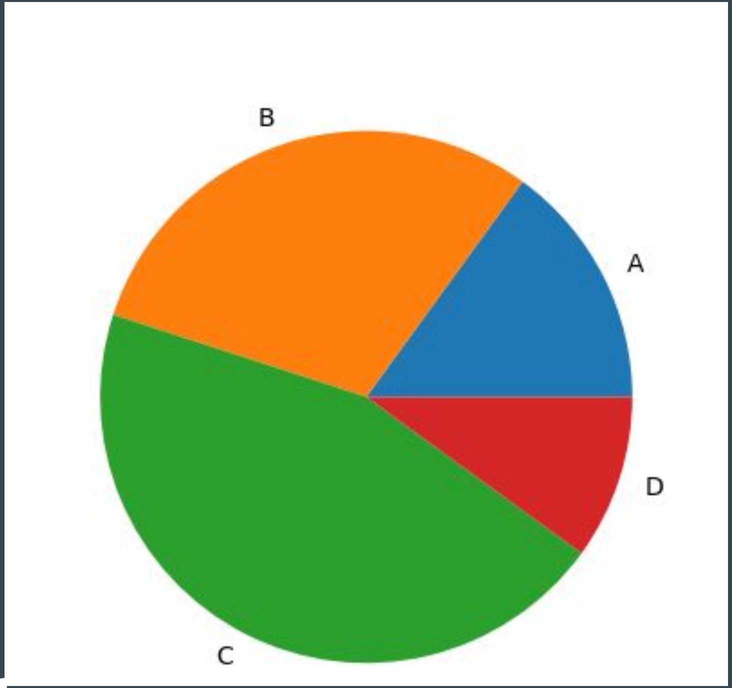


pie chart with Matplotlib:

A pie chart in Matplotlib is used to display data as slices of a circle, representing proportions of a whole. Each slice corresponds to a category and its size represents the proportion of that category relative to the total.

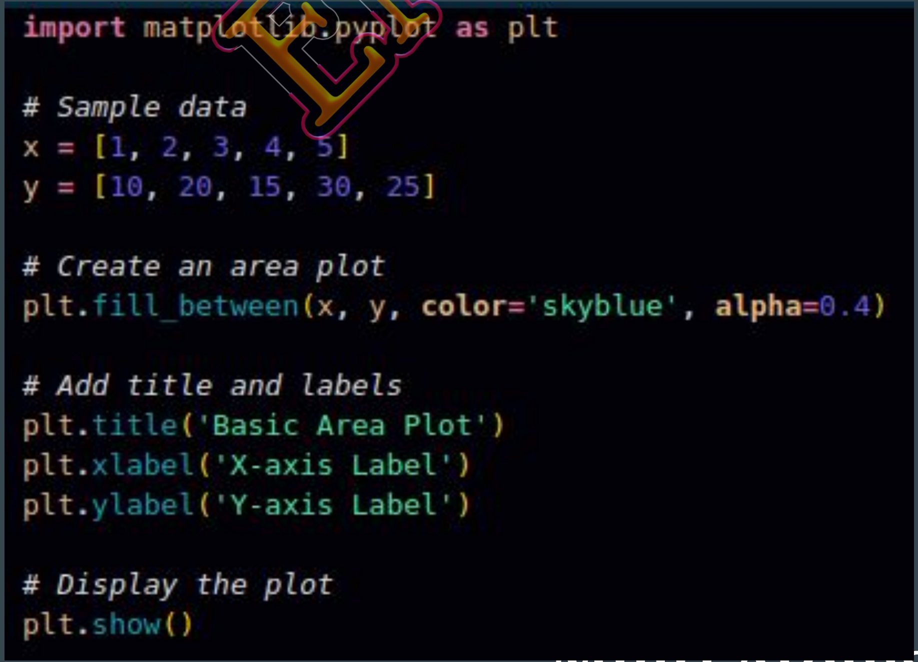


```
import matplotlib.pyplot as plt

# Sample data
labels = ['A', 'B', 'C', 'D']
sizes = [15, 30, 45, 10]

# Create a pie chart
plt.pie(sizes, labels=labels)

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

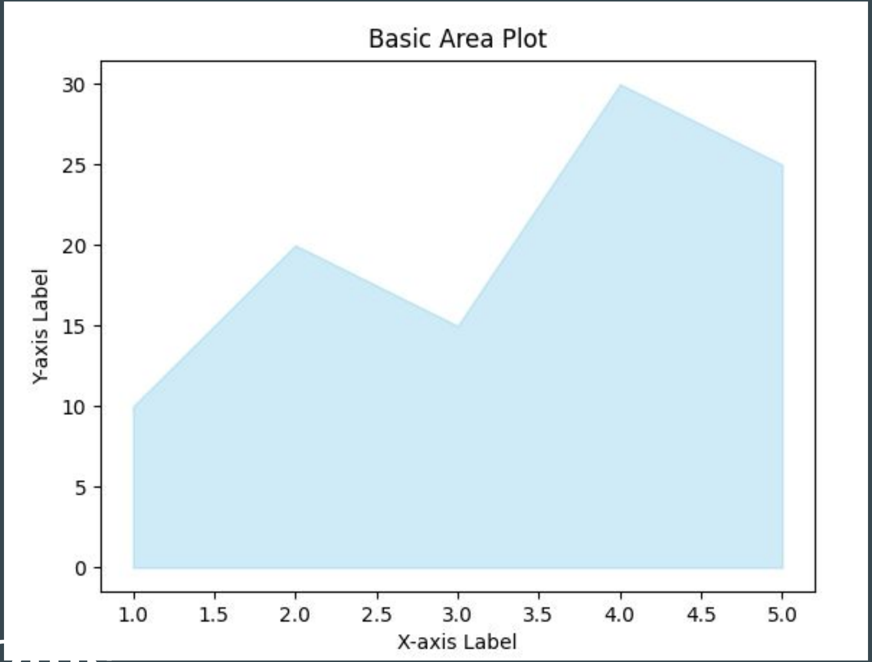


area plot in Matplotlib

An area plot in Matplotlib is used to display data where the area under a line is filled in, making it easy to visualize cumulative totals or the relative size of different categories over time.



```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 20, 15, 30, 25]

# Create an area plot
plt.fill_between(x, y, color='skyblue', alpha=0.4)

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

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



NumPy

NumPy is a fundamental library for numerical computing in Python. It provides support for arrays, matrices, and a wide range of mathematical functions to operate on these data structures.

Key Features of NumPy

N-dimensional Arrays:

numpy.array(): Creates arrays for efficient storage and manipulation of numerical data.

Mathematical Functions:

Functions for mathematical operations, including addition, subtraction, multiplication, and complex functions like trigonometric functions.

Array Operations:

Operations like element-wise addition, multiplication, and other mathematical operations on arrays.

Linear Algebra:

Functions for linear algebra operations such as dot products, matrix multiplication, and eigenvalue decomposition.

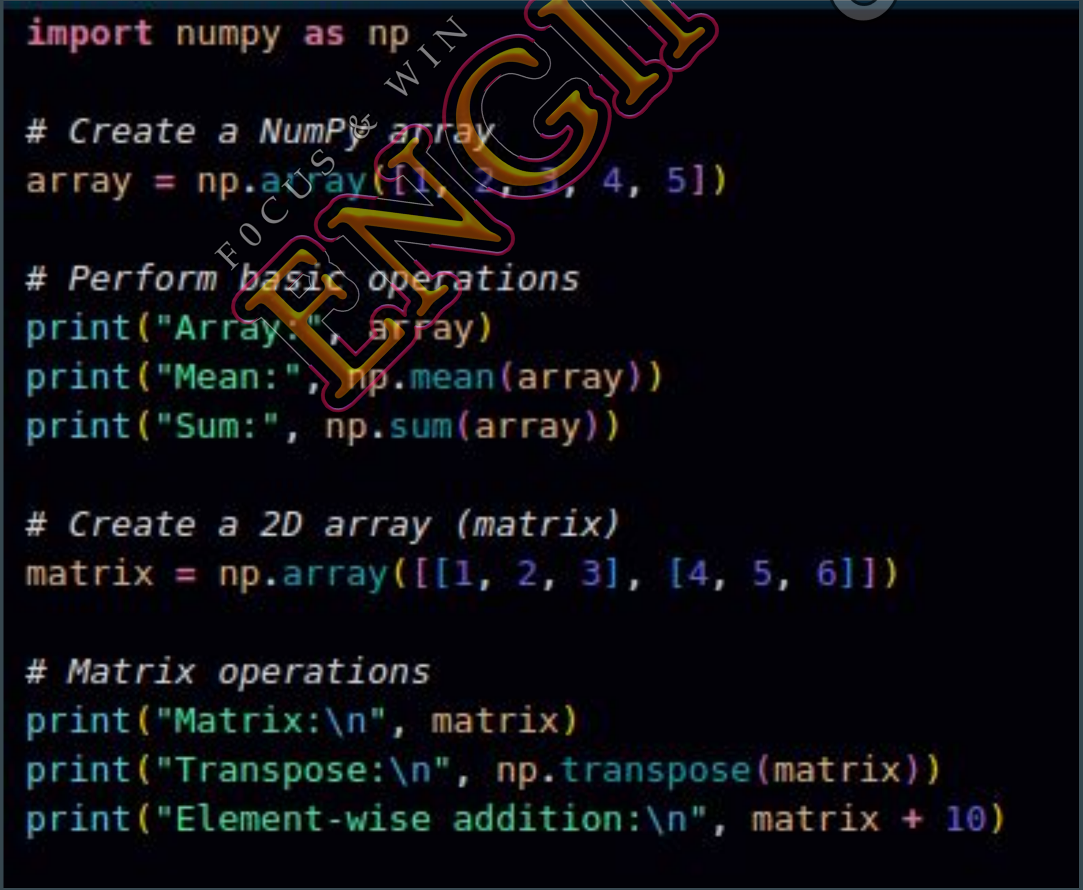
Random Number Generation:

Functions to generate random numbers and distributions.

Array Manipulation:

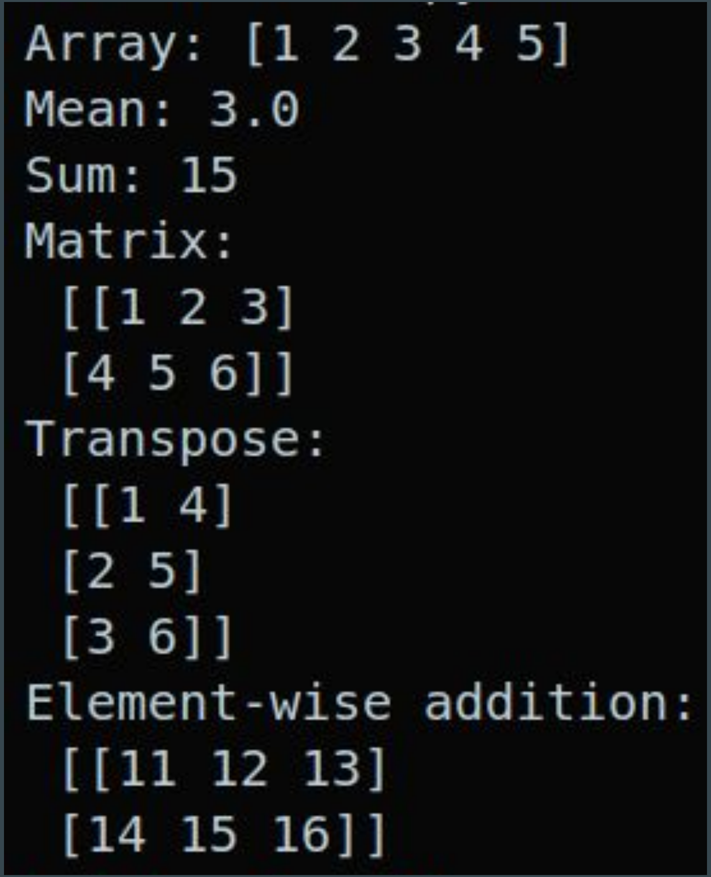
Functions for reshaping, slicing, and aggregating data.

Example



```python
import numpy as np

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

# Perform basic operations
print("Array:", array)
print("Mean:", np.mean(array))
print("Sum:", np.sum(array))

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

# Matrix operations
print("Matrix:\n", matrix)
print("Transpose:\n", np.transpose(matrix))
print("Element-wise addition:\n", matrix + 10)
```

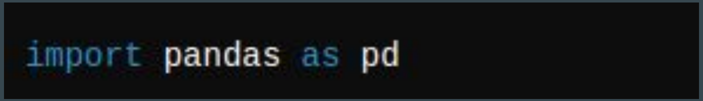
Output



```
Array: [1 2 3 4 5]
Mean: 3.0
Sum: 15
Matrix:
 [[1 2 3]
 [4 5 6]]
Transpose:
 [[1 4]
 [2 5]
 [3 6]]
Element-wise addition:
 [[11 12 13]
 [14 15 16]]
```

Pandas

Pandas is a powerful data manipulation and analysis library for Python. It provides data structures like DataFrames and Series that make it easy to handle and analyze large datasets. Pandas is especially useful for working with tabular data and provides a variety of functions to clean, transform, and analyze data.



```
import pandas as pd
```

Core Features of Pandas

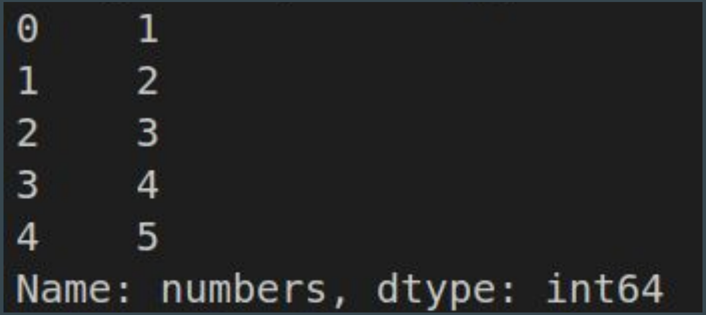
Data Structures:

Series: A one-dimensional labeled array that can hold any data type.

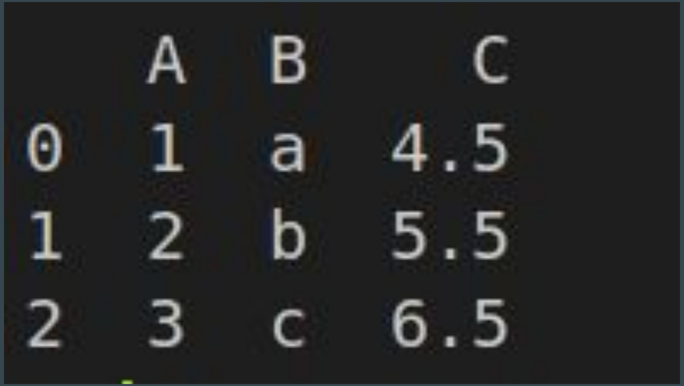
DataFrame: A two-dimensional labeled data structure with columns of potentially different data types.

Creating Data Structures

```
# Creating a Series
s = pd.Series([1, 2, 3, 4, 5], name='numbers')
print(s)
# Creating a DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3],
    'B': ['a', 'b', 'c'],
    'C': [4.5, 5.5, 6.5]
})
print(df)
```



```
0    1
1    2
2    3
3    4
4    5
Name: numbers, dtype: int64
```

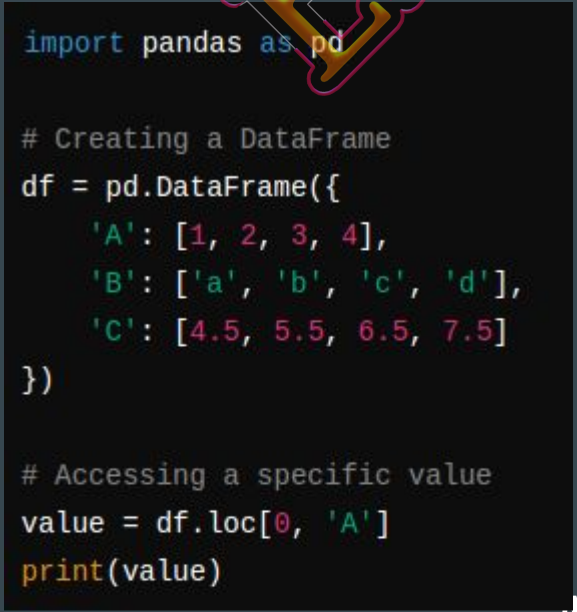


```
   A  B    C
0  1  a  4.5
1  2  b  5.5
2  3  c  6.5
```

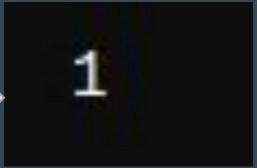
Data Manipulation:

Indexing and Selection: Accessing and manipulating data using labels and integer-based indexing.

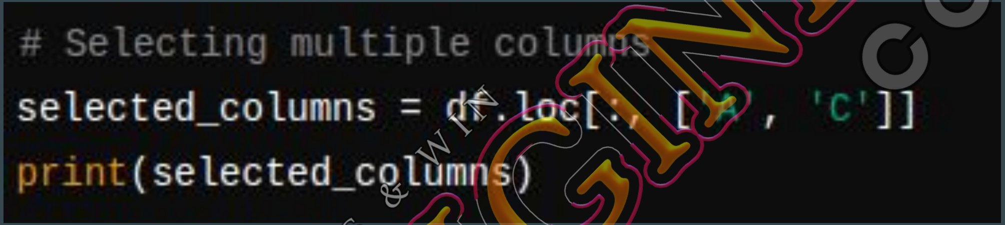
1. Indexing and Selection using Labels (loc): Accessing data using labels.



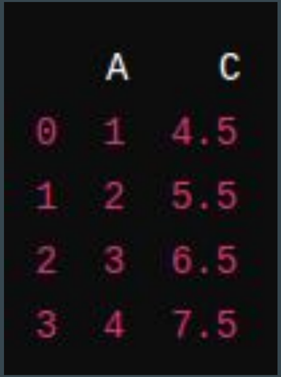
```
import pandas as pd

# Creating a DataFrame
df = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': ['a', 'b', 'c', 'd'],
    'C': [4.5, 5.5, 6.5, 7.5]
})

# Accessing a specific value
value = df.loc[0, 'A']
print(value)
```



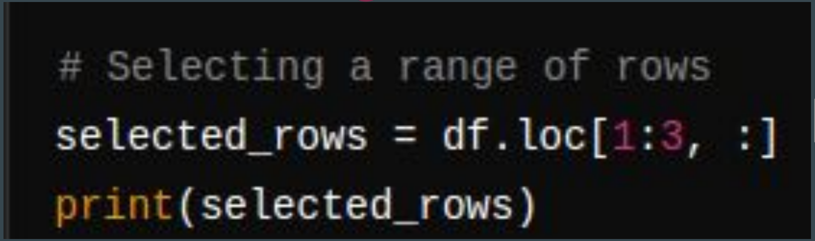
```
1
```



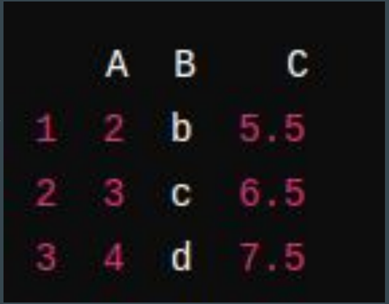
```
# Selecting multiple columns
selected_columns = df.loc[:, ['A', 'C']]
print(selected_columns)
```



```
   A    C
0  1  4.5
1  2  5.5
2  3  6.5
3  4  7.5
```

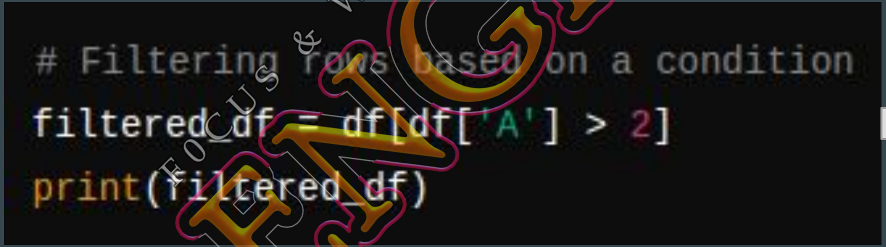


```
# Selecting a range of rows
selected_rows = df.loc[1:3, :]
print(selected_rows)
```

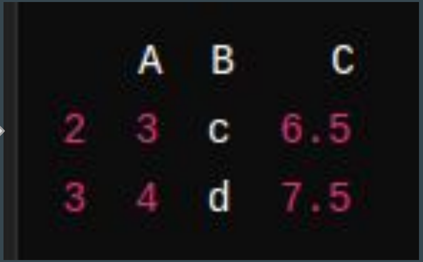


```
   A  B    C
1  2  b  5.5
2  3  c  6.5
3  4  d  7.5
```

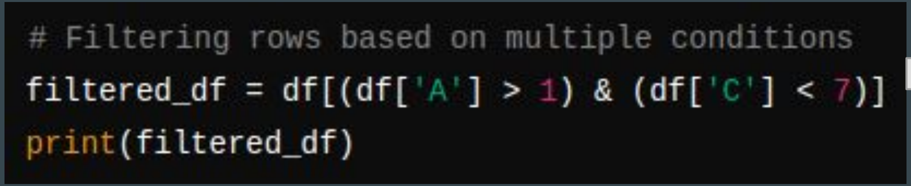
Filtering: Methods for filtering



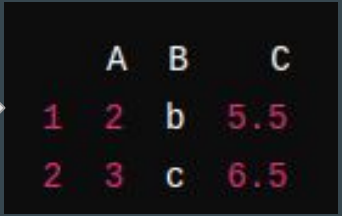
```
# Filtering rows based on a condition
filtered_df = df[df['A'] > 2]
print(filtered_df)
```



```
   A  B    C
2  3  c  6.5
3  4  d  7.5
```



```
# Filtering rows based on multiple conditions
filtered_df = df[(df['A'] > 1) & (df['C'] < 7)]
print(filtered_df)
```

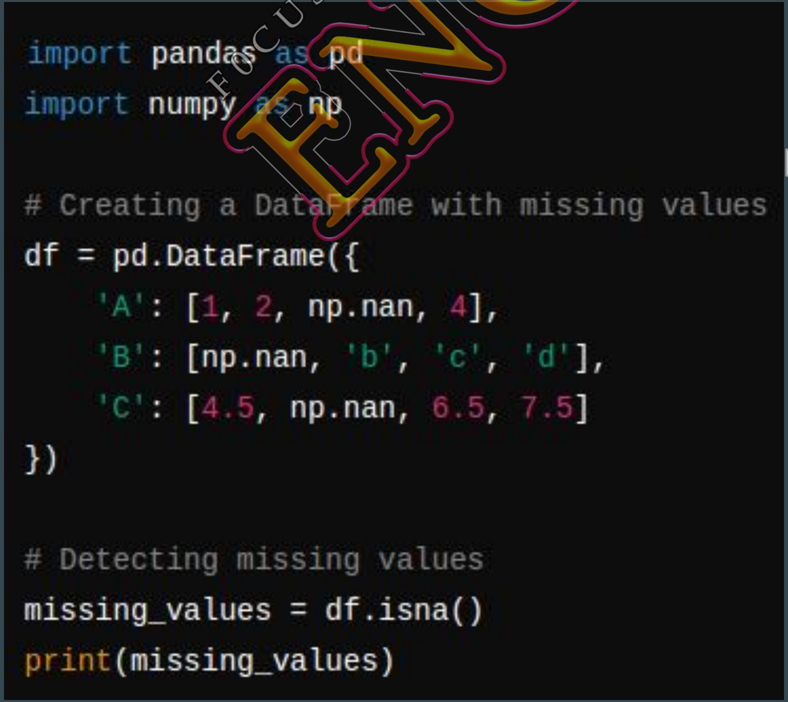


```
   A  B    C
1  2  b  5.5
2  3  c  6.5
```

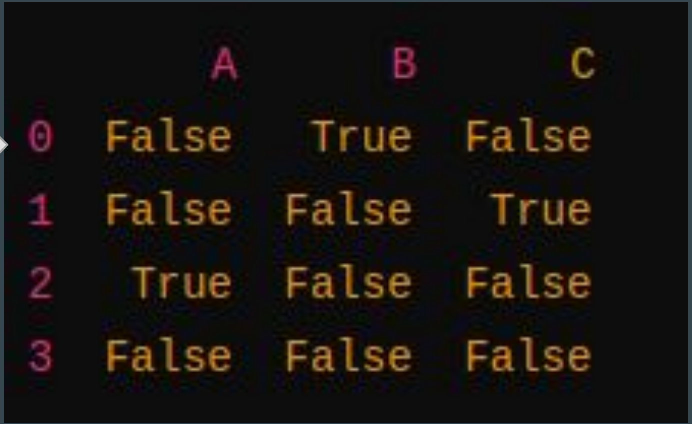
Data Cleaning:

Handling Missing Data: Functions for detecting, filling, and dropping missing values.

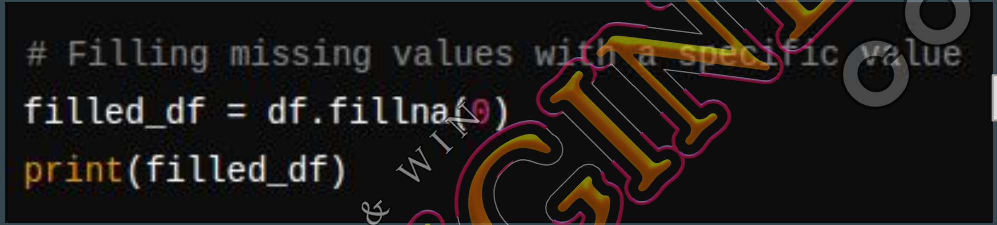
Data Transformation: Tools for reshaping and transforming data.



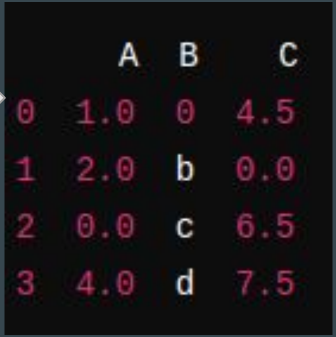
```
import pandas as pd
import numpy as np

# Creating a DataFrame with missing values
df = pd.DataFrame({
    'A': [1, 2, np.nan, 4],
    'B': [np.nan, 'b', 'c', 'd'],
    'C': [4.5, np.nan, 6.5, 7.5]
})

# Detecting missing values
missing_values = df.isna()
print(missing_values)
```



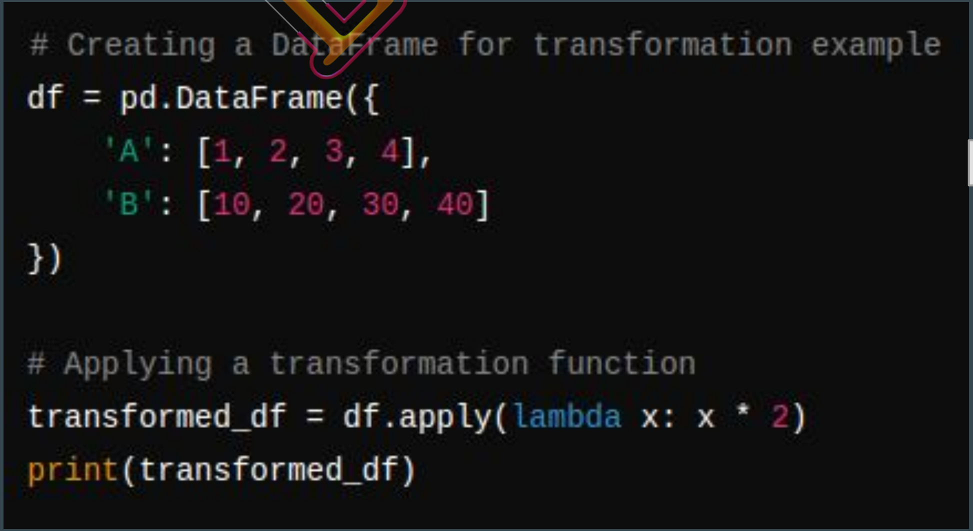
```
       A      B      C
0  False   True  False
1  False  False   True
2   True  False  False
3  False  False  False
```



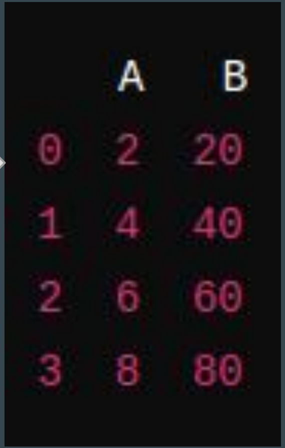
```python
# Filling missing values with a specific value
filled_df = df.fillna(0)
print(filled_df)
```



```
     A  B    C
0  1.0  0  4.5
1  2.0  b  0.0
2  0.0  c  6.5
3  4.0  d  7.5
```



```python
# Creating a DataFrame for transformation example
df = pd.DataFrame({
    'A': [1, 2, 3, 4],
    'B': [10, 20, 30, 40]
})

# Applying a transformation function
transformed_df = df.apply(lambda x: x * 2)
print(transformed_df)
```



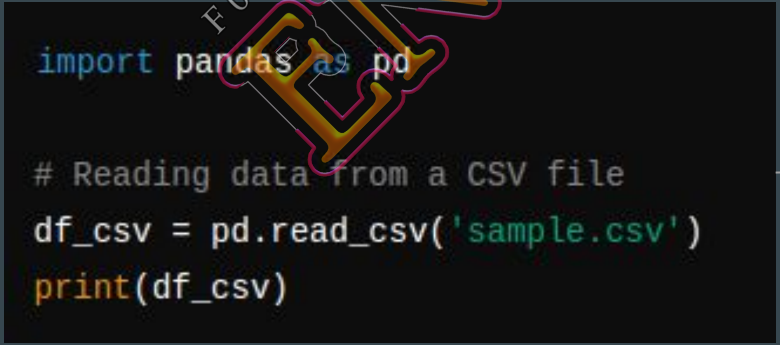
```
   A   B
0  2  20
1  4  40
2  6  60
3  8  80
```

Multi Atoms Plus

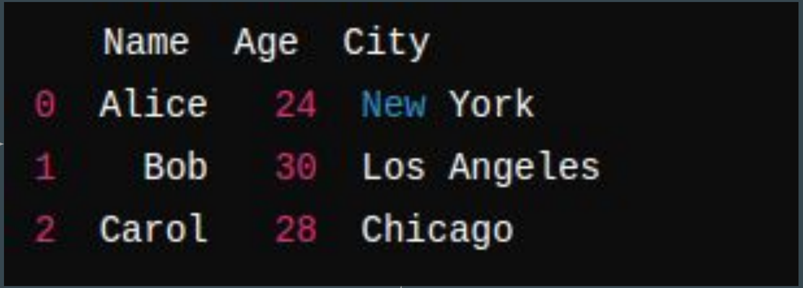
Data Input/Output:

Reading/Writing Data: Functions for reading from and writing to various file formats including CSV, Excel

1. **Reading CSV Files: Using pd.read_csv() to read data from a CSV file.**

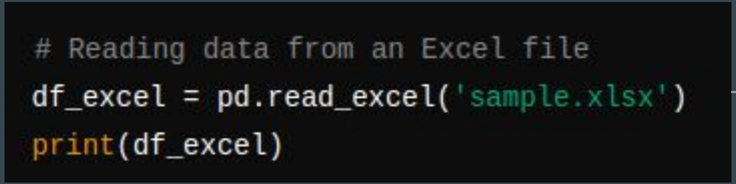


```
import pandas as pd

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

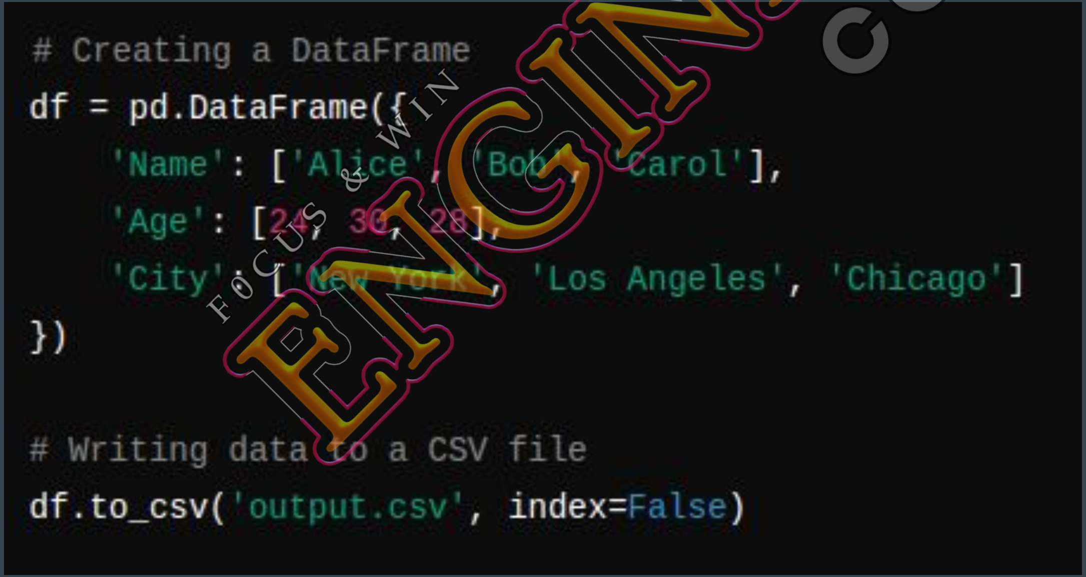


```
    Name  Age         City
0  Alice   24     New York
1    Bob   30  Los Angeles
2  Carol   28      Chicago
```

2. **Reading Excel Files: Using pd.read_excel() to read data from an Excel file.**



```
# Reading data from an Excel file
df_excel = pd.read_excel('sample.xlsx')
print(df_excel)
```



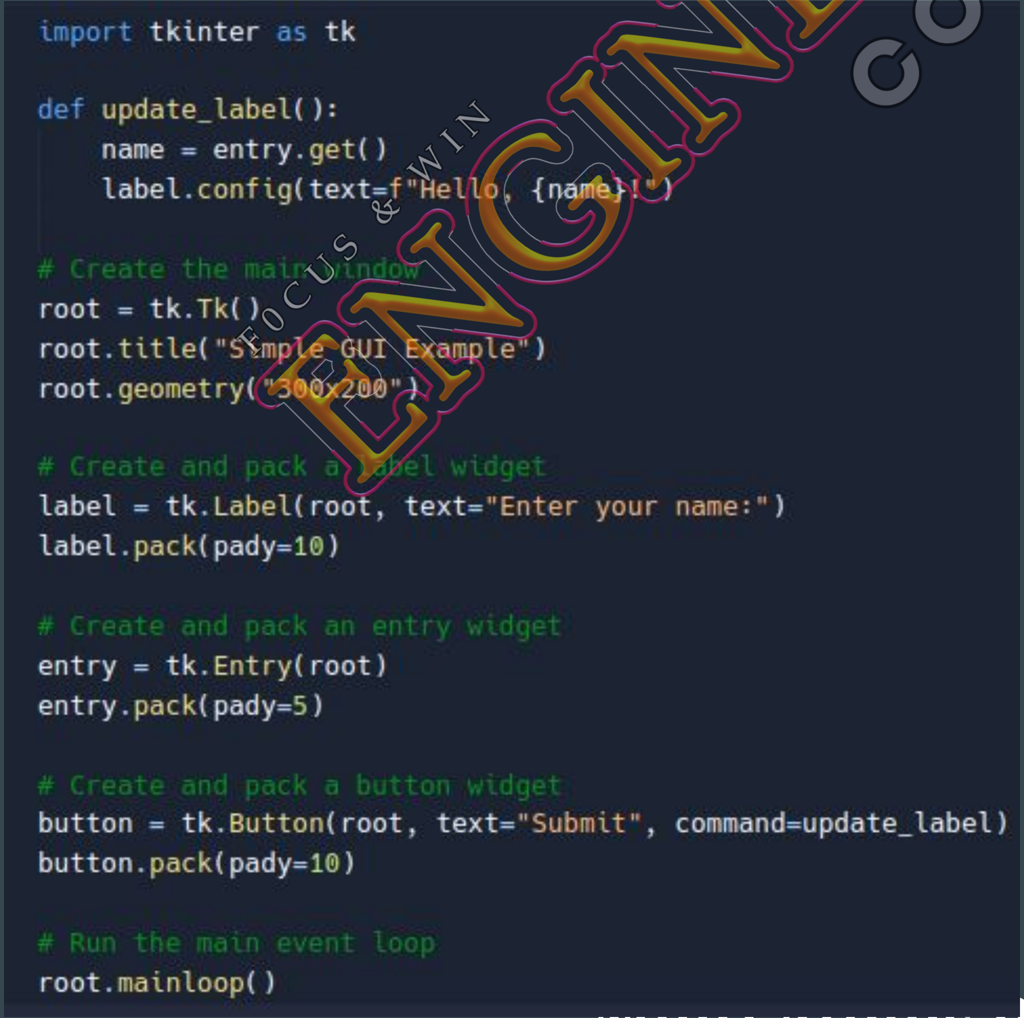
```
# Creating a DataFrame
df = pd.DataFrame({
    'Name': ['Alice', 'Bob', 'Carol'],
    'Age': [24, 30, 28],
    'City': ['New York', 'Los Angeles', 'Chicago']
})

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

The to_csv('output.csv', index=False) function writes the DataFrame df to the CSV file output.csv. The index=False parameter ensures that the row indices are not written to the file.

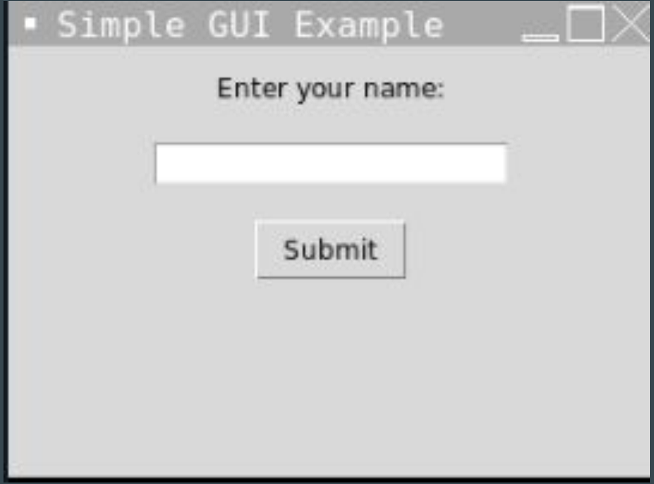
What is a GUI in Python?

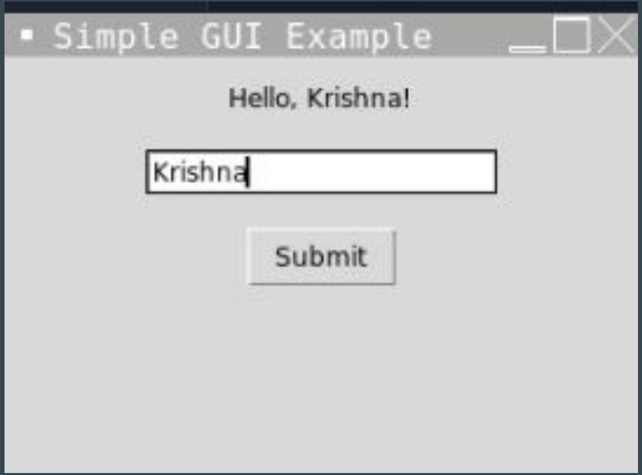
A Graphical User Interface (GUI) in Python is a visual interface that allows users to interact with the application through graphical elements like windows, buttons, text fields, and other widgets, rather than using text-based commands. GUIs make applications user-friendly and visually appealing.

Python offers several libraries to create GUIs, with Tkinter being the most commonly used due to its simplicity and integration with Python's standard library. Other popular GUI frameworks include PyQt, Kivy, and wxPython.



```python
import tkinter as tk

def update_label():
    name = entry.get()
    label.config(text=f"Hello, {name}!")

# Create the main window
root = tk.Tk()
root.title("Simple GUI Example")
root.geometry("300x200")

# Create and pack a label widget
label = tk.Label(root, text="Enter your name:")
label.pack(pady=10)

# Create and pack an entry widget
entry = tk.Entry(root)
entry.pack(pady=5)

# Create and pack a button widget
button = tk.Button(root, text="Submit", command=update_label)
button.pack(pady=10)

# Run the main event loop
root.mainloop()
```





Explanation

1. Import the Tkinter module:

```
import tkinter as tk
```

This imports the Tkinter module and makes it available in the script.

2. Define the function to update the label:

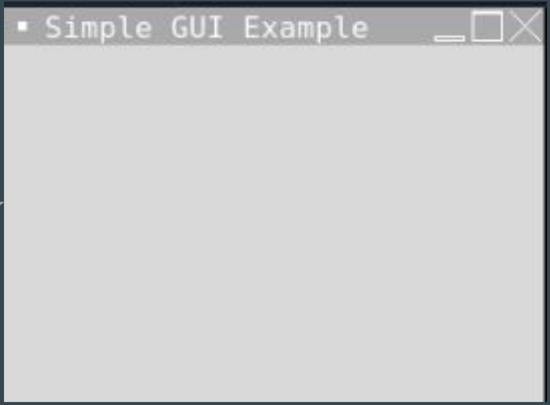
```
def update_label():
    name = entry.get()
    label.config(text=f"Hello, {name}!")
```

This function retrieves the text from the entry widget and updates the label widget with a greeting.

3. Create the main window:

- tk.Tk() creates the main window.
- root.title("Simple GUI Example") sets the title of the window.
- root.geometry("300x200") sets the size of the window.

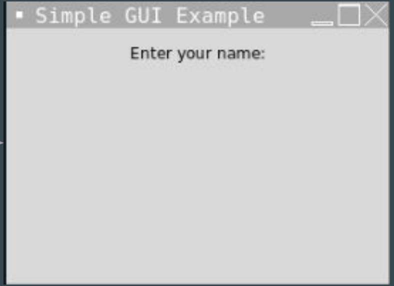
```
root = tk.Tk()
root.title("Simple GUI Example")
root.geometry("300x200")
```



4. Create and pack the label widget:

```
label = tk.Label(root, text="Enter your name:")
label.pack(pady=10)
```

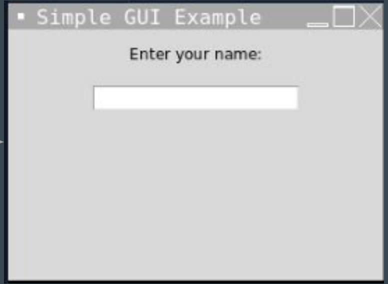
- tk.Label(root, text="Enter your name:") creates a label widget with the specified text.
- label.pack(pady=10) adds the label to the window with some padding.



5. Create and pack the entry widget:

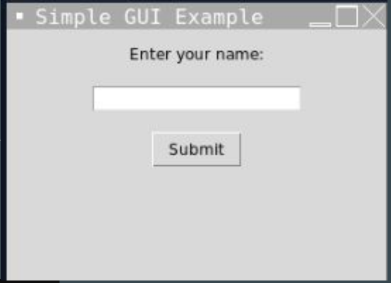
```
entry = tk.Entry(root)
entry.pack(pady=5)
```

- tk.Entry(root) creates an entry widget for text input.
- entry.pack(pady=5) adds the entry widget to the window with some padding.



6. Create and pack the button widget:

- tk.Button(root, text="Submit", command=update_label) creates a button widget that calls the update_label function when clicked.



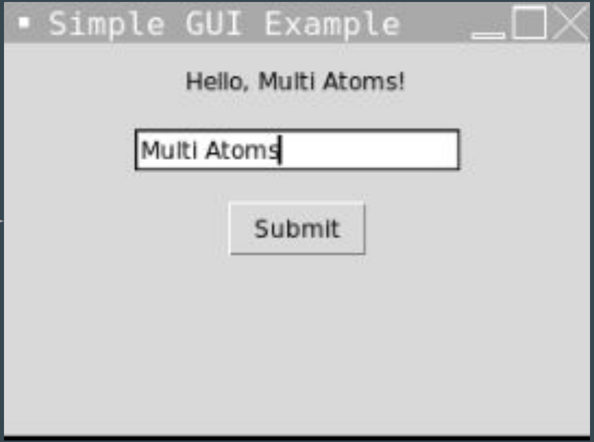
- button.pack(pady=10) adds the button to the window with some padding.

```
button = tk.Button(root, text="Submit", command=update_label)
button.pack(pady=10)
```

7. Run the main event loop:

```
root.mainloop()
```

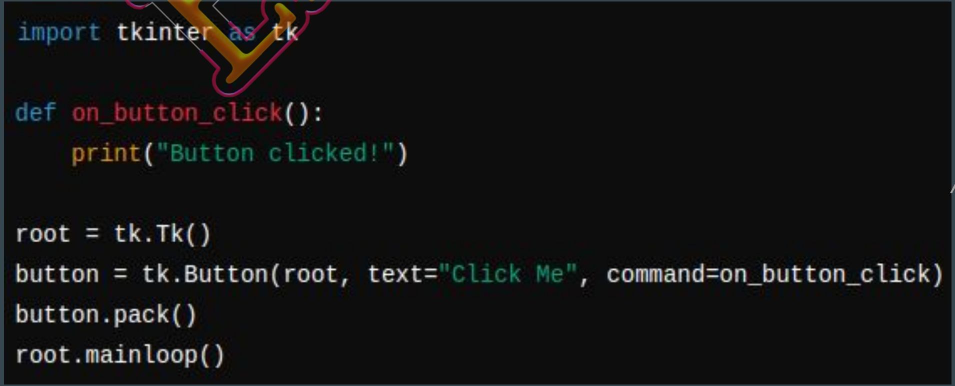
This starts the Tkinter event loop, which waits for user interactions and updates the GUI accordingly.



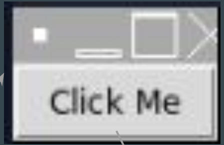
Common Tkinter Widgets with Examples and Definitions

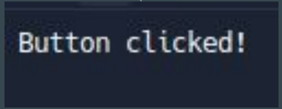
Tkinter provides a variety of widgets to create interactive GUI applications. Here are some of the common Tkinter widgets along with their definitions and examples:

1. **Button:** A Button widget is used to display a clickable button that can trigger a function or event when clicked.



```
import tkinter as tk

def on_button_click():
    print("Button clicked!")

root = tk.Tk()
button = tk.Button(root, text="Click Me", command=on_button_click)
button.pack()
root.mainloop()
```

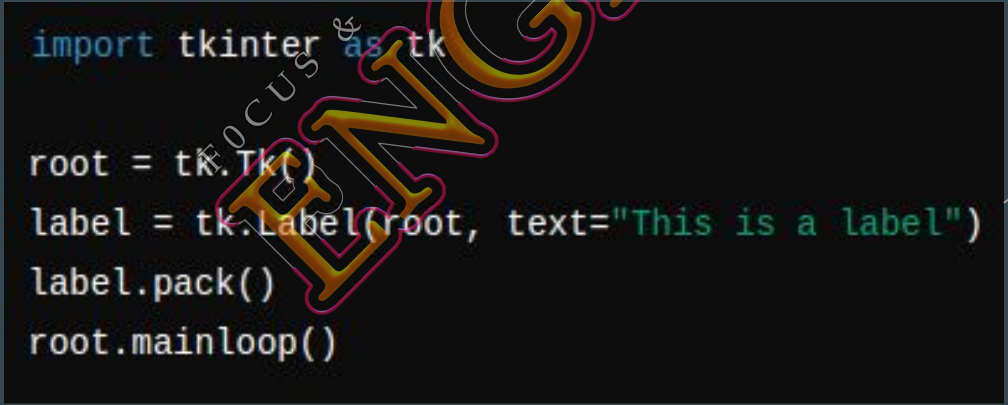




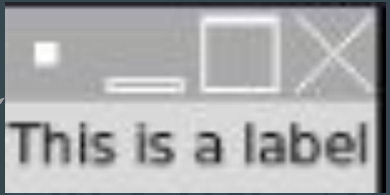
```
Button clicked!
```

- tk.Button(root, text="Click Me", command=on_button_click): Creates a button with the text "Click Me" that calls the on_button_click function when clicked.
- button.pack(): Adds the button to the window.

2. Label: A Label widget is used to display text or images on the screen.



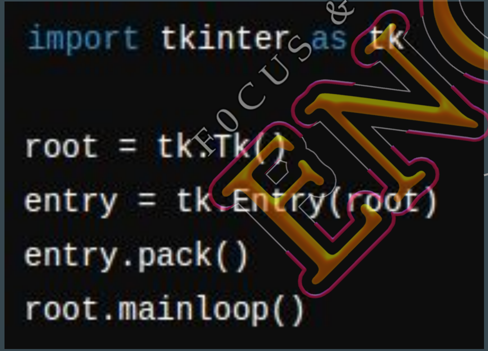
```
import tkinter as tk

root = tk.Tk()
label = tk.Label(root, text="This is a label")
label.pack()
root.mainloop()
```



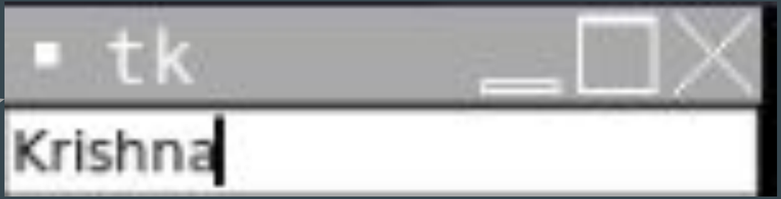
This is a label

- tk.Label(root, text="This is a label"): Creates a label with the text "This is a label".
- label.pack(): Adds the label to the window.

3. Entry: An Entry widget is used to create a single-line text input field.



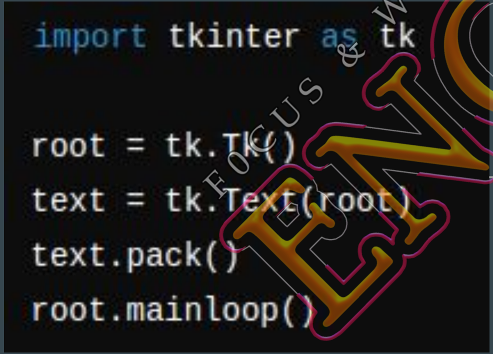
```
import tkinter as tk

root = tk.Tk()
entry = tk.Entry(root)
entry.pack()
root.mainloop()
```



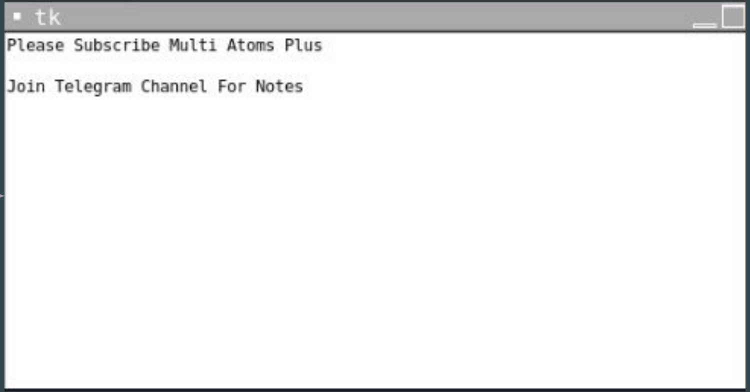
```
Krishna
```

- tk.Entry(root): Creates a single-line text input field.
- entry.pack(): Adds the entry field to the window.

4. Text: A Text widget is used to create a multi-line text input field.



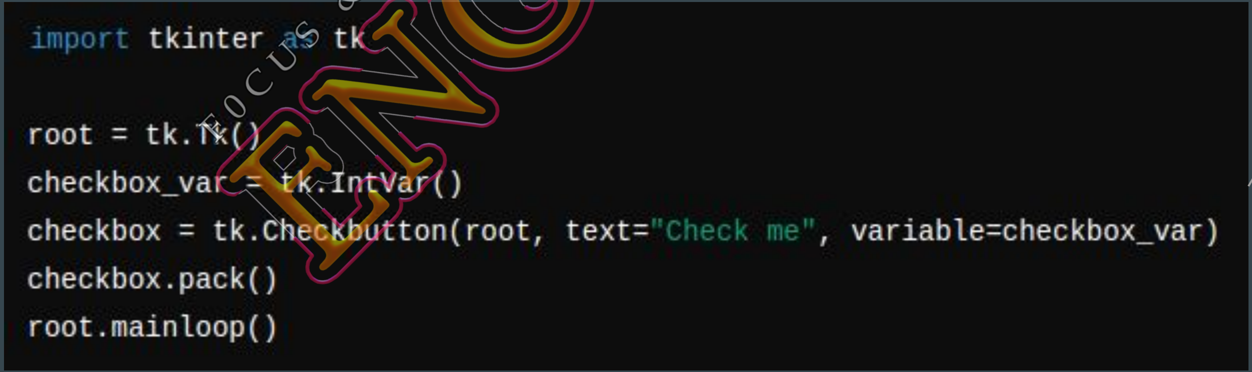
```python
import tkinter as tk

root = tk.Tk()
text = tk.Text(root)
text.pack()
root.mainloop()
```



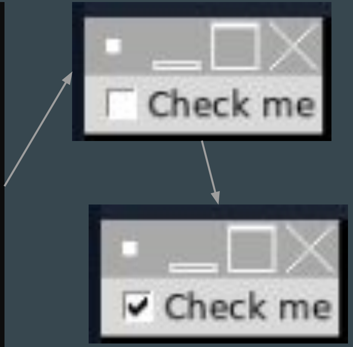
```
Please Subscribe Multi Atoms Plus

Join Telegram Channel For Notes
```

- tk.Text(root): Creates a multi-line text input field.
- text.pack(): Adds the text field to the window.

5. Checkbutton (Checkbox): A Checkbutton widget is used to create a checkbox that can be toggled on or off.



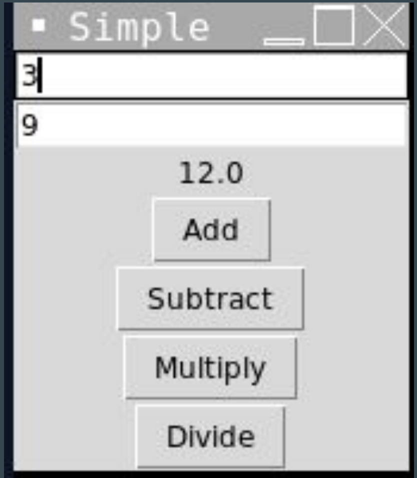
```
import tkinter as tk

root = tk.Tk()
checkbox_var = tk.IntVar()
checkbox = tk.Checkbutton(root, text="Check me", variable=checkbox_var)
checkbox.pack()
root.mainloop()
```



- checkbox_var = tk.IntVar(): Creates an integer variable to hold the state of the checkbox (1 for checked, 0 for unchecked).
- tk.Checkbutton(root, text="Check me", variable=checkbox_var): Creates a checkbox with the text "Check me" linked to the checkbox_var variable.
- checkbox.pack(): Adds the checkbox to the window.

Simple Calculator Using Tkinter

```
import tkinter as tk

def add():
    result.set(float(entry1.get()) + float(entry2.get()))

def subtract():
    result.set(float(entry1.get()) - float(entry2.get()))

def multiply():
    result.set(float(entry1.get()) * float(entry2.get()))

def divide():
    result.set(float(entry1.get()) / float(entry2.get()))

root = tk.Tk()
root.title("Simple Calculator")

entry1 = tk.Entry(root)
entry1.pack()

entry2 = tk.Entry(root)
entry2.pack()

result = tk.DoubleVar()
result_label = tk.Label(root, textvariable=result)
result_label.pack()

add_button = tk.Button(root, text="Add", command=add)
add_button.pack()

subtract_button = tk.Button(root, text="Subtract",
command=subtract)
subtract_button.pack()

multiply_button = tk.Button(root, text="Multiply",
command=multiply)
multiply_button.pack()

divide_button = tk.Button(root, text="Divide",
command=divide)
divide_button.pack()

root.mainloop()
```



1. Import Tkinter Module:

```python
import tkinter as tk
```

2. Define Functions:

```python
def add():
    result.set(float(entry1.get()) + float(entry2.get()))
```

This function retrieves the values from entry1 and entry2, adds them, and sets the result in the result variable.

```python
def subtract():
    result.set(float(entry1.get()) - float(entry2.get()))
```

```python
def multiply():
    result.set(float(entry1.get()) * float(entry2.get()))
```

```python
def divide():
    result.set(float(entry1.get()) / float(entry2.get()))
```

3. Create Main Window:

```python
root = tk.Tk()
root.title("Simple Calculator")
```

This creates the main window of the application and sets its title to "Simple Calculator".

4. Create Entry Widgets:

```python
entry1 = tk.Entry(root)
entry1.pack()
```

```python
entry2 = tk.Entry(root)
entry2.pack()
```

Creates the second entry widget for user input and adds it to the window.

5. Create Variable for Result:

```python
result = tk.DoubleVar()
```

This creates a Tkinter DoubleVar to hold the result of the calculations.

6. Create Result Label:

```python
result_label = tk.Label(root, textvariable=result)
result_label.pack()
```

This creates a label that displays the result of the calculation. The label's text is bound to the result variable.

7. Create Buttons:

```
add_button = tk.Button(root, text="Add", command=add)
add_button.pack()
```

```
subtract_button = tk.Button(root, text="Subtract", command=subtract)
subtract_button.pack()
```

```
multiply_button = tk.Button(root, text="Multiply", command=multiply)
multiply_button.pack()
```

```
divide_button = tk.Button(root, text="Divide", command=divide)
divide_button.pack()
```

8. Run the Main Event Loop:

```
root.mainloop()
```

This starts the Tkinter event loop, which waits for user interactions and updates the GUI accordingly.

Python Programming with IDE (Integrated Development Environment)

An Integrated Development Environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE typically includes a source code editor, build automation tools, and a debugger. Here are some popular IDEs for Python programming and their key features:

1. PyCharm: Developed by JetBrains, PyCharm is a powerful and widely-used IDE for Python. It comes in two versions: Community (free) and Professional (paid).

- **Intelligent Code Editor:** Code completion, real-time error checking, and quick fixes.
- **Debugging and Testing:** Integrated debugger and test runner.
- **Version Control:** Supports Git, SVN and more.
- **Web Development:** Supports Django, Flask, and other web frameworks.

2. Visual Studio Code (VS Code): A lightweight, open-source code editor developed by Microsoft, with extensive Python support through extensions.

- **Extensible:** Rich ecosystem of extensions, including Python support.
- **Debugging:** Built-in debugger.
- **Integrated Terminal:** Access to the terminal within the editor.
- **Version Control:** Built-in Git support.

3. Spyder: An open-source IDE specifically designed for scientific computing and data analysis, often used with Anaconda distribution.

- **Integrated IPython Console:** Enhanced interactive Python shell.
- **Editor:** Syntax highlighting, code completion, and introspection.
- **Documentation Viewer:** Built-in help system.
- **Plotting:** Inline plotting with Matplotlib support.

2. Visual Studio Code (VS Code): A lightweight, open-source code editor developed by Microsoft, with extensive Python support through extensions.

- **Extensible:** Rich ecosystem of extensions, including Python support.
- **Debugging:** Built-in debugger.
- **Integrated Terminal:** Access to the terminal within the editor.
- **Version Control:** Built-in Git support.

3. Spyder: An open-source IDE specifically designed for scientific computing and data analysis, often used with Anaconda distribution.

- **Integrated IPython Console:** Enhanced interactive Python shell.
- **Editor:** Syntax highlighting, code completion, and introspection.
- **Documentation Viewer:** Built-in help system.
- **Plotting:** Inline plotting with Matplotlib support.

IDEs that support Python development

- PyCharm
- Visual Studio Code
- Spyder
- Jupyter Notebook
- Atom
- Sublime Text
- Eclipse with PyDev
- Thonny
- Wing IDE
- Komodo IDE
- Eric Python IDE
- Rodeo
- Anaconda Navigator
- IDLE
- Geany
- NetBeans with Python Plugin
- Pyzo
- Bluefish
- Emacs with Python Mode
- IntelliJ IDEA with Python Plugin

Thank You

Please Subscribe Multi Atoms & Multi Atoms Plus
Join Telegram Channel for Notes
All the Best for your Exams

Multi Atoms Plus

THNX FOR VISITING

DEEPAK YADAV

ENGINEERING COLLEGE HUB

FOCUS & WIN

"You can be discouraged by failure or you can learn from it. So go ahead and make mistakes. Make all you can because that's where you will find success, on the other side of failure."