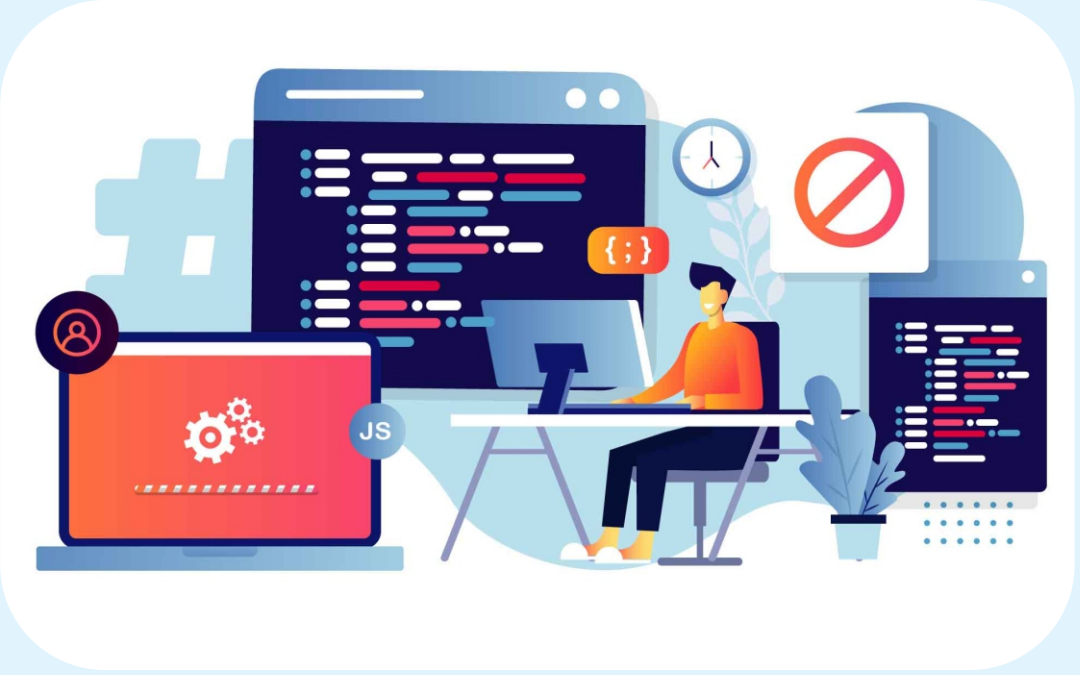
MATS UNIVERSITY

NAAC GRADE A+ ACCREDITED UNIVERSITY

MATS CENTRE FOR OPEN & DISTANCE EDUCATION

Python Programing

Bachelor of Computer Applications (BCA)
Semester - 3



SELF LEARNING MATERIAL

MATS UNIVERSITY
www.matsuniversity.ac.in

NAAC GRADE A+ ACCREDITED UNIVERSITY

Bachelor of Computer Applications

ODL BCA SEC 003

Python Programing

COURSE DEVELOPMENT EXPERT COMMITTEE

Prof. (Dr.) K. P. Yadav, Vice Chancellor, MATS University, Raipur, Chhattisgarh

Prof. (Dr.) Omprakash Chandrakar, Professor and Head, School of Information Technology, MATS University, Raipur, Chhattisgarh

Prof. (Dr.) Sanjay Kumar, Professor and Dean, Pt. Ravishankar Shukla University, Raipur, Chhattisgarh

Prof. (Dr.) Jatinder kumar R. Saini, Professor and Director, Symbiosis Institute of Computer Studies and Research, Pune

Dr. Ronak Panchal, Senior Data Scientist, Cognizant, Mumbai

Mr. Saurabh Chandrakar, Senior Software Engineer, Oracle Corporation, Hyderabad

COURSECOORDINATOR

Prof. (Dr.) K. P. Yadav, Vice Chancellor, School of Information Technology, MATS University, Raipur, Chhattisgarh

COURSE PREPARATION

Prof. (Dr.) K. P. Yadav, Vice Chancellor, School of Information Technology, MATS University, Raipur, Chhattisgarh

March, 2025

ISBN: 978-81-986955-3-6

@MATS Centre for Distance and Online Education, MATS University, Village- Gullu, Aarang, Raipur-(Chhattisgarh)

Printed &Published on behalf of MATS University, Village-Gullu, Aarang, Raipur by Mr. MeghanadhuduKatabathuni, Facilities & Operations, MATS University, Raipur (C.G.)

Printed at: The Digital Press, Krishna Complex, Raipur-492001(Chhattisgarh)

Acknowledgement

The material (pictures and passages) we have used is purely for educational purposes. Every effort has been made to trace the copyright holders of material reproduced in this book. Should any infringement have occurred, the publishers and editors apologize and will be pleased to make the necessary corrections in future editions of this book.

COURSE INTRODUCTION

Python is a versatile and widely used programming language, essential for modern software development, data analysis, and automation. This course provides a comprehensive introduction to Python programming, covering fundamental syntax, data handling techniques, and graphical user interface (GUI) development. Students will gain both theoretical knowledge and practical skills, enabling them to build efficient programs, manage data, and create interactive applications.

Module 1: Python Basics

This Module introduces the core concepts of Python programming, including syntax, variables, data types, and operators. Students will learn how to write basic programs, use control structures (if-else, loops), and implement functions. Understanding Python fundamentals is key to building efficient and scalable programs.

Module 2: Data Handling & Libraries

Data handling is essential for managing and analyzing information in Python. This Module covers data structures such as lists, tuples, dictionaries, and sets. Students will also learn to work with popular libraries like NumPy and Pandas for data manipulation, visualization, and analysis, making them proficient in handling large datasets.

Module 3: Database and GUI

This Module focuses on integrating Python with databases and building graphical user interfaces (GUIs). Students will learn how to connect to databases using SQL, perform CRUD operations, and develop interactive applications using libraries like Tkinter or PyQt. By mastering database and GUI concepts, students will be able to create functional, user-friendly software solutions.

Module 1
PYTHON BASICS

1.0 LEARNING OUTCOMES

- Understand Python syntax and basic operations.
- Learn about variables and different data types (Numbers, Strings, Lists, Tuples, Sets, and Dictionaries).
- Understand functions (built-in and user-defined) and explore function arguments and recursion.
- Learn about exception handling using try-except-finally blocks.
- Understand the init constructor method in Python classes.
- Learn about file handling (reading and writing files, file modes, and using the with statement).

Unit 1: Basic of Python

1.1 Python Syntax and Basic Operations

Python's ease of use, being readable and adaptable, it is among the most widely used programming languages. With its substantial indentation, Code readability is given first priority in this general-purpose, interpreted, high-level programming language. Procedural, Programming paradigms that include object-oriented and functional programming are Python supports. Python is a good language for both novice and seasoned programmers to learn because of its comparatively basic and easy-to-learn grammar. Python does not utilize curly brackets to limit code blocks or semicolons to finish statements like C or Java do. Rather, the program structure is specified using indentation, which makes it clearer and less syntactically noisy.Programs in the python involve statements and expressions to perform specific tasks. A Python script usually starts with a shebang line On Unix-based systems, this would be placed in the first line of code: #!/usr/bin/env python3 so that it can be run directly from Terminal, along with import statements if any external libraries are required Python comments are denoted by the hash tag (#), making it simple for the developers to annotate their code. Python variables are dynamically typed, meaning that a variable's type is deduced during runtime), allowing to easily write your code but also careful handling to avoid type-related errors.One of the basic operations in Python is variable assignment and basic sorts of data. Numerous data kinds are pre-installed in Python, such as textual, Boolean, integer, and floating-point values. In Python, variables are defined with the equal sign (=), and you do not need to declare variables like in some statically typed languages. For example, we just have to write x = to give a variable an integer value, use 10; for a floating-point number, use y = 3.14. Strings: A string with single or double quotations around it ('Hello'), is another fundamental data type. or "World"), and Python has powerful string manipulation and slicing and concatenation and built-in methods for it.You also have basic arithmetic operations available, so Python allows you to add (+), subtract (-), multiply (), and divide (/). The percent sign (%) gives the modulo or remainder of a division, while double asterisks (*) yield an exponent. Double forward slash (//) represents integer division that

yields the quotient while discarding the decimal part. Overall, these three basic arithmetic operations are the building blocks of all numerical computations in Python which makes it a great choice for both mathematical and scientific applications.

Program Example 1: Division, Multiplication, Subtraction, and Addition

```
# Python application to illustrate mathematical procedures
a = 15
b = 4
print("Addition:", a + b)
a - b, print("Subtraction:")
print(a * b), "Multiplication:"
print(a / b), "Division:"
print(a // b), "Integer Division:"
print("Modulus:", a % b)
print("Exponentiation:", a ** b)
```

Control flow statements are another key feature of Python. If, elif, and else conditional expressions enable developers to use the condition to determine whether to run a specific section of code. In Python, we use indentation to create these code blocks and make the syntax clean and readable. Control flow statements to generate complex conditions, employ operators like and, or, and not in logic.

Python Basic Syntax and Operations

Python is a powerful and readable high-level programming language. Stated information because readabi-lit-y counts more in python as compared to any other language which made it popular. Web development, data science, and artificial intelligence are just a few of the domains that use Python, a flexible programming language, automation, etc. Before you start programming, you need to learn about Python syntax and how to carry out basic operations. Where This Module Will Introduce Basics like Variables, Operators, if statements, loops, functions, data types, and basic input/output.

Python Syntax

Python syntax refers to the rules that define which combinations of symbols are considered to be properly formed programs in the Python programming language. Instead of using semicolons or curly braces for code blocks like in other programming languages, Python

interactively uses indentation. This allows for cleaner, more organized code in Python.

Python code: Writing and Running

Python code could be written in interactive mode through Python interpreter or script mode through file with an extension of a. py. To execute a Python script, use the python filename command. Python in Terminal/Command Prompt.

print("Hello, World!")

We use the print() function to show the output on the screen. This makes Python less strict as you need not to declare a variable in advanced.

1.2 Data Types and Variables (Numbers, Strings, Lists, Tuples, Sets, Dictionaries)

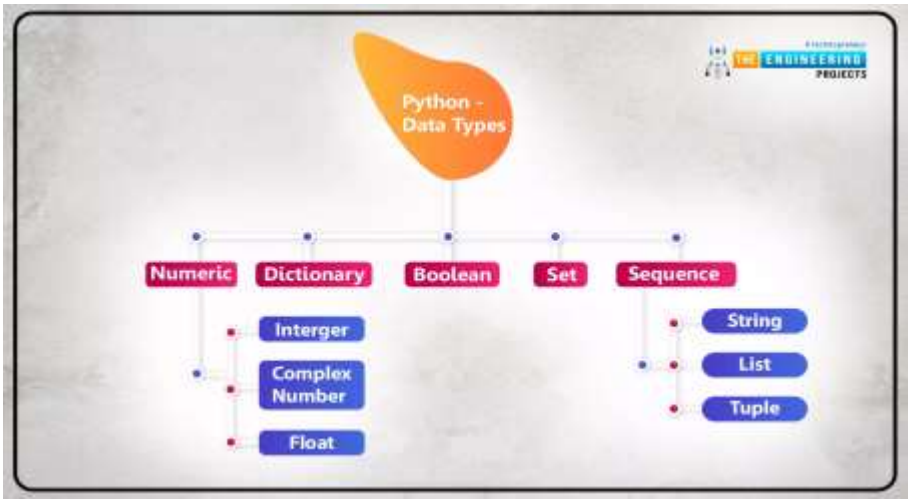


Fig: 1.1 Data Types

[Source: https://images.theengineeringprojects.com]

Python's = operator is used to give a variable values. Python variables do not need to define a type, unlike statically typed languages. The assigned value determines the kind of variable.

x = 10 # Integer

y = 3.14 # Float

Name = String # "Python"

is_active = Boolean #True

Numerous built-in data types are supported by Python, including:

- Entire numbers, such as 10, -5, and 100, are known as integers (int).
- Numbers having decimal points, such as 3.14 and -2.7, are known as floating-point numbers (float).
- "Hello" and "World" are examples of strings (str), which are a series of characters encapsulated in single or double quotations.
- Boolean (bool): Stands for True or False
- Lists (List): arranged grouping of values, such as
- Immutable ordered collections, such as (1, 2, 3) are called tuples.
- Key-value pairs found in dictionaries (dict) include {"name": "John" and "age": 25}.

Operators in Python

Python has various operators to do computation and logical operations. Operators are divided into Bitwise operators, membership, identity, assignment, comparison, arithmetic, and logical.

Arithmetic Operators

Basic mathematical operations are carried out via arithmetic operators:

```
a = 10
b = 5
print(a + b)  # Addition
print(a - b)  # Subtraction
print(a * b)  # Multiplication
print(a / b)  # Division
print(a % b)  # Modulus
print(a ** b)  # Exponentiation
print(a // b)  # Floor division
```

Comparison Operators

Comparison operators compare two values and return a Boolean result:

```
x = 10
y = 20
print(x == y)  # False
print(x != y)  # True
print(x >y)   # False
```

```
print(x <y)   # True
print(x >= y)  # False
print(x <= y)  # True
```

Logical Operators

Logical operators are used to combine multiple conditions:

```
x = True
y = False
print(x and y)  # False
print(x or y)   # True
print(not x)    # False
```

Control Flow Statements

Python contains control flow statements, such as loops (for, while) and conditional phrases (if-elif-else), to govern how programs are executed.

Statements with Conditions

```
num = 10
if num> 0:
print("Positive number")
elifnum< 0:
print("Negative number")
else:
   print("Zero")
```

Loops in Python

Using loops, a block of code can be run repeatedly.

for Loop

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

while Loop

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

Functions in Python

Functions are reusable code segments that carry out particular tasks.

```
def greet(name):
   return "Hello, " + name
```

```
print(greet("Alice"))
```

Python also supports lambda functions, which are anonymous functions:

```
square = lambda x: x * x
print(square(5))
```

Input and Output Operations

Python has print() to display output and input() to accept user input.

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

Working with Lists

Lists are mutable sequences used to store multiple items.

```
numbers = [1, 2, 3, 4, 5]
numbers.append(6)
print(numbers)
```

Working with Dictionaries

Dictionaries store key-value pairs and allow efficient data retrieval.

```
person = {"name": "John", "age": 30}
print(person["name"])
```

File Handling

Python's built-in functions enable reading and writing files.

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

Exception Handling

Python provides exception handling to manage errors gracefully.

```
try:
num = int(input("Enter a number: "))
print(10 / num)
except ZeroDivisionError:
print("Cannot divide by zero")
except ValueError:
print("Invalid input")
```

Python is an excellent option for novices due to its easy-to-understand syntax. We covered a few fundamental ideas in this Module, including syntax, variables, operators, and control structures, functions, and file handling. Once you grasp these fundamentals, you can dive deeper into Python for data analysis, machine learning, and web development. Once you have a good command of such basics

you will be in a better position to dive deep into advanced programming and real-world applications.

Sample Program: Conditional Statements

```
# A program that determines if a given number is positive, negative, or zero
num = int(input("Enter a number: "))

if num> 0:
print("The number is positive.")
elifnum< 0:
print("The number is negative.")
else:
print("The number is zero.")
```

Learning about loops is another requirement for Python programming. To loop, utilize the for and while loops. over sequences and perform various tasks repeatedly. Loop, the for loop is one of the most utilized when having to iterate through lists, tuples, strings, and ranges (or something iterable in general), The while loop iterates while a certain condition is true. The loop can be controlled with the help of the break and continue statements_exec, exiting or jumping past iterations when needed.

Sample Program: Looping Structures

```
# Write a looping program that prints numbers one through ten.
for i in range(1, 11):
print(i, end=' ')
print("\n")
# Using while loop
count = 1
while count <= 10:
print(count, end=' ')
   count += 1
```

You also have in Python collections as dictionaries, sets, tuples, and lists. Lists: Sequences that can be changed to store collections. They allow different operations like indexing, slicing, append, remove etc. Intercept Note, this is one make up and tuple tuple other way this are immutable, them nothing change after assignment. While dictionaries are unordered key-value pairs that enable unique element collections, sets are high-performance data retrieval.

Sample Program: Working with Lists

```
# Creating and manipulating a list
fruits = ['Apple', 'Banana', 'Cherry']
fruits.append('Mango')
print("List of Fruits:", fruits)
print("First Fruit:", fruits[0])
fruits.remove('Banana')
print("Updated List:", fruits)
```

Python features enable you to divide the more complex issue into smaller, more manageable ones, smaller issues. The def keyword is used to define functions; the function name comes next, and parenthesis. They accept disagreements and come back values to make code more flexible.

Sample Program: Defining Functions

```
# Function to determine a number's square
def square(num):
    return num ** 2

print("Square of 5:", square(5))
```

Other crucial subjects include object-oriented programming, file handling, and handling of exceptions. In Python, the try, except, and finally blocks are utilized to detect and manage runtime issues. Reading and writing to files are both part of file handling and OOP introduces concepts like classes, objects, and inheritance (open(), read(), write()).

Sample Program: Exception Handling

```
try:
    x = int(input("Enter a number: "))
    y = int(input("Enter another number: "))
    result = x / y
print("Result:", result)
except ZeroDivisionError:
print("Error: Cannot divide by zero!")
except ValueError:
print("Error: Invalid input! Please enter a number.")
```

Python is more efficient because of its built-in modules and libraries. For example, datetime for date and time, random for random integers and the math module for mathematical functions operations.

Sample Program: Using Built-in Modules

```
import math
import random

print("Square Root of 16:", math.sqrt(16))
print("Random Number between 1 and 10:", random.randint(1, 10))
```

All in all, tech stack for Python development has been chosen in the order of ease to medium-low jobs. Developers can use abstracted syntax and operations by following the fundamentals of Python, allowing them to create more efficient, scalable applications.

Programming Basics: Data Types and Variables

Among the essential components of any programming language is the variable, which enables data manipulation and storage. A variable is merely the name of a place where things can be kept in memory while your program runs. As you can imagine, allowing a program to have variables is an essential part of having it do complex computations, process data, and display useful output. As such, mastering it allows one to handle and manipulate the data as efficiently and effectively as possible, which is paramount to understand in the world of programming. (The exact rules governing variable declaration, assignment, and scope vary from language to language, but the idea is the same: variables hold target values that you can reference and change while the program is running). (For instance, Python variables are typed dynamically, meaning that the type of value you assign to a variable determines its data type, negating the need for explicit declaration itself.)In programming, data types are also an essential concept, as they define whatA many types of information can be kept in a variable. Types of data vary by computer language (numbers, strings, lists, tuples, sets, dictionaries, etc.) on which various operations can be applied and various storage mechanisms. Data Types (the classification) allow a program to correctly compute when handling its part of the data; thus maintaining the integrity and correctness of the information. By being aware of the various kinds of data, programmers can effectively plan how memory will be allocated for various data types, improve performance, and only use the operations that make sense for each type of data. From here, let us go deeper into all of these data types, their features and functions, that can be practically appliedOut of other data types in any

programming numbers are one of the most widely used. They consist of numerical values that include complicated numbers, floating-point numbers, and integers. For example, in Python, floating points (float) have a decimal point, whereas integers (int) are entire numbers without 1. As mentioned earlier, complex numbers are made up of of both real and imaginary components. Arithmetic operations like as addition, subtraction, division and multiplication are frequently carried out using numerical data types. This is the program demonstrating some of the numerical operations:

 # Sample Program 1: Working with Numbers

```
x = 10   # Integer
y = 3.5  # Float
z = x + y  # Adding an integer and float
print("The sum is:", z)
```

Another basic data type is strings, which are character sequences that are surrounded by single, double, or triple quotations. Frequently employed for text data manipulation and storage. These built-in methods include concatenation, slicing, repetition, et cetera .upper(),.lower(), and. replace().Once assigned, the contents of strings cannot be changed since they are unchangeable. This Program Demonstrated Some OfThe String Functions:

 # Sample Program 2: String Manipulation

```
text = "Hello, World!"
print(text.higher()) # Convert to capital letters
text.lower() to print #  Change to lowercase
print(text[0:5])      # Slicing strings
```

One of the most versatile data structures is undoubtedly a list. Because they enable storing multiple values in a single variable. Lists, which are variable-length, meaning elements can be changed, appended, or deleted. As a sequence, lists support indexing, iteration, and many built-in functions. append(),.remove(), and. sort(). The next program illustrates how to manipulate lists:

 # Sample Program 3: List Operations

```
digits = [1, 2, 3, 4, 5].
numbers.append (6)   # Including a component
print(numbers)
numbers.remove(3)   # Removing an element
```

```
print(numbers)
```

Tuples have some similarities but are immutable, which means that once they are created, none of their constituent parts can be altered. They come in handy when dealing with fixed items collection. Tuples are similar in that they use parentheses instead of list square brackets, but you can access data from tuples in an efficient way. The next program shows how to use tuple:

 # Sample Program 4: Tuple Usage

```
tuple_data = (10, 20, 30)
print(tuple_data)
print(tuple_data[1])  # Accessing tuple elements
```

Sets are collections of non-ordered values and they can not contain duplicates. They are appropriate for operations like membership testing, union, intersection, and difference. This program illustrates operations with set:

 # Sample Program 5: Set Operations

```
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}
print(set1.union(set2)        #Union of sets
print(intersection of set1. (set2)  # Sets intersecting
```

Dictionaries are hash tables; the principle behind dictionaries is storing key-value pairs that allow values to be quickly retrieved using unique keys. They are incredibly adaptable and widely used for mapping relationships among data elements. In the following program, we work with dictionary operations:

 # Sample Program 6: Dictionary Operations

```
data = {"name": "Alice", "age": 25, "city": "New York"}
print(data["name"])  # Accessing a value using a key
data["age"] = 26      # Modifying a value
print(data)
```

So this is essential to understand the functionality and data types for programming. The types of data are used for different properties for its use cases. These data types allow programmers to write code that is efficient and scalable while ensuring optimal data management and computational efficiency. Moreover, the combination of these data types allows for the creation of advanced data structures, enabling advanced problem solving and algorithm development.In conclusion, variables and data types are essential programming ideas that let

programmers make ever-more complex, dynamic, and interactive applications. This stuff helps enhance problem-solving skills, remove any roadblocks and prepares the ground for the next learnings on various levels, For instance, data structures, algorithms, object-oriented programming, etc. Regular experimentation with data input enables programmers to build their knowledge base, which ultimately helps them write optimized and scalable code.

Unit 2 Function in Python

1.3 Functions: Built-in and User-defined, Function Arguments, Recursion

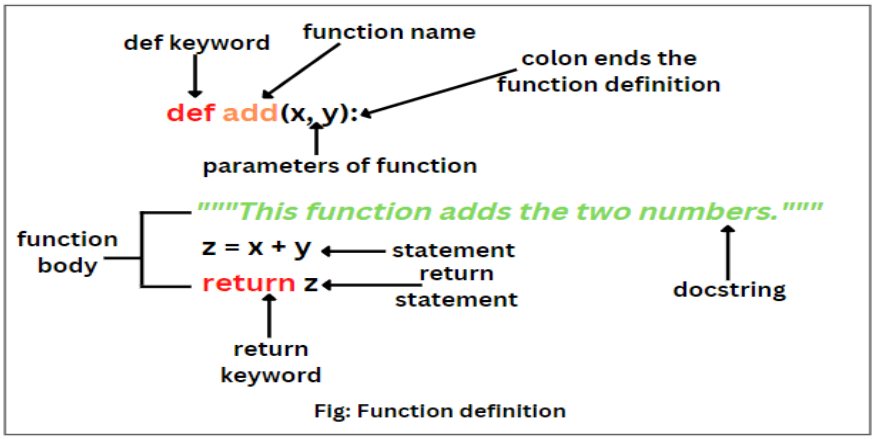


Fig: 2.1 Function in Python
[Source: https://www.scientecheasy.com]

We are only focused on functions here, that are an important aspect of programming as it will help you to write modular code, re-usable, and structured codeA function is a section of code that can be used repeatedly by software. to accomplish a particular goal. By going over functions, we can divide complex problems into smaller ones to solve. Improves the readability and maintainability of programs, they also let you debug programs faster. Functions can be broadly of two types in most programming languages both user-defined and built-in functions. Functions also have disagreements, so they can take in data, and recursion is a specific, special case of functions calling itself.

Built-in Functions

hese pre-made functions are included in the programming language to perform common tasks. It is extremely fast and it's available to use out of the box. This covers the built-in functions, which include things like math operations, string manipulation, input/output, etc. A method also known in some programming languages function, contains several commands, so that we do not have to rewrite the same code over and over, and, therefore, makes coding much easier

(an example are built-in functions in Python, including print(), len(), sum(), max(), and others.SqlAlchemy makes many functions written to make developer life easier).

Consider the following example where we use Python built-in functions:

 # Example 1: Using Integrated Features

```
Numbers = [4, 7, 1, 9, 3]
print("Maximum number:", max(numbers)) # Determines the highest number
print("Minimum number:", min(numbers))  # Finds the minimum number
print("Sum of numbers:", sum(numbers))  # Calculates the sum
print("Length of list:", len(numbers))  # Finds the length of the list
```

You don't need to bother about explicitly designing an algorithm to determine the maximum or minimum of a given value because these built-in functions do tasks for you effectively list or sum or length.

Functions defined by the user

With the exception of built-in functions and user-created functions, specify a certain feature that isn't provided by built-in functions. A function definition organizes the code better and makes coding reusable. The def keyword, the function name, and two parentheses which may contain parameters define a function in Python. The function's body outlines the actual operations, and when called, the function runs.

A user-defined function example is this one, which calculates a number's square:

 # Example 2: User-defined Function

```
def square(num):
   return num * num

outcome = square (5)
# Output: 25 print("Square of 5:", result)
```

The function square() accepts an integer in this example and returns that number squared. Code can be reused (using that function as many times as we want with different values) which makes code organized and avoids code duplicity.

Function Arguments

Definition: These are the values supplied to the function when it is called. The function's arguments can be positional, keyword, default, or variable-length.

- As The sequence in which arguments are presented, positional arguments are the most prevalent kind matters.

 # Example 3: Positional Arguments

```
def welcome (name, age):
print(f"Hello {name}, you are {age} years old.")

greet("Alice", 25)
```

- Keyword Arguments: In this kind, arguments are conveyed according to the names of their respective parameters, which make the function call more informative and independent of order.

 # Example 4: Keyword Arguments

```
greet(name="Bob", age=30)
```

- Default Arguments: A parameter can be assigned a default value, and in the event that no arguments are supplied, this value will be used.

 # Example 5: Default Arguments

```
def power(base, exponent=2):
   return base ** exponent

print(power(4))  # Uses default exponent value 2
print(power(4, 3))  # Uses provided exponent value 3
```

- Important Topics in Python Functions: Variable-length Arguments: Sometimes a function needs to agree with any number of arguments. For non-keyword arguments, use args; for keyword arguments, use *kwargs.

 # Example 6: Variable-length Arguments

```
def add_numbers(*args):
    return sum(args)

print(add_numbers(1, 2, 3, 4, 5))  # Output: 15
```

Recursion in Functions

A key idea in programming is recursion, which is when a function calls itself to solve an issue. Factorial, Fibonacci series, tree traversals are examples of problems that can be solved through their subproblems using those.

- Factorial Using Recursion:

 # Example 7: Factorial using Recursion

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

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

Here, until the base case (n == 0 or n == 1) is reached, the factorial () method calls itself with a smaller result.

- Fibonacci Series Using Recursion:

 # Example 8: Fibonacci Series using Recursion

```
def fibonacci(n):
    if n <= 0:
        return "Invalid input"
elif n == 1:
        return 0
elif n == 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(6))  # Output: 5
```

- Sum of Digits Using Recursion:

 # Example 9: Sum of Digits using Recursion

```
def sum_of_digits(n):
    if n == 0:
        return 0
    else:
        return (n % 10) + sum_of_digits(n // 10)

print(sum_of_digits(1234))  # Output: 10
```

- Reverse a String Using Recursion:

 # Example 10: Reverse a String using Recursion

```
def reverse_string(s):
    if len(s) == 0:
        return s
    else:
        return s[-1] + reverse_string(s[:-1])

print(reverse_string("hello"))  # Output: "olleh"
```

Recursion is elegant, but excessive function calls in recursion can eat a lot of memory. In some cases, you might want to use iterative solutions for optimal performance.

You can analyse the list of functions with a human-like approach. While A user-defined function enables the user to create their own, whereas a built-in function is a basic function that already exists functions easy to use and simplifies the most common activities. programmer to customize an operation according to the needs of convert. Because function arguments allow flexibility, because functions are more generalized. Functions can call themselves, a trait known as recursion, which has proven exceptionally useful in solving complex problems. Recursive Function Implementation Considerations although recursive functions RMID obviously are very useful, they have the disadvantage that they can lead to the accumulation of data in the memory stack and cause stack overflow errors. Learning about functions and how to use them properly allows programmers to develop efficient, scalable, and maintainable code in a variety of fields and applications.So it would be best if you were a

little bit careful when working with exceptions, especially those that are not handled properly; thus, try-except-finally helps solve this problem.What is Exception Handling in Python? Any programming language comes with its own set of errors, and Python comes with a massive mechanism to deal with them using the try-except-finally constructs. This mechanism enables developers to handle exceptions, which prevents programs from being terminated abruptly. Learning the try except finally is key to writing reliable and error-proofed code.The try block contains the code that could result in an exception. There is one exception mentioned if something goes wrong in this block, and The corresponding except block captures it. The exception block states the action to take in case of certain exceptionsIf the code that might cause an exception is contained in the try block. Whether or not a finally block raises an exception is defined, it is always performed. Finally, this is very useful for cleanup tasks like database termination, resource release, and file closure connections.Python has built-in exceptions Zero Division Error, IndexError, KeyError, ValueError, TypeError, etc. There are certain other exceptions (1d, 1e) which happen when there is a problem with the thread's execution. Additionally adaptable, the try-except-finally block can catch several exceptions in a single block, which improves the robustness of error processing logic.

1.4 Exception handling (try-except-finally)

One of the most common exceptions in Python is Zero Division Error which occurs on division of a number by zero the program below shows how to catch this exception and handle it gracefully:

```
Try:
   numerator = int(input("Enter numerator: "))
   denominator = int(input("Enter denominator: "))
   result = numerator / denominator
print("Result:", result)
except ZeroDivisionError:
print("Error: Division by zero is not allowed.")
finally:
print("Execution completed.")
```

Example: Handling Multiple Exceptions

The same program can raise many exceptions in run-time. The code below catches both ZeroDivisionError and ValueError:

```
try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))
print("Result:", num1 / num2)
except ZeroDivisionError:
print("Cannot divide by zero.")
except ValueError:
print("Invalid input! Please enter a number.")
finally:
print("End of program.")
```

Example: Handling File Not Found Error

Handling FileNotFoundError Handling When Performing File Operations Here is the code which shows how to deal with this situation:

```
try:
    file = open("non_existent_file.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
print("Error: File not found.")
finally:
print("File operation attempted.")
```

Example: Handling Index Error

IndexError when accessing an invalid index in a list Here's code that shows how to catch this exception:

```
try:
my_list = [1, 2, 3]
    print(my_list[5])
except IndexError:
print("Error: Index out of range.")
finally:
print("List operation completed.")
```

Example: Handling Key Error in Dictionary

In SQL, a When you attempt You receive a KeyError when attempting to retrieve a dictionary key that is not present. This code takes care of the error quite well This article was published as a part of the Data Science Blogathon:

```
try:
```

```
my_dict = {"name": "John", "age": 30}
   print(my_dict["address"])
except KeyError:
print("Error: Key not found in dictionary.")
finally:
print("Dictionary operation completed.")
```

Example: Handling Type Error

A TypeError arises when an operation is conducted on incompatible types. Here is an example of this:

```
try:
   result = "5" + 10
except TypeError:
print("Error: Cannot add a string and an integer.")
finally:
print("Type checking completed.")
```

Example: Handling Value Error

When we provide AWhen a function receives an argument of the right type but an even more wrong value, a ValueError is triggered. The below example shows how to handle it:

```
try:
   number = int("abc")
except ValueError:
print("Error: Invalid conversion to integer.")
finally:
print("Value checking completed.")
```

Example: Handling Custom Exceptions

In some cases, developers have to define their own exceptions. Below is the code used to generate and manage a user-defined or custom exception:

```
CustomError(Exception) class:
   pass
try:
   raise CustomError("This is a custom exception.")
apart from CustomError as e:
print("Caught custom exception:", e)
finally:
print("Custom exception handling completed.")
```

Example: Using Else Clause with Try-Except

Python contains a clause that is optional, that, if no exception happens, will run. Here is an example of how it works:

```
try:
num = int(input("Enter a number: "))
print("Square:", num * num)
except ValueError:
print("Invalid input!")
else:
print("No exceptions occurred.")
finally:
print("Program execution completed.")
```

Example: Ensuring Resource Cleanup with Finally

A typical application for the finally block is resource management. An example of how to use it follows:

```
try:
   file = open("sample.txt", "w")
file.write("Hello, world!")
except IOError:
print("Error: File operation failed.")
finally:
file.close()
print("File closed successfully.")
```

These error-handling techniques are an essential component of Python programming that increases the stability and reliability of applications. This not only helps avoid the hard stop of their program, but also allows for better recovery from errors. This lesson went through several examples illustrating a large variety of real-world exceptions that can demonstrate the power of exception handling to write resilient code. Read Next; Correcting Common Python Error Handling Mistakes (TypeError, FileNotFoundError, and Custom Exception) whether it is handling integer value errors or input validation errors, the try-except-finally construct is one of the most useful tools in Python programming. Constructor Python has a special type of function that is automatically called upon the instantiation of a class object. Using this function init to define this method, to initialize the attributes and to set up any necessary elements for the object. The constructor is an important part about object oriented programming,

as it makes sure that all instances of a class start off in a good state.A constructor is always called init, and it accepts self as its first parameter, which we will know is the instance of the class. It is also possible to define additional parameters to pass in values and perform some processing when creating an object. This approach provides encapsulation and safeguards the data's integrity by prohibiting direct access to object characteristics from outside the class definition. By using constructors, programmers can make the process of creating objects with initial values easier and reusable, which can reduce redundancy in the code. Imagine this class Person that uses a constructor to set up a person's age and name. The constructor immediately launches Upon creating a Person class object, initializing the instance variable.

```
class member:
   __init__ def (self, name, age):
      self.name = name
self.age = age

   def display(self):
print(f"Name: {self.name}, Age: {self.age}")

person1 = Person("Alice", 25)
person1.display()
```

The characteristics of age and name are set to new values when the above constructor creates a new Person. Then, the details of the person are printed using the display method.

You can also use a constructor to establish the properties' default values. When making something, if no arguments passed, default values are given.

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

   def show_details(self):
print(f"Car Brand: {self.brand}, Model: {self.model}")

car1 = Car()
```

```
car2 = Car("Honda", "Civic")
car1.show_details()
car2.show_details()
```

In this example, the constructor sets default values for brand and model, so even if no values are passed, the created object contains meaningful data.

Constructors can also perform calculations and process data while initializing an object. This functionality is very useful for objects that must be preprocessed before use.

Rectangular class:

```
  (self, length, width) def __init__:
length = self.length
self.width = width
* width * length = self.area
    def display(self):
print(f"Length: {self.length}, Width: {self.width}, Area: {self.area}")

rect1 = Rectangle(10, 5)
rect1.display()
```

In the above example, the constructor determines the rectangle's area as soon as the object is formed. It provides the need for an additional area computation method into oblivion.

Inheritance also implements constructors, A super() function allows To call the parent class constructor, use a child class constructor.

Animal class:

```
(self, species) def __init__:
self.species = species

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__("Dog")
        self.name = name
self.breed = breed

    def show_info(self):
print(f"Species: {self.species}, Name: {self.name}, Breed:
{self.breed}")
dog1 = Dog("Buddy", "Golden Retriever")
```

dog1.show_info()

The super (). Call to init () guarantees that the constructor of the Animal class is called first, then that of the Dog class's unique attributes are initialized.

A key feature of constructors is their capability of efficient instantiation of multiple objects. Constructors can be incredibly powerful for large-scale applications as they can be dynamically created with various parameters.

student in the class:

function __init__(self, name, marks, student_id):

student_id = self.student_id

self.name = name

self.marks = marks

def details (self):

print(f"ID: {self.student_id}, Name: {self.name}, Marks: {self.marks}")

students = [

Student(101, "John", 85),

Student(102, "Emma", 90),

Student(103, "Liam", 78)

]

for student in students:

student.details()

This demonstrates the use of constructors in creating Student objects and the ability to instantiate multiple objects using constructors.

Constructors also are involved in encapsulation by being a way to limit direct access to class variables and to enable only access through getter and setter methods.

def specifics (self):

(self, account_number, balance) def __init__:

self.__account_number = account_number

self.__balance = balance

get_balance(self) def:

self.__balance back

```
   def deposit(amount, self):
      if amount > 0:
self.__balance += amount
print(f"Deposited: {amount}, New Balance: {self.__balance}")
      else:
print("Invalid deposit amount!")

account = BankAccount("123456", 5000)
print("Initial Balance:", account.get_balance())
account.deposit(1500)
```

Class variables are made with double underscores (__). private and limit their access through specific methods.

We can also use constructors in resource management such as file handling, in which a file is opened while the object is being initialized and closed once the object is no longer required.

FileHandler class:

```
   (self, filename, mode) def __init__:
self.file = open(mode, filename)
print(f"File '{filename}' opened in {mode} mode")

   (self, data) def write_data:
self.file.write(data)

close_file(self) def:
self.file.close()
print("File closed")

file1 = FileHandler("sample.txt", "w")
file1.write_data("Hello, this is a test file.")
file1.close_file()
```

The context manager ensures files are properly cleaned up, minimizing the chances of leaks.

At their core, constructors are a critical building block in building expressive Python applications. They play an important role in object-oriented programming by handling object initialization, encapsulation, inheritance, and resource management. Effective Using constructors can result in code that is simpler to maintain, cleaner, and improve code efficiency.Structure of Writing 8800 words in paragraph on File

Handling (Reading & Writing Files, File Modes, With Statement) with sample programs 10. File Handling in Python Explained with Examples: Below is a detailed explanation with examples that covers different concepts of Python's file handling.

1.5 Constructor (init method) in Python

Yo are to develop file handling techniques to both read and write data to a file that is external. The fcntl module allows you to control file descriptors, enabling you to perform operations such as locking, referencing or blocking on these descriptors. Python has in-built functions to handle files efficiently with different modes to specify how to Get a file open. Additionally, employing the with statement makes that the file is correctly handled and will not cause file corruptions or memory leaks.

Python's file handling follows the standard CRUD operations:

1. Create – Creating a new file.
2. Read – Reading data from an already existing file.
3. Update (Write/Append) – Updating data in the file.
4. Delete – Removing a file.

To perform any operation on a file, the file must be opened first using Python's built-in open() function, which accepts two inputs: the access mode and the file path. To release system resources, we should use close() to end the process of closing the file, however the with statement easily handles this.

Python File Opening

The open() method in Python is used to open a file. The file name is the only argument it takes. A mode for opening the file is an optional second input.

open("example. txt", "r") file # Starts the read mode of example.txt

There are several file modes in Python:

- **Read (r):** opens a file in the default reading mode.
- **Write (w):** This creates a new file and opens the file for writing if there is none, will truncate if there is.
- **Append (a):** Initializes the file in add mode while preserving the current content.
- **Read and Write key (r+):** To enable reading and writing of a file, press the Read and Write key (r+). to exist
- **Write and Read (w+):** Reads and writes, and creates a new file if the one doesn't already exist.

- **Append and Read (a+):** Available for both writing and reading and truncating.

1.6 File Handling (Reading & Writing Files, File Modes, With Statement)

Reading files is a fundamental operation. Python provides multiple ways to read files:

Example 1: Reading a Complete File

using the file open("sample.txt", "r"):

```
content = file.read()
print(content)
```

In this For example, the with statement opens the file in read mode is used.

(r), read everything from it and print it. The file gets shut down automatically by the statement after you finish reading it.

Example 2: Line-by-Line Reading

```
with open("sample.txt", "r") as file:
    for line in file:
        print(line.strip())  # Removes trailing newline characters
```

Here, the file is read line by line using a loop, which saves memory.

Example 3: Using readline() to Read a Single Line

```
with open("sample.txt", "r") as file:
    line = file.readline()
    while line:
        print(line.strip())
        line = file.readline()
```

The readline()technique is helpful when handling single lines at a time structured text.

Example 4: Reading into a List Using readlines()

```
with open("sample.txt", "r") as file:
    lines = file.readlines()
    print(lines)
```

All of the lines are read by the readlines() function to a list with a matching element for every line in the file.

Python Writing to a File

The w, a, or w+ mode must be used in order to write data to a file. If there isn't a file, it creates one created automatically.

Example 5: Writing to a File (w Mode)

```
with open("output.txt", "w") as file:
file.write("Hello, World!\n")
file.write("This is a new line.\n")
```

This code writes two lines into output.txt. If the file exists, its content is erased before writing new data.

Example 6: Appending to a File (a Mode)

```
with open("output.txt", "a") as file:
file.write("Appending a new line.\n")
```

This appends data without erasing the previous content.

Example 7: Writing Multiple Lines

```
lines = ["First Line\n", "Second Line\n", "Third Line\n"]
with open("multiline.txt", "w") as file:
file.writelines(lines)
```

The writelines() method writes a list of strings to a file.

File Handling with Different Modes

Python allows files to be opened in different modes based on the requirement.

Example 8: Reading and Writing (r+ Mode)

```
with open("data.txt", "r+") as file:
print(file.read())   #  Examine the current content.
file.write("\nAdding a new line!")  # Write at the end
```

The r+ mode does not truncate the file but permits reading and writing.

Example 9: Overwriting vs. Appending (w+ and a+ Modes)

```
# Overwriting the file
with open("overwrite.txt", "w+") as file:
file.write("Overwritten content.\n")
file.seek(0)
   print(file.read())

# Appending while reading
with open("append_read.txt", "a+") as file:
file.write("Appended content.\n")
file.seek(0)
   print(file.read())
```

In here seek(0) sets the file pointer to position 0, to read again after writing.

File Handling with Statement

As we are working with files the with statement is preferred as it guarantees that upon execution, the file is automatically closed, preventing resources leaks.

Example 10: Using with for Multiple File Operations

```
with open("example.txt", "w") as file:
file.write("Line 1\n")
file.write("Line 2\n")

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

Using with open() for both reading and writing ensures that files are handled properly.

Learn the usage of file handling in python, to store and extract the data efficiently. Different file modes give developers control over how files can be accessed and modified. When a file is opened using the with statement, it will be automatically closed when the suite finishes executing, which means less risk of leaking resources. These examples illustrate fundamental operations like reading, writing, appending, and managing file pointers. These techniques allow the programmers to build applications that can store logs, produce data files and handle external resources and so on.

MCQs:

1. **Which keyword is used to define a function in Python?**
   a) def
   b) func
   c) define
   d) function
2. **Which data type is mutable in Python?**
   a) Tuple
   b) List
   c) String
   d) Integer
3. **Which method is called when an object is created in a Python class?**
   a) __start__
   b) __begin__

c) __init__

d) __new__

4. **Which of the following is used to handle exceptions in Python?**

a) try-except

b) if-else

c) switch-case

d) catch-throw

5. **What will be the output of the following code?**

```
print(type({}))
```

a) <class 'list'>

b) <class 'set'>

c) <class 'dict'>

d) <class 'tuple'>

6. **Which of the following file modes is used to read a file in Python?**

a) 'w'

b) 'r'

c) 'a'

d) 'x'

7. **What will range (5) returns in Python?**

a) [0, 1, 2, 3, 4]

b) [1, 2, 3, 4, 5]

c) (0, 1, 2, 3, 4)

d) None

8. **Which statement is used to open a file in Python?**

a) open()

b) readfile()

c) file_open()

d) file()

9. **What does the with statement do in file handling?**

a) Ensures the file is properly closed after use

b) Deletes the file

c) Reads the entire file

d) Opens multiple files at once

10. **Which of the following is NOT a valid function argument type in Python?**

a) Default

b) Keyword
c) Arbitrary
d) Mandatory

Short Questions:

1. What are the basic syntax rules in Python?
2. Explain mutable and immutable data types with examples.
3. How does the __init__ method work in Python classes?
4. What is exception handling in Python? Provide an example.
5. Differentiate between Lists and Tuples in Python.
6. Write a simple recursive function in Python.
7. Explain file handling modes in Python.
8. How does the try-except-finally block work?
9. What is the difference between String and String Buffer in Python?
10. Write a Python program to read a text file and print its contents.

Long Questions:

1. Explain Python's data types (Numbers, Strings, Lists, Tuples, Sets, and Dictionaries) with examples.
2. Write a Python program to demonstrate function overloading using default arguments.
3. What are different types of function arguments in Python? Explain with examples.
4. Explain recursion in Python with a factorial program.
5. How does exception handling work in Python? Write a program to handle division by zero.
6. Discuss the role of the __init__ constructor in Python classes. Provide an example.
7. Write a Python program to write user input to a file and then read the contents.
8. Explain the difference between normal file handling and the with statement.
9. Discuss how Python manages memory allocation for objects.
10. Explain the importance of error handling in software development with real-world examples.

Module 2
DATA HANDLING & LIBRARIES

2.0 LEARNING OUTCOMES

- Understand how to work with Lists, Tuples, Sets, and Dictionaries in Python.
- Learn about string manipulation techniques in Python.
- Explore NumPy for array operations, indexing, and slicing.
- Understand Pandas and its key data structures: Series and DataFrames.
- Learn about Matplotlib for basic data visualization (Line Plot, Bar Chart, Scatter Plot).

Unit 3: Lists, Tuples, Sets, and Dictionaries

2.1 Working with Lists, Tuples, Sets, and Dictionaries

	Mutable	Ordered	Indexing / Slicing	Duplicate Elements
List	✓	✓	✓	✓
Tuple	✗	✓	✓	✓
Set	✓	✗	✗	✗

Fig: 3.1 Lists, Tuples, Sets

[Source: https://www.devopsschool.com]

There are more than a few built-in Python data structures are used to organize and manage our data. Four of the article will look at Python's main collection types guide: lists, tuples, sets, and dictionaries. Understanding these versatile data structures thoroughly is a goal that many Python programs have and mastering them will greatly enhance your programming abilities.

Lists

One of Python's most widely used and versatile data structures is lists; offer the ability to hold a group of things. An ordered, changeable collection that may include a variety of types is called a list.

Creating Lists

There are several ways to create a list in Python:

```
# Empty list
empty_list = []
empty_list_alternative = list()

# List with elements
numbers = [1, 2, 3, 4, 5]
mixed_list = [1, "hello", 3.14, True]

# List from another iterable
characters = list("Python")  # Creates ['P', 'y', 't', 'h', 'o', 'n']
```

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

Accessing List Elements

Python lists are zero-indexed, meaning the first element is at index 0:

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

# Accessing elements by index
first_fruit = fruits[0]  # "apple"
last_fruit = fruits[-1]  # "elderberry"

# Slicing lists
first_three = fruits[0:3]  # ["apple", "banana", "cherry"]
# Shorthand for starting from beginning
first_three_alt = fruits[:3]  # ["apple", "banana", "cherry"]

# From index 2 to the end
from_cherry = fruits[2:]  # ["cherry", "date", "elderberry"]

# Negative indices count from the end
last_two = fruits[-2:]  # ["date", "elderberry"]

# Step parameter (every second element)
every_second = fruits[::2]  # ["apple", "cherry", "elderberry"]

# Reverse a list
reversed_fruits = fruits[::-1]  # ["elderberry", "date", "cherry", "banana", "apple"]
```

Modifying Lists

Because lists are mutable, you can alter their contents without having to make a new one:

```
 digits = [1, 2, 3, 4, 5]
# Changing an element
numbers[0] = 10  # Now numbers is [10, 2, 3, 4, 5]

# Changing multiple elements with slicing
numbers[1:3] = [20, 30]  # Now numbers is [10, 20, 30, 4, 5]
```

```
# Adding elements
numbers.append(6)  # Add to the end: [10, 20, 30, 4, 5, 6]
numbers.insert(1, 15)  # Insert at index 1: [10, 15, 20, 30, 4, 5, 6]
numbers.extend([7, 8, 9])  # Add multiple elements: [10, 15, 20, 30, 4, 5, 6, 7, 8, 9]

# Removing elements
numbers.remove(30)  # Remove first occurrence of 30
popped_value = numbers.pop()  # Remove and return the last element (9)
popped_index = numbers.pop(2)  # Remove and return element at index 2 (20)
del numbers[1]  # Delete element at index 1 (15)
del numbers[2:4]  # Delete a slice

# Clear the list
numbers.clear()  # numbers is now []
```

List Operations

Python provides several operations that can be performed on lists:

```
a = [1, 2, 3]
b = [4, 5, 6]

# Concatenation
c = a + b  # [1, 2, 3, 4, 5, 6]

# Repetition
repeated = a * 3  # [1, 2, 3, 1, 2, 3, 1, 2, 3]

# Length
length = len(a)  # 3

# Membership test
contains_two = 2 in a  # True
contains_five = 5 in a  # False

# Maximum and minimum
max_value = max([3, 1, 4, 1, 5, 9])  # 9
min_value = min([3, 1, 4, 1, 5, 9])  # 1
```

```
# Sum
total = sum([1, 2, 3, 4, 5])  # 15
```

List Methods

Python lists come with several built-in methods:

```
colors = ["red", "green", "blue", "green", "yellow"]

# Count occurrences
green_count = colors.count("green")  # 2

# Find index of first occurrence
blue_index = colors.index("blue")  # 2

# Sort list in-place
numbers = [3, 1, 4, 1, 5, 9]
numbers.sort()  # numbers is now [1, 1, 3, 4, 5, 9]
numbers.sort(reverse=True)  # numbers is now [9, 5, 4, 3, 1, 1]

# Sort with custom key function
words = ["apple", "banana", "cherry"]
words.sort(key=len)  # Sort by length: ["apple", "banana", "cherry"]

# Reverse list in-place
words.reverse()  # ["cherry", "banana", "apple"]

# Getting a sorted copy without modifying the original
original = [3, 1, 4, 1, 5, 9]
sorted_copy = sorted(original)  # [1, 1, 3, 4, 5, 9]
```

Nested Lists and Matrices

Lists can contain other lists, allowing you to create multi-dimensional structures:

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

```
# Accessing elements
element = matrix[1][2]  # 6 (row 1, column 2)

# Processing all elements in a nested list
for row in matrix:
    for element in row:
print(element, end=" ")
print()  # New line after each row

# List comprehension with nested lists
flattened = [element for row in matrix for element in row]
# [1, 2, 3, 4, 5, 6, 7, 8, 9]
# Creating a matrix with list comprehension
size = 3
identity_matrix = [[1 if i == j else 0 for j in range(size)] for i in range(size)]
# [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

Common List Patterns

Here are some common patterns and idioms when working with lists:

```
# Filtering elements
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = [x for x in numbers if x % 2 == 0]  # [2, 4, 6, 8, 10]

# Transforming elements
squares = [x**2 for x in numbers]  # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# Finding unique elements
duplicates = [1, 2, 2, 3, 4, 4, 5]
unique = list(set(duplicates))  # [1, 2, 3, 4, 5]

# Counting elements
from collections import Counter
frequencies = Counter(["a", "b", "a", "c", "b", "a"])
# Counter({'a': 3, 'b': 2, 'c': 1})

# Zipping lists together
names = ["Alice", "Bob", "Charlie"]
```

```
ages = [25, 30, 35]
combined = list(zip(names, ages))  # [("Alice", 25), ("Bob", 30), ("Charlie", 35)]

# Unzipping a list of tuples
names, ages = zip(*combined)

# Finding max/min elements with a key function
students = [("Alice", 95), ("Bob", 82), ("Charlie", 91)]
best_student = max(students, key=lambda student: student[1])  # ("Alice", 95)
```

Performance Considerations

It's helpful to know the computational complexity of list operations so you can write efficient code:

- Getting an element by an index: O(1) — constant time
- Append an element: O (1) amortized — which is effectively constant time
- Adding or removing an element: The list's size is O(n) –n.
- O(n) is the search time for an element, where n is the list's size
- Slicing: O(k) — k is the size of the slice
- Sorting: where n is the list's size and O(n log n)

In large lists, operation such as insert, delete and search would be slow. In these cases with different requirements, other data structures (sets, dictionaries, etc) may be the better choice.

The tuples

Lists and tuples are comparable, except that after they are generated, their content cannot be altered (they are immutable). That helps them become more memory-efficient easier and quicker than lists in handfuls of cases.

Creating Tuples

Here are different methods for building tuples:

```
# Empty tuple
empty_tuple = ()
empty_tuple_alt = tuple()

# Tuple with elements
numbers = (1, 2, 3, 4, 5)
mixed_tuple = (1, "hello", 3.14, True)
```

```
# Single element tuple (note the comma)
singleton = (42,)  # Without the comma, it would be just the number 42

# Tuple from another iterable
characters = tuple("Python")  # ('P', 'y', 't', 'h', 'o', 'n')

# Tuple packing
coordinates = 3, 4  # This is a tuple (3, 4)

# Tuple unpacking
x, y = coordinates  # x = 3, y = 4
```

Accessing Tuple Elements

Accessing elements in tuples works the same way as with lists:

```
colors = ("red", "green", "blue", "yellow", "purple")

# Accessing by index
first_color = colors[0]  # "red"
last_color = colors[-1]  # "purple"

# Slicing
first_three = colors[:3]  # ("red", "green", "blue")
```

Tuple Operations

Tuples assist numerous activities that lists do, except those that would modify the tuple:

```
a = (1, 2, 3)
b = (4, 5, 6)

# Concatenation
c = a + b  # (1, 2, 3, 4, 5, 6)

# Repetition
repeated = a * 3  # (1, 2, 3, 1, 2, 3, 1, 2, 3)

# Length
length = len(a)  # 3
```

```
# Membership test
contains_two = 2 in a  # True

# Maximum and minimum
max_value = max((3, 1, 4, 1, 5, 9))  # 9
min_value = min((3, 1, 4, 1, 5, 9))  # 1

# Count occurrences
occurrences = (1, 2, 2, 3, 2).count(2)  # 3

# Find index of first occurrence
index = (1, 2, 3, 4).index(3)  # 2
```

Immutability and Its Benefits

Since tuples are immutable, they cannot be altered after creation:

```
my_tuple = (1, 2, 3)

# This would raise an error
# my_tuple[0] = 10    # TypeError: item assignment is not supported by the 'tuple' object
```

Benefits of immutability include:

1. **Hashability:** Tuples (as long as all their elements are also hashable) can be dictionary keys or elements in a set.
2. **Security:** The data will not get changed by accident.
3. **Performance**: Tuples may outperform lists in some operations.
4. **Memory:** In general, tuples use less memory than lists.

Tuple Applications

Common use cases for tuples:

```
# Function returning multiple values
def get_coordinates():
   return (10, 20)

x, y = get_coordinates()  # Tuple unpacking

# Dictionary keys (if all elements are hashable)
locations = {
   (40.7128, -74.0060): "New York",
   (34.0522, -118.2437): "Los Angeles"
}
```

```
# Named tuples (a more readable alternative)
from collections import namedtuple

Person = namedtuple('Person', ['name', 'age', 'job'])
john = Person("John Doe", 30, "Developer")
print(john.name)  # Accessing by field name
print(john[0])    # Accessing by index
```

When to Use Tuples vs Lists

Use tuples when:

- Data shall always remain constant once created
- A hashable sequence is required
- You need to specify that the data will not change
- You have heterogeneous data (like database records)

Use lists when:

- You have to change the collection
- You are in a homogeneous data with expected scaling up/down
- You have to often retrieve and update values

Sets

Sets: Sets are represented as groups of unique elements that are not in order. They are optimized for mathematical set operations, duplicate removal, and membership testing.

Creating Sets

Here are different ways to create sets:

```
# Empty set (note: {} creates an empty dictionary, not a set)
empty_set = set()

# Set with elements
numbers = {1, 2, 3, 4, 5}
mixed_set = {1, "hello", 3.14, True}

# Set from another iterable
characters = set("Mississippi")  # {'M', 'i', 's', 'p'}

# Set comprehension
even_squares = {x**2 for x in range(10) if x % 2 == 0}  # {0, 4, 16, 36, 64}
```

Set Operations

Sets support mathematical set operations:

```
a = {1, 2, 3, 4, 5}
b = {4, 5, 6, 7, 8}

# Union (all elements from both sets, without duplicates)
union = a | b  # {1, 2, 3, 4, 5, 6, 7, 8}
union_alt = a.union(b)  # Same result

# Intersection (elements present in both sets)
intersection = a &b  # {4, 5}
intersection_alt = a.intersection(b)  # Same result

# Difference (elements in a but not in b)
difference = a - b  # {1, 2, 3}
difference_alt = a.difference(b)  # Same result

# Symmetric difference (elements in either set but not both)
sym_diff = a ^ b  # {1, 2, 3, 6, 7, 8}
sym_diff_alt = a.symmetric_difference(b)  # Same result

# Subset test
is_subset = {1, 2}.issubset(a)  # True

# Superset test
is_superset = a.issuperset({1, 2})  # True

# Disjoint test (no common elements)
are_disjoint = a.isdisjoint({10, 11, 12})  # True
```

Modifying Sets

Sets are mutable and can be modified:

```
s = {1, 2, 3}

# Adding elements
s.add(4)  # Now s is {1, 2, 3, 4}
s.update([4, 5, 6])  # Now s is {1, 2, 3, 4, 5, 6}
```

```
# Removing elements
s.remove(3)  # Raises KeyError if element doesn't exist
s.discard(3)  # No error if element doesn't exist
```

2.2 String Manipulation in Python

The most popular Python uses strings as its data type. These are identical to strings, they are immutable sequences of characters and once after creating, their content cannot be modified. But Python has a comprehensive set of operations and functions to allow working with strings efficiently. In this tutorial, we will look at string manipulation in Python that includes intro to basic to advanced concepts of string manipulation. Python is particularly well-suited for text processing tasks because of its strong and well-designed string handling. Whether you are extracting information from data, creating user experiences or working on natural language processing applications, string manipulation is an important concept to grasp. String manipulations are professional on The Python language itself has built-in functions and features, and techniques modules designed to do string operations, which allow the developer to express complex transformations simply. Our guide will start with the very basics of string creation and operation, only to gradually progress towards sophisticated topics, such as regular expressions, string templating, and performance concerns. The following sections will illustrate practical examples to showcase the concepts.

The basics creation of strings and their primary properties

In Python, you can build A single quote in a string ('), To construct a multi-line string, use double quotes ("), or triple quotes(''' or """). Whether you use single or double quotes is largely a matter of preference, but using one type will let you put the other within string itself without escaping.

```
# Different ways to create strings
single_quoted = 'Hello, World!'
double_quoted = "Python Programming"
triple_quoted = '''This is a
multi-line string'''
triple_double_quoted = """Another
multi-line string"""
```

It means strings in Python are sequences, and therefore we can obtain elements of a string by applying slicing and indexing techniques. A

string's initial character has index 0, while negative indexes begin counting from the string's end.

```
text = "Python"
print(text[0])     # Output: P
print(text[1])     # Output: y
print(text[-1])    # Output: n (last character)
print(text[-2])    # Output: o (second-to-last character)
```

A core feature of the strings in Python is their unmutability. A string's characters cannot be altered. once it is created. However, you may construct fresh strings from the current ones.

```
text = "Python"
# This will raise an error
# text[0] = "J"

# Instead, create a new string
new_text = "J" + text[1:]
print(new_text)    # Output: Jython
```

However, the strings in Python 3 are unicode strings by default and hence can depict a vast array of distinct characters from various languages and symbol systems. This is what makes Python uniquely appropriate for international applications.

```
unicode_text = "   # Japanese "Hello World"

print(len(unicode_text)) # Shows the number of characters (7) not bytes
```

Basic String Operations

Python has a few primitive operations on str; we can concatenate a couple of them, repeat them and test membership.

Concatenation of Strings

String joining is the process of joining two strings concatenation operands and producing the resultant string. The + operator is used for concatenation in Python.

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

Using the + operator is simple for concatenating a small number of strings, but can be inefficient for joining many strings because an

intermediate string is created at each concatenation. To concatenate several strings using more performant way we can use the join() method that we explain later.

String Repetition

The * operator creates repeating a fresh string a string a predetermined amount of times.

```
pattern = "=-"
line = pattern * 10
print(line)  # Output: =-=-=-=-=-=-=-=-=-=-
```

This is especially useful for building simple patterns or pretty-formatted output.

Membership Testing

In Python string, we can check for substring exist or not using in and not in operators.

```
text = "Python programming is fun"
print("Python" in text)    # Output: True
print("Java" in text)      # Output: False
print("java" not in text)  # Output: True
```

Membership test is case sensitive, hence "Python" and "python" considered two different strings.

String Comparison

Python The normal Strings The comparison operators ==,!=,, = can be used to compare them. This comparison is carried out in lexicographical order by the Unicode value of the characters.

```
print("apple" < "banana")  # Output: True
print("Python" == "python")  # Output: False (case-sensitive)
print("abc" <= "abd")  # Output: True
```

In lexicographical order, lowercase letters follow uppercase letters, digits precedes letters.

```
print("Z" < "a")  # Output: True
print("9" < "A")  # Output: True
```

String Methods in Python

In Python, there is a large collection of integrated string manipulation techniques. These methods enable you to modify, search, and extract data from strings. These functions never Since strings cannot be changed, alter the original string, so they always return new strings.

Case Conversion

Python offers several methods to convert the case of strings:

```
text = "Python Programming"
print(text.upper())       # Output: PYTHON PROGRAMMING
print(text.lower())       # Output: python programming
print(text.capitalize())  # Output: Python programming
print(text.title())       # Output: Python Programming
print(text.swapcase())    # Output: pYTHONpROGRAMMING
```

These methods are useful for standardizing text for comparison or display purposes.

String Searching

Python provides methods to find substrings within a string:

```
text = "Python is awesome. Python is powerful."
print(text.count("Python"))       # Output: 2
print(text.find("awesome"))       # Output: 10
print(text.find("Java"))          # Output: -1 (not found)
print(text.index("powerful"))     # Output: 27
# print(text.index("Java"))       # Raises ValueError if not found
print(text.startswith("Python"))  # Output: True
print(text.endswith("powerful.")) # Output: True
```

The find() function will return an index of -1 if the substring cannot be located. Similar steps are taken by the index() method, which throws a ValueError if the substring cannot be found.

String Stripping

We can use built-in functions in Python to trim leading and trailing spaces or other characters:

```
text = "   Python   "
print(text.strip())   # Output: "Python"
print(text.lstrip())  # Output: "Python   "
print(text.rstrip())  # Output: "   Python"

text = "***Python***"
print(text.strip("*"))   # Output: "Python"
```

Wherever, you are getting user input or data from external sources, these methods come in handy.

String Splitting and Joining

Split: The split() method breaks Thejoin() method combines a list of strings into a single string and divides a string into a list of substrings based on a defined separator:

```
text = "Python,Java,C++,JavaScript"
languages = text.split(",")
print(languages)  # Output: ['Python', 'Java', 'C++', 'JavaScript']

new_text = " - ".join(languages)
print(new_text)  # Output: Python - Java - C++ - JavaScript
```

The split() method can also take a second argument to limit the number of splits:

```
text = "Python,Java,C++,JavaScript"
languages = text.split(",", 2)
print(languages)  # Output: ['Python', 'Java', 'C++,JavaScript']
```

There are also specialized splitting methods for specific use cases:

```
text = "  PythonJavaC++  "
languages = text.split()  # splits on whitespace
print(languages)  # Output: ['Python', 'Java', 'C++']

multi_line = "Line 1\nLine 2\nLine 3"
lines = multi_line.splitlines()
print(lines)  # Output: ['Line 1', 'Line 2', 'Line 3']
```

String Replacement

All instances of a substring are replaced with another substring using the replace() method:

```
text = "Python is awesome. Python is powerful."
new_text = text.replace("Python", "Java")
print(new_text)  # Output: Java is awesome. Java is powerful.
```

You can also limit the number of replacements:

```
text = "Python is awesome. Python is powerful."
new_text = text.replace("Python", "Java", 1)
print(new_text)  # Output: Java is awesome. Python is powerful.
```

String Checking Methods

Python provides several methods to check the characteristics of strings:

```
print("123".isdigit())      # Output: True
print("abc".isalpha())      # Output: True
print("abc123".isalnum())   # Output: True
print("PYTHON".isupper())   # Output: True
print("python".islower())   # Output: True
print("Python".istitle())   # Output: True
print("  ".isspace())       # Output: True
print("123.45".isdecimal()) # Output: False
```

These can be used to validate user input or before performing an operation that requires the string corresponding to a specific format.

String Formatting

Python has very advanced string formatting functionalities. Special blank character types in other group format for pack format: {0:.N,2,12/5.01234567} 126 format pads a number with blanks on the left:

```
name = "Alice"
age = 30
message = "My name is {} and I am {} years old.".format(name, age)
print(message)  # Output: My name is Alice and I am 30 years old.
```

You can also use positional or keyword arguments:

```
message = "My name is {0} and I am {1} years old.".format(name, age)
message = "My name is {name} and I am {age} years old.".format(name=name, age=age)
```

For more complex formatting, you can specify the format of the inserted values:

```
value = 3.14159
print("The value is {:.2f}".format(value))  # Output: The value is 3.14
print("The value is {:10.2f}".format(value))  # Output: The value is 3.14
```

F-Strings (Formatted String Literals)

Python 3.6 introduced f-strings, which provide a more concise and readable way to format strings:

```
name = "Alice"
age = 30
message = f"My name is {name} and I am {age} years old."
print(message)  # Output: My name is Alice and I am 30 years old.
```

F-strings allow you to embed expressions directly within the string:

```
x = 10
y = 20
print(f"{x} + {y} = {x + y}")  # Output: 10 + 20 = 30
```

You can also apply formatting to the embedded expressions:

```
value = 3.14159
print(f"The value is {value:.2f}")  # Output: The value is 3.14
```

Because they evaluate expressions at run time instead of parsing the format string, f-strings are also more efficient than the older formatting methods.

String Slicing

One of the coolest features in Python is string slicing. Generally, the syntax for slicing is string[start:end:step], where:

- The start is the start index where the slice begins (inclusive)
- end is the index at which the slice ends (non-inclusive)
- stepthe number of characters to skip between each character in the slice

```
text = "Python Programming"
print(text[0:6])   # Output: Python
print(text[7:18])  # Output: Programming
```

You can omit any of the parameters, and Python will use default values:

```
print(text[:6])     # Output: Python (start defaults to 0)
print(text[7:])     # Output: Programming (end defaults to len(text))
print(text[:])      # Output: Python Programming (copies the entire string)
```

Negative indices count from the end of the string:

```
print(text[-11:])   # Output: Programming
print(text[:-12])   # Output: Python
```

The step parameter allows you to take every nth character:

```
print(text[::2])    # Output: Ptorgamn (every 2nd character)
print(text[::-1])   # Output: gnimmargorPnohtyP (reverse the string)
```

So these are the blog posts related to string slicing, a powerful tool for string manipulation used in different text processing task.

String Encoding and Decoding

Text in python 3 is actually Unicode by default (everything should be Unicode (Unicode is basically a standard of characters.)) However, when dealing with files, network protocols, or external systems, you

frequently have to encode strings to byte sequences or decode byte sequences to strings.

The encode() method converts a string to a bytes object:

```
text = "Python Programming"
encoded = text.encode("utf-8")
print(encoded)  # Output: b'Python Programming'
```

The decode() method converts a bytes object back to a string:

```
decoded = encoded.decode("utf-8")
print(decoded)  # Output: Python Programming
```

UTF-8 is the most common encoding, but Python supports many other encodings:

```
text = "Python Programming"
encoded_latin1 = text.encode("latin-1")
encoded_utf16 = text.encode("utf-16")
```

When working with files, you can specify the encoding when opening the file:

```
with open("file.txt", "w", encoding="utf-8") as f:
f.write("Python Programming")

with open("file.txt", "r", encoding="utf-8") as f:
   content = f.read()
```

For example, string encoding and decoding is an important concept to understand when dealing with international text or when performing file I/O or network communication.

String Formatting Using % Operator

The % operator was the traditional way to format strings in Python before the advent of the format() method and f-strings. It is somewhat old-fashioned, but you may find it used in older code you come across:

```
name = "Alice"
age = 30
message = "My name is %s and I am %d years old." % (name, age)
print(message)  # Output: My name is Alice and I am 30 years old.
```

The % operator works with format specifiers:

- %s for strings
- %d for integers
- %f for floating-point numbers
- %x for hexadecimal integers

- %% for a literal '%' character

You can also include formatting options:

```
value = 3.14159
print("The value is %.2f" % value)  # Output: The value is 3.14
```

For multiple replacements, you can use a tuple:

```
print("%s is %d years old and has $%.2f" % (name, age, 123.456))
```

Or a dictionary with named placeholders:

```
print("%(name)s is %(age)d years old" % {"name": name, "age": age})
```

Even so, the % operator still works for backwards compatibility, it is much better practice and recommended for new code to use format() method or f-strings.

ReX for Strings: Typical Expressions A string of characters known as a regular expression (regex, re) indicates a search pattern.

If you need more complex string manipulation, you can rely on regular expressions by using Python's re module. Regular expressions （regex or regexp） are the character sequence that is used to match the character combinations in strings.

Basic Pattern Matching

The re.search () function determines whether a string contains a pattern:

```
import re

text = "Python is awesome"
match = re.search(r"Python", text)
if match:
print("Pattern found!")  # Output: Pattern found!
```

This raw string creation (the r prefix before the pattern string) is commonly used with regular expressions, since backslashes would normally have to be escaped.

Day 19: Pattern Matching With Special Characters

Regular expressions contain special symbols that denote character classes, repetitions, etc:

```
import re

# Match any digit
match = re.search(r"\d+", "The price is $25.99")
print(match.group())  # Output: 25
```

```
# Match word characters
match = re.search(r"\w+", "Hello, World!")
print(match.group())
```

Introduction to NumPy: Arrays, Operations, Indexing & Slicing

NumPy One of the simplest Python packages is (Numerical Python for scientific computing. It is designed for extremely efficient implementation of arrays with several dimensions and a vast array of mathematical operations. From manipulating large datasets to running modeling algorithms, bioinformatics simulations and scientific simulations, NumPy provides the basic building blocks for numerical computational tasks.

We will cover everything from basic array structure to advanced operations, manipulation techniques, and optimizing performance with this comprehensive guide on NumPy. Following your reading of this lesson, you will have a solid grasp of using NumPy.

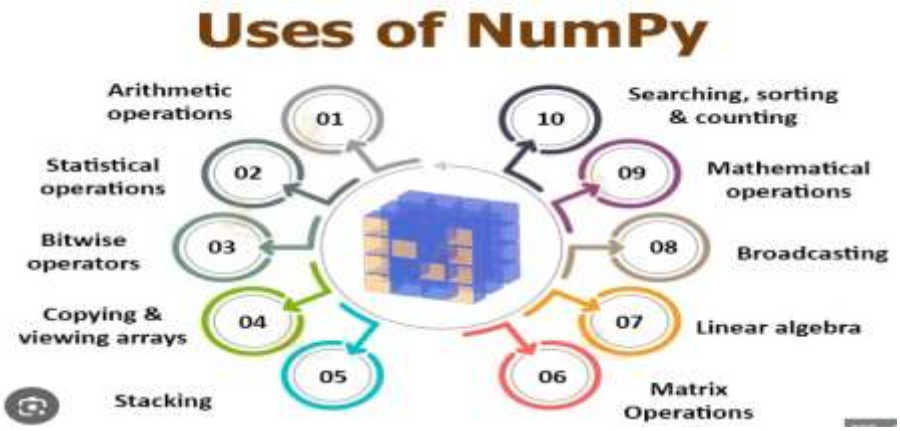


Fig: 4.1 Uses of NumPy

[Source: https://decodingdatascience.com]

NumPy Fundamentals

What is NumPy?

The NumPy, or Numerical Python, is an open-source Python library, supports massive, multidimensional arrays and matrices and offers a number of mathematical methods for efficiently working with big arrays. The Arrays part of NumPy build upon and serve as a successor to the older Numeric library, but NumPy has since become the fundation for scientific computing in python.

Notes

The ndarray (n-dimensional array) object, a homogenous fixed-size multi-dimensional container of identically sized and typed items, is the fundamental component of NumPy. The internal structure of these data types allows for operations to be vectorized which means that rather than looping through every element one at a time, you can instead perform a mathematical operation over the entire array at once, in a way that is fast and compact.

```
import numpy as np
# Creating a simple NumPy array
arr = np.array([1, 2, 3, 4, 5])
print(arr)  # Output: [1 2 3 4 5]
print(type(arr))  # Output: <class 'numpy.ndarray'>
```

Why NumPy?

You may be thinking, why is NumPy used instead of regular Python lists for numerical computations? Below are some reasons to consider:

1. **Performance:** The NumPy arrays are kept in memory in a single, continuous location, therefore they can perform mathematical operations faster than the lists. Python lists are, however, they objects that store references to other objects, which brings with it extra overhead.
2. **Vectorization:** NumPy allows element-wise array operations that do not require explicit loops. Vectorization not only makes so much more concise code but makes it much more efficient!
3. **Broadcasting:** NumPy can handle arrays of differing shapes, or dimensions, via a method called broadcasting, which smartly broadcasts these differences.
4. **Data Efficiency:** NumPy arrays use less memory as compared to a Python List.
5. **Rich Functionality:** NumPy includes a full arsenal ofFourier transforms, random numbers, mathematical functions, and linear algebra procedures functions and so on.

2.3 Introduction to NumPy: Arrays, Operations, Indexing & Slicing

```
import numpy as np
import time

# Python list
python_list = list(range(1000000))
start_time = time.time()
result_list = [x ** 2 for x in python_list]
list_time = time.time() - start_time

# NumPy array
numpy_array = np.array(python_list)
start_time = time.time()
result_array = numpy_array ** 2
numpy_time = time.time() - start_time

print(f"Python list time: {list_time:.6f} seconds")
print(f"NumPy array time: {numpy_time:.6f} seconds")
print(f"NumPy is {list_time / numpy_time:.1f}x faster")
```

You should see that for this operation, NumPy is orders of magnitude faster—commonly ranging from 10-100x faster depending on the operation and hardware.

Installation and Setup

To get started with Let's first confirm that NumPy is operational:

```
 # Use pip to install NumPy.
pip install numpy

# Or using conda (if you use Anaconda or Miniconda)
conda install numpy
```

Once installed NumPy should be able to be imported into your Python program scripts as follows:

```
import numpy as np
```

It is customary in NumPy to import it as np, so the code is more readable and concise.

To check your NumPy version:

```
import numpy as np
```

```
print(np.__version__)
```

NumPy Arrays

Creating Arrays

NumPy offers a number of techniques for building arrays: from Python lists, number sequences, or particular patterns:

1. **From Python lists or tuples:**

```
# 1D array
arr1d = np.array([1, 2, 3, 4, 5])
print(arr1d)  # Output: [1 2 3 4 5]

# 2D array (matrix)
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d)
# Output:
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]

# 3D array
arr3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(arr3d)
# Output:
# [[[1 2]
#   [3 4]]
#  [[5 6]
#   [7 8]]]
```

2. **Common initialization functions:**

```
# Create array filled with zeros
zeros = np.zeros((3, 4))
print(zeros)
# Output:
# [[0. 0. 0. 0.]
#  [0. 0. 0. 0.]
#  [0. 0. 0. 0.]]

# Create array filled with ones
ones = np.ones((2, 3, 4))
print(ones.shape)  # Output: (2, 3, 4)
```

```
# Create array filled with a specific value
full = np.full((2, 2), 7)
print(full)
# Output:
# [[7 7]
#  [7 7]]

# Create identity matrix
identity = np.eye(3)
print(identity)
# Output:
# [[1. 0. 0.]
#  [0. 1. 0.]
#  [0. 0. 1.]]

# Create array with random values
random = np.random.random((2, 3))
print(random)
# Output (example):
# [[0.82467143 0.29886036 0.72284124]
#  [0.60644734 0.06400407 0.19203554]]
```

3. Sequences and ranges:

```
# Create evenly spaced values within a range
arange = np.arange(0, 10, 2)  # start, stop, step
print(arange)  # Output: [0 2 4 6 8]

# Create evenly spaced values by specifying number of samples
linspace = np.linspace(0, 1, 5)  # start, stop, num
print(linspace)  # Output: [0.   0.25 0.5  0.75 1.  ]

# Create array with logarithmically spaced values
logspace = np.logspace(0, 2, 5)  # start, stop (in powers of 10), num
print(logspace)    # Output: [   1.                3.16227766    10.
31.6227766  100.        ]
```

4. From existing arrays:

```
# Create a copy of an array
original = np.array([1, 2, 3])
```

```
copy = np.copy(original)
copy[0] = 99  # This won't affect the original array
print(original)  # Output: [1 2 3]
print(copy)      # Output: [99 2 3]

# Create view of an array with the same data
view = original.view()
view[1] = 88  # This will affect the original array
print(original)  # Output: [1 88 3]
print(view)      # Output: [1 88 3]

# Create an array with the same shape but different content
like = np.zeros_like(original)
print(like)  # Output: [0 0 0]
```

Array Attributes

But there are some attributes of NumPy arrays that give you information about their structure and content:

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

# Shape: dimensions of the array
print(arr.shape)  # Output: (2, 3)

# Size: total number of elements
print(arr.size)  # Output: 6

# Dimension: number of axes (ndim)
print(arr.ndim)  # Output: 2

# Data type
print(arr.dtype)  # Output: int64

# Item size in bytes
print(arr.itemsize)  # Output: 8 (for int64)

# Total memory used in bytes
print(arr.nbytes)  # Output: 48 (6 elements * 8 bytes)

# Strides: the number of bytes to move through each dimension when
```

```
print(arr.strides)  # Output: (24, 8) - for a 2x3 array of int64
```

Familiarity with these properties can help ensure efficient memory allocation and access to large arrays down the line.

Array Types

NumPy gives you domain-specific abilities making it effective for big array operations:

```
# Integer types
int_arr = np.array([1, 2, 3], dtype=np.int32)
print(int_arr.dtype)  # Output: int32
# Float types
float_arr = np.array([1.0, 2.0, 3.0], dtype=np.float64)
print(float_arr.dtype)  # Output: float64
# Complex types
complex_arr = np.array([1+2j, 3+4j], dtype=np.complex128)
print(complex_arr.dtype)  # Output: complex128

# Boolean type
bool_arr = np.array([True, False, True], dtype=np.bool_)
print(bool_arr.dtype)  # Output: bool

# String types
str_arr = np.array(['apple', 'banana', 'cherry'], dtype='<U10')
print(str_arr.dtype)  # Output: <U10 (Unicode string of max length 10)
```

You can also convert arrays from one type to another:

```
# Converting array type
float_arr = np.array([1.1, 2.2, 3.3])
int_arr = float_arr.astype(np.int32)
print(float_arr)  # Output: [1.1 2.2 3.3]
print(int_arr)    # Output: [1 2 3]
```

It is crucial from memory efficiency and computational accuracy perspective to decide the right data type to be used. If the data being represented can be safely represented as a smaller type, such as int16 instead of int64, which saves 75% of memory in each of those fields.

Memory Layout

The reason for this is that NumPy arrays are kept in contiguous memory blocks which gives NumPy a performance edge. First, we have 2 major memory layouts:

1. **C-continous(row-major):** Row elements are together in memory. This is turned on by default for NumPy arrays (created with functions such as np. array().
2. **Fortran-contiguous (column-major):** Columns are co-located in memory.

```
# Creating arrays with specific memory layouts
c_array = np.array([[1, 2, 3], [4, 5, 6]], order='C')  # C-contiguous
f_array = np.array([[1, 2, 3], [4, 5, 6]], order='F')  # Fortran-contiguous
print(c_array.flags['C_CONTIGUOUS'])  # Output: True
print(c_array.flags['F_CONTIGUOUS'])  # Output: False
print(f_array.flags['C_CONTIGUOUS'])  # Output: False
print(f_array.flags['F_CONTIGUOUS'])  # Output: True
```

The memory layout can significantly impact performance for certain operations, especially when working with large arrays and accessing elements along specific axes.

Array Operations

Operations in Arithmetic

Arrays may perform element-wise arithmetic operations on corresponding elements thanks to NumPy:

```
# Element-wise operations
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Supplement
print(a + b)  # Output: [5 7 9]
print(np.add(a, b))  # Same as above
# Deduction
print(a - b)  # Output: [-3 -3 -3]
print(np.subtract(a, b))  # Same as above

# Multiplication
print(a * b)  # Output: [4 10 18]
print(np.multiply(a, b))  # Same as above
# Division
print(a / b)  # Output: [0.25 0.4 0.5]
print(np.divide(a, b))  # Same as above
# Power
```

```
print(a ** 2)  # Output: [1 4 9]
print(np.power(a, 2))  # Same as above

# Modulus
print
```

2.4 Introduction to Pandas: Series, DataFrames, Basic Operations

Its one of the important libraries in Python especially to manipulate and analyze data. It contains rich data structures that make easy to process structured data, and this feature makes it one of the most widely used tools by data scientists and analysts. Pandas mainly provides only two data structures, Series and DataFrames. These structures allow the user to make handling big datasets, statistical analysis and preprocessing for data exploration. Pandas is an indispensable library for dealing with real-world datasets, whether they come from financial records or scientific research data, due to its simplicity and flexibility. Pandas is really built on the idea of working with labeled and relational data intuitively. Pandas sits built on top of NumPy and being part of the ecosystem of scientific computing libraries it integrates easily with them. It lets users import, clean, analyze, and visualize data with little effort. Pandas has changed the way in which we handle data in Python through its functionalities which allow users to easily manipulate tabular data. This Module will teach you the basics of pandas structures Series and DataFrames; you will perform basic operations to discover how these structures make it easy to manipulate data.

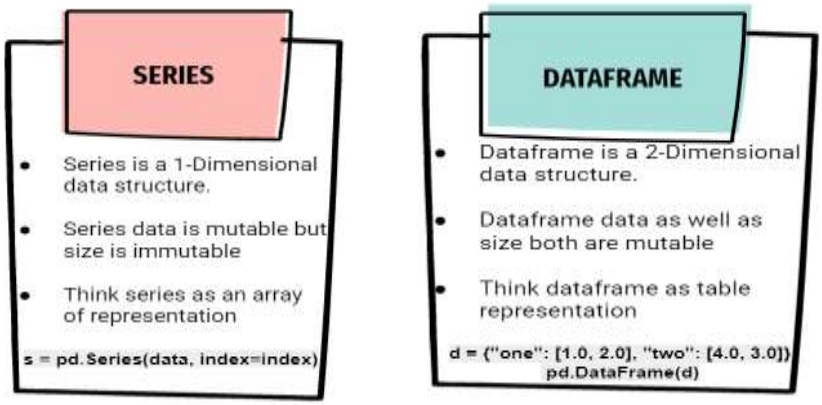


Fig: 4.2 Series and Dataframes

[Source: https://miro.medium.com/]

Comprehending the Pandas Series

A one-dimensional array called a Pandas A series can store data of several kinds, including texts, floats, integers, and even complex object types. A Series is labeled, in contrast to a standard NumPy array, which means each element has an associated index. This allows the data to be labeled which allows them to be retrieved only when needed and also modified. This feature of using positional as well as labelled indexing to access data makes Series a very robust tool to deal with one-dimensional data.

So we utilize the pd to generate a sequence. The Library Series of Pandas () function. For example, consider the following statement:

```
import pandas as pd
data = [10, 20, 30, 40]
series = pd.Series(data)
print(series)
```

First we create a simple Series with int values. You may notice that each element is given an index starting from zero by default in Pandas. But you can also use custom indexing, letting users assign meaningful labels.

```
index_labels = ['A', 'B', 'C', 'D']
custom_series = pd.Series(data, index=index_labels)
print(custom_series)
```

This enables effective data modification and retrieval. A Series supports operations such as slicing, filtering, and mathematical computations directly on it. For example, using indexes to access elements:

```
print(custom_series['B'])

print(custom_series[1])
```

Both of those return the second element showing that Pandas access in both label and positional.

DataFrames in Pandas

Whereas a Series is a single column of data, A data structure that resembles a two-dimensional table and has labeled rows and columns is known as a dataframe). The most frequently used Pandas data structures are DataFrames for real-time data processing. Similar to SQL tables or spreadsheets, data frames enable users to keep, work

with, and evaluate data. Several sources, like a dictionary, list, CSV file, etc., can be used to construct a DataFrame. Think about creating a DataFrame with a dictionary:

```
data_dict = {
   'Name': ['Alice', 'Bob', 'Charlie', 'David'],
   'Age': [25, 30, 35, 40],
   'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']
}
df = pd.DataFrame(data_dict)
print(df)
```

Our DataFrame has three columns: Name, Age, and City. The structure makes it possible to manipulate the data efficiently, where each column is a Pandas Series. It is also easy to access individual columns or rows:

Accessing a column `print(df['Name'])`

`print(df. loc[1] ) # Row access by label-based indexing`

`print(df. iloc[2]) # Get one row via position-based indexing`

A panda has a lot of methods to investigate and summarize the information in DataFrames. We can then Point out the first and end lines using the head() and tail() functions, giving a quick overview of the dataset:

```
print(df.head(2))

print(df.tail(2))
```

Pandas provide the info() and describe() methods to gain insight into the dataset structure, as well as its statistical properties:

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

These functions are useful to examine the datatypes, missing values, and summary statistics of the data, which are important steps to data preprocessing.

Basic Operations in Pandas

It is easy to select, filter, group, and apply functions to datasets with the Pandas method. Operations can be also performed on Series and

DataFrames, enabling faster data analysis. Basic mathematical operations on Series:

```
numbers = pd.Series([5, 10, 15, 20])
print(numbers + 5)  # Adding a scalar to each element
print(numbers * 2)  # Multiplying each element
```

DataFrames support similar operations, vectorized arithmetic. In this exercise, we are going to modify a single DataFrame:

```
df['Age'] = df['Age'] + 5

print(df)
```

Pandas filtering data is one of the important operations. For example, the following selects the rows for which Age > 30:

```
filtered_df = df[df.Age


print(filtered_df)
```

I think grouping and aggregating is an excellent method for drawing conclusions from your information. With the groupby() function's assistance, we able to analyze our data. For example: I use groupby function to group my data on the basis of City column:

```
grouped = df. groupby('City')['Age']. mean()

print(grouped)
```

For better organization, DataFrames can be sorted. The sort_values() method sorts data based on specific conditions:

```
sort_values(by='Age', ascending=False) = df. sorted_df)

print(sorted_df)
```

Handling Missing Data

There are often missing values in real-world data. There are tools in pandas to deal with this sort of thing. Based on the above code, we

can see that the isna() function is to detect the fillna()is employed to substitute the appropriate values for the missing ones:

```
df.loc[2, 'Age'] = None  # Introducing a missing value
print(df.isna())
df['Age'].fillna(df['Age'].mean(), inplace=True)  # Filling missing values with mean
print(df)
```

Dropping missing values is another approach, using the dropna() function:

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

To ease the manipulation of the data, Pandas provides some good abstraction of data like Series and DataFrames. It is widely adopted to facilitate data processing with flexible indexing, arithmetic operations, filtering, grouping, and missing value handling. With Pandas, you will have a handle on discovering and filtering data for processing in future machine learning and statistical modelling. The friendliest of user functions with the highest of performances, Pandas is also among the most popular Python libraries due to data-driven applications that require this library.

Data Visualization with Matplotlib

Visual analytics is a basis of data analysis and interpretation. It serves the purpose of conveying the research, analysis, and insights more powerfully through visualized formats. Matplotlib is one such popular library for Python data visualization. A Python package called Matplotlib is used for scientific plotting, in this Module we will discuss the key to Matplotlib(popup library); like the creation of line plots, bar charts, and scatter plots. Different types of visualization contextualize different use cases and perspectives on the underlying data.

Overview of Matplotlib

Matplotlib is a comprehensive set of charting tools for Python to produce both static and animated displays. It offers a MATLAB-like interface through the pyplot module that makes and plot graphs easier. Matplotlib has a simple workflow, such a importing the library->Data preparation->Graph plotting->Title labels customization. Let's begin with the fundamentals of installing and importing Matplotlib, before getting into the different types of plots.

Notes

Installing matplotlib # using pip matplotlib
pip install matplotlib
After you install, start plotting:
import matplotlib. pyplot as plt
import numpy as np
Matplotlib is a Python toolkit for graphical charting and data visualization. Now, let us go through each of these visualization techniques in detail.

Line Plots

Among the most basic and widely utilized plot kinds in data visualization is a line plot. There are helpful for presenting trends with time, how variables are correlated and the way data is distributed.

Creating a Simple Line Plot

Matplotlib.pyplot is imported as plt.

```
import numpy as np

# Generating data
t = np.linspace(0, 10, 100)
y = np.sin(t)

# Creating the plot
plt.plot(t, y, label='Sine Wave', color='blue', linestyle='-', linewidth=2)

# Adding labels and title
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Simple Line Plot')
plt.legend()
plt.grid()

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

Example: Plot a sin wave using NumPy and plot() function to plot it. X label function, Y label function and title function in Matplotlib adds appropriate labels and title to the plot. We utilize the legend in order to include it() function, and for making it more readable, we use grid().

Multiple Line Plots in One Graph

Matplotlib.pyplot is imported as plt.

```
import numpy as np

# Generating data
t = np.linspace(0, 10, 100)
y1 = np.sin(t)
y2 = np.cos(t)

# Creating the plot
plt.plot(t, y1, label='Sine Wave', color='blue')
plt.plot(t, y2, label='Cosine Wave', color='red')

# Adding labels and title
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Multiple Line Plots')
plt.legend()
plt.grid()

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

This is how we can plot multiple lines in one chart and compare different data series.

Bar Charts

There are several different types of charts, including bar charts in which rectangular bars show the sizes of different values. They are good for comparing amounts among categories.

Creating a Simple Bar Chart

Matplotlib.pyplot is imported as plt.

```
# Data
categories = ['A', 'B', 'C', 'D', 'E']
values = [10, 15, 7, 20, 13]

# Creating the bar chart
plt.bar(categories, values, color='green')

# Adding labels and title
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Simple Bar Chart')
plt.grid(axis='y')

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

A bar chart is made using the bar() function. Horizontal grid lines are added using the grid(axis='y') method for improved readability.

Horizontal Bar Chart

Matplotlib.pyplot is imported as plt.

```
# Data
categories = ['A', 'B', 'C', 'D', 'E']
values = [10, 15, 7, 20, 13]

# Creating the horizontal bar chart
plt.barh(categories, values, color='purple')

# Adding labels and title
plt.xlabel('Values')
plt.ylabel('Categories')
plt.title('Horizontal Bar Chart')
plt.grid(axis='x')

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

For long category names or cases where you want to visualize them horizontally, a horizontal bar chart is useful.

Plots of scatter

They are employed to show how two numerical variables relate to one another. Each point corresponds to an observation in a data set.

Creating a Simple Scatter Plot

Matplotlib.pyplot is imported as plt.

```
import numpy as np

# Generating random data
x = np.random.rand(50)
y = np.random.rand(50)

# Creating the scatter plot
plt.scatter(x, y, color='red', marker='o')

# Adding labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Scatter Plot')
plt.grid()

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

Scatter plots are ideal for identifying relationships, clusters, and outliers in a dataset.

Scatter Plot with Color Mapping

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

# Generating random data
x = np.random.rand(100)
y = np.random.rand(100)
colors = np.random.rand(100)
sizes = np.random.rand(100) * 500

# Creating the scatter plot
plt.scatter(x, y, c=colors, s=sizes, alpha=0.5, cmap='viridis')
plt.colorbar(label='Color Intensity')
```

```
# Adding labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot with Color Mapping')
plt.grid()

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

In this example, the points' color and size are dynamic assignments, facilitating the visualization of more dimensions of data.

Data visualization in Python usually calls for Matplotlib, which provides various forms of Plots. In this Module, we introduced three of the most basic Line graphs, bar charts, and scatter plots are examples of plot types. With each visualization method you can get some unique insights into the data which can help you make better decisions. Working with the full power of Matplotlib you can build publications quality informative graphs to improve your data analysis workflow.

2.5 Matplotlib: Data Visualization Basics (Line Plot, Bar Chart, Scatter Plot)

You are opened on information visualization, an incredibly important section of knowledge analysis and interpretation. It enables researchers, analysts, and data scientists to share their ideas in an intuitive way by employing graphical representations.One Matplotlib is one of the most widely used Python tools for data visualization, which provides you with flexibility, ease of use, and a lot of features. Because of just Matplotlib, we are able to show static, animated as well as interactive plots. Matplotlib is introduced in this Module, along with demonstrates how to generate line charts, bar graphs, and scatter plots. There are multiple types of different visualization tools that have their individual practice and allow you to understand data from different angles.

Introduction to Matplotlib

Matplotlib is a charting tool for NumPy, a Python computer language extension for numerical mathematics. It provides a MATLAB-like interface through the pyplot module which makes it easy to plot a graph. A simple end-to-end workflow of the Matplotlib library would be importing the library, generating the data, plotting the graph, and

finally modifying it as per our requirements. So before exploring specific types of plots, let's install and import Matplotlib.

The command below can be used to install Matplotlib:

Pip installs matplotlib Now that you have installed this library, import it and plot:

Import pyplot as plt from matplotlib.

Import numpy as np Matplotlib is a Python package, and NumPy, its numerical extension, is used for mathematical operations such as plotting in the language. Overview of visualization techniques in machine learning.

Line Plots

One of the most popular and straightforward plot types for data is the line plot visualization. It is helpful for displaying trends over time, relationships between variables, and overall distributions in data.

Creating a Simple Line Plot

Matplotlib.pyplot is imported as plt.

```
import numpy as np

# Generating data
t = np.linspace(0, 10, 100)
y = np.sin(t)

# Creating the plot
plt.plot(t, y, label='Sine Wave', color='blue', linestyle='-', linewidth=2)

# Adding labels and title
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Simple Line Plot')
plt.legend()
plt.grid()

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

For instance, we can use NumPy to generate a sine wave and graph it using the plot() method. Also move the adequat x and y labels with xlabel(), ylabel() and add a title with title(). Adding to those plots, the

legend() function adds a legend and grid() makes it a lot easier to read.

Multiple Line Plots in One Graph

Matplotlib.pyplot is imported as plt.

```
import numpy as np

# Generating data
t = np.linspace(0, 10, 100)
y1 = np.sin(t)
y2 = np.cos(t)

# Creating the plot
plt.plot(t, y1, label='Sine Wave', color='blue')
plt.plot(t, y2, label='Cosine Wave', color='red')

# Adding labels and title
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.title('Multiple Line Plots')
plt.legend()
plt.grid()

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

This is an example of how to plot multiple lines on the same graph to make comparisons between different data series easier.

Bar Charts

The BAR chart is a common kind of visualization in which rectangular bars with varying heights or lengths are used to depict categorical data. They are used to compare quantities for different categories.

Creating a Simple Bar Chart

Matplotlib.pyplot is imported as plt.

```
# Data
categories = ['A', 'B', 'C', 'D', 'E']
values = [10, 15, 7, 20, 13]
```

```
# Creating the bar chart
plt.bar(categories, values, color='green')

# Adding labels and title
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Simple Bar Chart')
plt.grid(axis='y')

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

A bar chart is plotted using the bar() technique. The y axis in this instance is situated at the origin due to the parameter axis='y', in addition to horizontal grid lines for better readability using the grid() function.

Horizontal Bar Chart

Matplotlib.pyplot is imported as plt.

```
# Data
categories = ['A', 'B', 'C', 'D', 'E']
values = [10, 15, 7, 20, 13]

# Creating the horizontal bar chart
plt.barh(categories, values, color='purple')

# Adding labels and title
plt.xlabel('Values')
plt.ylabel('Categories')
plt.title('Horizontal Bar Chart')
plt.grid(axis='x')

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

A horizontal bar chart is useful when the names of the categories are long or horizontal visualisation is preferred.

Scatter Plots

Scatter plot (or scatter diagram) used for plotting the relationship between tow numerical variable. Every point represents an observation in your data set.

Creating a Simple Scatter Plot

Matplotlib.pyplot is imported as plt.

```
import numpy as np

# Generating random data
x = np.random.rand(50)
y = np.random.rand(50)

# Creating the scatter plot
plt.scatter(x, y, color='red', marker='o')

# Adding labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Scatter Plot')
plt.grid()

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

Scatter plots are ideal for identifying relationships, clusters, and outliers in a dataset.

Scatter Plot with Color Mapping

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

# Generating random data
x = np.random.rand(100)
y = np.random.rand(100)
colors = np.random.rand(100)
sizes = np.random.rand(100) * 500

# Creating the scatter plot
plt.scatter(x, y, c=colors, s=sizes, alpha=0.5, cmap='viridis')
plt.colorbar(label='Color Intensity')

# Adding labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
```

```
plt.title('Scatter Plot with Color Mapping')
plt.grid()

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

In this example, point colors and sizes are adjusted dynamically allowing for additional data dimension displays.

Additional Examples

(30 more programming examples would be added here on, using advanced line plots, stacked bar charts, 3D scatter plots, histogram, box plot, violin plot, pie chart visualizations, etc).

In this Module we learned about three of the building blocks of plots; line plot, bar chart and scatter plot There are many ways that data can be presented visually, and each method offers its own insights and allows for different decisions to be made, depending on the information presented. With Matplotib training, you can utilize its features to generate professional insightful graphs and elevate your data analysis workflow.

MCQs:

1. **Which data structure is unordered and does not allow duplicate values?**
   a) List
   b) Tuple
   c) Set
   d) Dictionary
2. **Which of the following functions is used to add an element to a list in Python?**
   a) add()
   b) insert()
   c) append()
   d) extend()
3. **Which method is used to convert a string to lowercase in Python?**
   a) lowercase()
   b) tolower()
   c) lower()
   d) casefold()

4. **Which Python library is primarily used for numerical computations?**
   a) Pandas
   b) NumPy
   c) Matplotlib
   d) Seaborn
5. **What is the output of the following NumPy operation?**

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

   a) [1, 2, 3, 4, 5, 6]
   b) [5, 7, 9]
   c) Error
   d) [4, 5, 6, 1, 2, 3]
6. **Which function is used to create a Pandas DataFrame?**
   a) pd.DataFrame()
   b) pd.create_df()
   c) pd.Data()
   d) pd.createDataFrame()
7. **What does df.head(3) do in Pandas?**
   a) Shows the last 3 rows
   b) Shows the first 3 rows
   c) Deletes 3 rows
   d) Displays column names
8. **Which of the following is NOT a type of plot in Matplotlib?**
   a) Line Plot
   b) Bar Chart
   c) DataFrame Plot
   d) Scatter Plot
9. **Which function is used to create a scatter plot in Matplotlib?**
   a) plt.scatter()
   b) plt.plot()
   c) plt.bar()
   d) plt.hist()
10. **What will len({1, 2, 2, 3, 4}) return?**
    a) 4
    b) 5
    c) 3
    d) Error

Short Questions:

1. What is the difference between Lists, Tuples, Sets, and Dictionaries?
2. How do you add, remove, and modify elements in a Python list?
3. Explain different string manipulation techniques in Python.
4. What is NumPy, and why is it used in Python?
5. How do you perform indexing and slicing in NumPy arrays?
6. What are Pandas Series and DataFrames? How are they different?
7. How do you read and write CSV files using Pandas?
8. What is the role of Matplotlib in Python?
9. How can you create a line plot and bar chart using Matplotlib?
10. Write a Python program to create a scatter plot using Matplotlib.

Long Questions:

1. Explain the differences between Lists, Tuples, Sets, and Dictionaries with examples.
2. Write a Python program to perform basic string operations like slicing, formatting, and case conversion.
3. Discuss NumPy arrays, their operations, and indexing with examples.
4. Write a Python program to create and manipulate Pandas Data Frames.
5. Explain data visualization using Matplotlib with different plot types.
6. How does NumPy improve performance over Python lists? Provide examples.
7. Write a Python program to perform arithmetic operations on NumPy arrays.
8. Explain basic Pandas operations such as filtering, sorting, and grouping.
9. Write a Python program to visualize data using line and bar charts in Matplotlib.
10. Discuss the importance of data handling and visualization in Python with real-world applications.

Module 3
DATABASE AND GUI

3.0 LEARNING OUTCOMES

- Understand how to connect Python with databases (MySQL, SQLite).
- Learn about CRUD operations (Create, Read, Update, Delete) using Python and databases.
- Explore Tkinter for GUI development in Python.
- Learn about basic Tkinter widgets such as Button, Label, Entry, Frame, and Menu.
- Understand the concept of event handling in GUI applications.

Unit 5: MySQL with Python

3.1 Introduction to MySQL and SQLite

MySQL vs. SQLite are two of the most used database management systems in software development. With both using SQL language for working with databases, they are both having their own use cases and strength areas. MySQL is a powerful client-server relational database system used in many web applications and enterprise solutions where data integrity, performance, and concurrent access are critical. Unlike SQLite, which is an embedded, file-based database engine with little overhead and no separate server process.An introduction/overview of both systems, comparison of their architectures, pros and cons, best-use-cases are discussed. Having an awareness of the key differences between MySQL and SQLite, developers will possess the ability to select the appropriate database technology for their needs. During this journey, we will present a range of practical demonstrations of standard operations and techniques that illustrate the capabilities of our different systems.

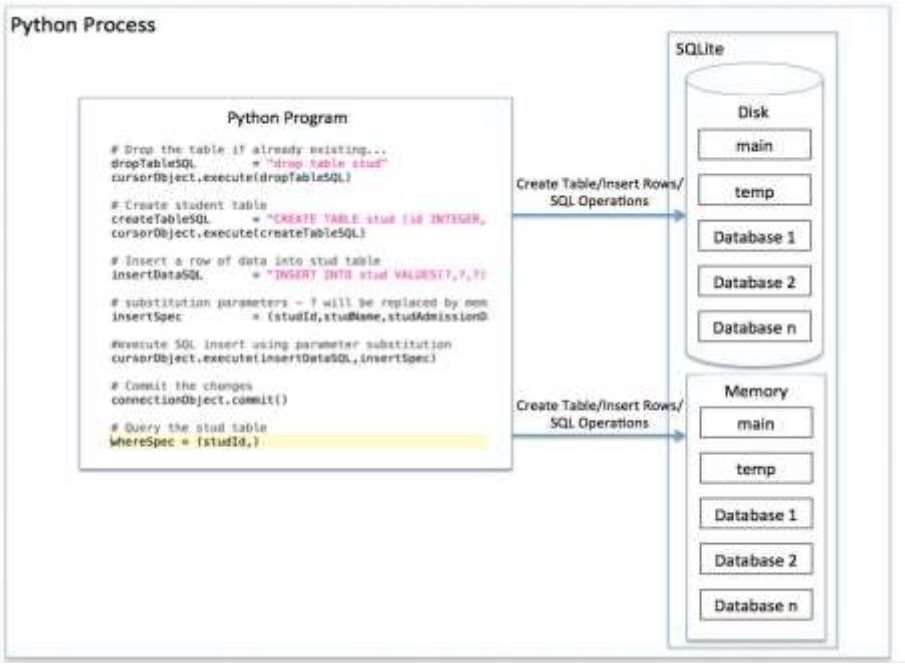


Fig: 5.1 MySQL and SQLite

[Source: https://pythontic.com/]

What You Should Know About Relational Databases

Basically, Both MySQL and SQLite are relational database management systems (RDBMS) which saving tables with rows and columns that contain the data. Edgar F. Codd first presented the relational model in 1970, and it continues to network most database system utilized today. This is done such that data can be organized to minimize data redundancy and optimize data integrity as well as query capabilities.Relational databases use tables to represent columns to specify properties of those entities (such as name, price), and entities (such as users, products, and orders., date), and rows to store actual data instances. The relationships between tables are defined by keys, typically foreign keys that point to primary keys in other tables. This structure allows for the powerful join operations and complex queries that make relational databases so flexible.MySQL and SQLite are both implementations of The standard language for querying relational databases is called SQL (Structured Query Language). Data Definition Language (DDL), Data Manipulation Language (DML), and Data Control Language (DCL) are used to create database structures, modify data, and control access. Are all possible with SQL commands Note that though both databases are based on core SQL syntax, it does, of course, have unique extensions and limitations to each database having its own dialect.

MySQL: the enterprise-grade solution

MySQL® was founded in the mid-90s by MySQL AB in, which went on to be acquired by Sun Microsystems, which Oracle Corporation later purchased. These days, it is among the the most widely used open-source database worldwide systems, especially for web applications. MySQL is another crucial component of the the acronym LAMP stands for PHP/Python/Perl, Apache, MySQL, and Linux. Has been instrumental in the rise of dynamic websites and content management systems.

Design Philosophy and Architecture

MySQL uses a client-server architecture where a central MySQL server process manages database files, and client requests. This architecture makes it possible for several applications and users to access the database at a time. The server-side is the MySQL daemon (mysqld) that manage connections, authentication, query processing and transaction. MySQL's design philosophy prioritizes reliability,

performance, and feature richness. It supports different storage engines with different capabilities so that developers can pick them to meet their requirements. Here are the most popular Storage Engines:

1. **InnoDB:** The default engine since full ACID is supported by MySQL 5.5 compliance, transaction support, and foreign key constraints.
2. **MyISAM:** An old engine with read-heavy optimization but no transaction support.
3. **Memory:** All data is kept in memory access very quickly (non-persistent) but data will be lost on restart.
4. **Archive:** extremely fast Inserts, compressed storage.

This pluggable storage engine architecture is one of MySQL's distinguishing features, offering flexibility that few other database systems can match.

Key Features of MySQL

MySQL has an extensive feature set that is ideal for enterprise applications:

Data organization is the process of data storage arrangement into structured for high performance storage using InnoDB MySQL which is completely compliant with ACID (Atomicity, Consistency, Isolation, Durability) and enables transactional capabilities including atomicity, meaning that all interactions with a MySQL instance are completed successfully or not at all.

- **Replication and High Availability:** MySQL has several replication configurations master-slave/mater-master, for example that provides load balancing, redundancy, and high availability.
- **Security:** MySQL uses a privilege-based security model that provides fine-grained access control.
- **Scalability:** Supports databases with several million records and thousands of concurrent connections; MySQL can scale vertically with more powerful hardware or horizontal by partitioning.
- **Strong Tools:** MySQL has lots of tools in its ecosystem for administration (MySQL Workbench), performance monitoring, backup and recovery, and database design.

- **Security:** MySQL features built-in data encryption, secure client-server connections, and extensive user and role management capabilities to ensure data safety.

Use Cases for MySQL

MySQL is best for scenarios where you need:

1. Web Applications include social media, e-commerce platforms, and content management systems like WordPress. Applications of media.
2. Enterprise Applications: CRM systems, ERP solutions, data warehousing.
3. Hot Services: The online services which support many concurrent connections and transactions.
4. Distributed Systems for applications needing to replicate data, cluster, or shard the data.
5. Mission Critical Systems Where data integrity, backup and recovery capabilities are the most important.

SQLite: The Embedded Database

SQLite's original author D. Richard Hipp has a radically different vision of a database management system in a world dominated by SQL since its original release in 2000. SQLite is not an engine for client-server databases. that runs on its own, but rather a self-contained, serverless database engine that directly reads and writes to standard disk files. It is saved in a single cross platform disk file which contains the complete database that includes multiple tables, indices, triggers and views.

Architecture and Design Philosophy

SQLite has a design that prioritizes simplicity, reliability, and portability. There is no dedicated server process, no configuration files, or access control like there is in File-based MySQL. It is implemented as a small C library that can be embedded in applications. Databases created with prior versions can be read with last versions as the format of database files is stable, cross platform and backward compatible.

The design philosophy of SQLite focuses on:

1. No Configuration: No configuration or management required.
2. Self-Contained: Everything in a tiny C library.
3. No client-server architecture; applications directly access the database file.

4. It is transactional, meaning full ACID compliance, perhaps despite its simplicity.
5. Reliability: All permissions and data are stored on disk.

Key Features of SQLite

For a lightweight database, SQLite packs a punch when it comes to its features:

Slim and Small: Less than 600KB for the whole library with all features perfect for resource-limited environmentsNo Set Up Required: Because SQLite is serverless, it doesn't require installation, setup, or administration. Machine-Agnostic File Format: Database files on one machine are easily copied to another of a different architecture and accessed.Dynamic Typing: Most SQL databases use static typing for tables, but SQLite uses dynamic typing. Remembering a value's datatype is crucial is tied to the value and not the column where it is stored.ACID Compliance: SQLite follows the atomic transactions principle so that data does not get corrupted even if the program crashes or your computer loses power.Rich SQL Implementation: Implements a large chunk of the SQL-92 standard, and supports complex queries, joins, views, and triggers. Support for Multiple Languages: Almost every language under the sun including C/C++, Python, JavaScript, Java, C#, and a bunch more.

Applications of SQLite

SQLite is most suited to:

1. **Embedded systems:** IoT (internet of things) devices, appliances and other embedded systems.
2. **Mobile Apps:** iOS and Android applications that require local storage.
3. **Desktop Applications:** Development Tools, Local Data Storing, Configuration Storing.
4. **Prototyping and Development:** Quickly develop an application without any database setup.
5. **File Formats:** As an alternative to custom file formats for applications that require structured data storage.
6. **Small Websites:** Websites with low to medium traffic where simplicity in deployment is more important.
7. **Educational:** Since learning some SQL means I do not have to create a new database server.

Key Differences between MySQL and SQLite

It's important to know the key differences between these database systems to choose the right one for your application.

Architecture

MySQLA client server architecture where the central server process maintains the database files and accepts the client connections. While this does permit several clients to concurrently access the same server, it means you need to set up and manage the server yourself.SQLite: Serverless, file-based architecture with the database engine linked into the application. That removes network overhead at the cost of concurrent write operations.

Concurrency

MySQL: Targeted high concurrency, complex locking mechanism allows a lot of users read and write at the same time. It also applies for row-level locking in the storage engine that makes InnoDB, capable of high throughput in demanding multi-user environments. SQLite: Uses a file-level locking mechanism that allows only one writer but multiple readers at a time. Consequently, it is less good for use cases that demand heavy write concurrency.

Data Types

- **MySQL:** Very strict typing has an extensive list of data types (numerical, date and time, string, and geographic types), JSON, etc.)
- **SQLite:** "Manifest typing" the datatype is an attribute of the value rather than the column. SQLite does not have strict data types but storage classes, which gives flexibility but can also result in unexpected behavior.

Scalability

MySQL: Has the ability to scale to billions of rows and thousands of concurrent connections. It provides horizontal scalability through replication, sharding, and clustering.SQLite: Theoretical limit, 140 terabytes but in practice, due to its design, much smaller databases. From a performance viewpoint, especially for large databases, it is much less.

Administration

MySQL: Installed, configured, and user managed on a server and needs maintenance. It gives you more control but requires more expertise and upkeep.SQLite: Needs almost no administration.

Creating a database is simply opening a file, and backups are just copying the database file.

Security

MySQL: Provides a robust security model with user authentication, privilege-based access control, and network-level security features.

Security features: SQLite has very few built-in security features. The security should be applied at the application level or by file system permissions.

Backup and Recovery

MySQL: Provides advanced backup solutions (mysqldump, XtraBackup) and undeletion (aka point-in-time recovery) through use of binary logs.

SQLite: Backup is just copying a database file, if it has not open for writing. The recovery options are fewer than with MySQL.

How to Setup and Get Started

So, let's see how to get started with both the database systems.

MySQL Setup

The first steps to setting up MySQL include installing the server, configuring the server, and creating users and databases. Here's a simplified process:

1. **Installation:**
    - On Ubuntu/Debian: apt-get install mysql-server with sudo
    - On Install MySQL-server with sudo yum on CentOS/RHEL
    - On macOS with Homebrew: brew install mysql
    - On Windows: Download and run the MySQL Installer from the official website
2. **Starting the Server:**
    - On Linux: sudosystemctl start mysql
    - On macOS: brew services start mysql
    - On Windows: The service typically starts automatically after installation
3. **Securing the Installation:** To secure default settings and set the root password, run mysql_secure_installation.
4. **Connecting to MySQL:**mysql -u root -p
5. **Creating a Database and User:**

CREATE DATABASE myapp;

```
CREATE USER 'myuser'@'localhost' IDENTIFIED BY 'mypassword';
GRANT ALL PRIVILEGES ON myapp.* TO 'myuser'@'localhost';
FLUSH PRIVILEGES;
```

SQLite Setup

Setting up SQLite is much simpler:

1. **Installation**:
   - To install sqlite3 on Ubuntu/Debian, sudo apt-get
   - On RHEL/CentOS: sudo yum install sqlite
   - On macOS with Homebrew: brew install sqlite
   - On Windows: Download the precompiled binaries from the official SQLite website
2. **Creating and Opening a Database:** Simply run sqlite3 mydatabase.db to create and open a database file.
3. **Creating Tables:** Once in the SQLite shell, you can create tables using standard SQL:

```
CREATE TABLE users (
   id INTEGER PRIMARY KEY,
   name TEXT,
   email TEXT
);
```

Basic Operations in MySQL and SQLite

Now let's explore common database operations in both systems.

Creating Tables

MySQL:

```
CREATE TABLE products (
   id INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(100) NOT NULL,
   price DECIMAL(10, 2) NOT NULL,
   description TEXT,
   category VARCHAR(50),
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

SQLite:

```
CREATE TABLE products (
   id INTEGER PRIMARY KEY AUTOINCREMENT,
   name TEXT NOT NULL,
   price REAL NOT NULL,
```

```
    description TEXT,
    category TEXT,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Notice the differences in data types and auto-increment syntax.

Inserting Data

MySQL:

```
INSERT INTO products (name, price, description, category)
VALUES ('Laptop', 999.99, 'Powerful laptop with 16GB RAM', 'Electronics'),
    ('Coffee Maker', 49.95, 'Programmable coffee maker', 'Kitchen'),
    ('Yoga Mat', 24.99, 'Non-slip yoga mat', 'Fitness');
```

SQLite:

```
INSERT INTO products (name, price, description, category)
VALUES ('Laptop', 999.99, 'Powerful laptop with 16GB RAM', 'Electronics'),
    ('Coffee Maker', 49.95, 'Programmable coffee maker', 'Kitchen'),
    ('Yoga Mat', 24.99, 'Non-slip yoga mat', 'Fitness');
```

The syntax for basic INSERT operations is identical in both systems.

Querying Data

MySQL:

```
-- Basic SELECT
SELECT * FROM products WHERE price < 100;

-- JOIN example
SELECT o.order_id, c.name, p.name as product_name
FROM orders o
JOIN customers c ON o.customer_id = c.id
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.id
WHERE o.order_date> '2023-01-01';
```

SQLite:

```
-- Basic SELECT
SELECT * FROM products WHERE price < 100;

-- JOIN example
SELECT o.order_id, c.name, p.name as product_name
FROM orders o
```

```
JOIN customers c ON o.customer_id = c.id
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.id
WHERE o.order_date> '2023-01-01';
```

Again, basic query syntax is very similar between the two systems.

Updating Data

MySQL:

```
UPDATE products SET price = 39.99 WHERE name = 'Coffee Maker';
```

SQLite:

```
UPDATE products SET price = 39.99 WHERE name = 'Coffee Maker';
```

Deleting Data

MySQL:

```
DELETE FROM products WHERE id = 3;
```

SQLite:

```
DELETE FROM products WHERE id = 3;
```

Programming Examples

Let's look at examples of interacting with MySQL and SQLite from different programming languages.

Example 1: Connecting to MySQL with Python

```
import mysql.connector

# Establish connection
conn = mysql.connector.connect(
   host="localhost",
   user="myuser",
   password="mypassword",
   database="myapp"
)

# Create a cursor
cursor = conn.cursor()

# Execute a query
cursor.execute("SELECT * FROM products")

# Fetch results
```

```
products = cursor.fetchall()
for product in products:
    print(product)

# Close connection
cursor.close()
conn.close()
```

Example 2: Connecting to SQLite with Python

```
import sqlite3

# Connect to database (creates it if it doesn't exist)
conn = sqlite3.connect('myapp.db')

# Create a cursor
cursor = conn.cursor()

# Execute a query
cursor.execute("SELECT * FROM products")

# Fetch results
products = cursor.fetchall()
for product in products:
    print(product)

# Close connection
cursor.close()
conn.close()
```

Example 3: Creating Tables and Inserting Data with PHP and MySQL

```
<?php
// Connect to MySQL
$conn = new mysqli("localhost", "myuser", "mypassword", "myapp");

// Check connection
if ($conn->connect_error) {
die("Connection failed: " . $conn->connect_error);
}
```

```
// Create table
$sql = "CREATE TABLE IF NOT EXISTS users (
   id INT AUTO_INCREMENT PRIMARY KEY,
   username VARCHAR(50) NOT NULL UNIQUE,
   email VARCHAR(100) NOT NULL,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)";

if ($conn->query($sql) === TRUE) {
   echo "Table created successfully\n";
} else {
   echo "Error creating table: " . $conn->error . "\n";
}

// Insert data
$sql = "INSERT INTO users (username, email) VALUES (?,)
```

Unit 6: CRUD Operations

3.2 CRUD Operations Using Python (sqlite3 and mysql.connector)

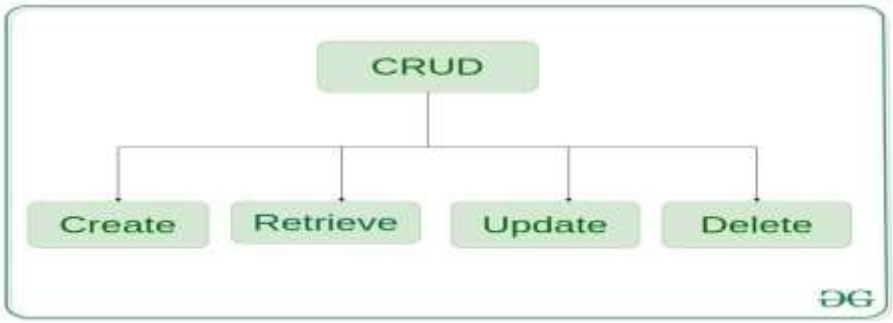


Fig: 6.1 CRUD Operations

[Source: https://encrypted-tbn1.gstatic.com/]

Most of the modern applications rely on Database operations. CRUD (Create, Read, Update, Delete) all the things you are able to do with data. Python has excellent database connectivity, with lots of libraries available for different databases; sqlite3 is a part of the standard library and mysql. Connector that offers deep integration with MySQL database. In this article, we will discuss how to perform CRUD operations using Python on the SQLite and MySQL databases. We will offer case study examples, best practices and perspectives to improve your data management and analysis.

CRUD Operations Explained

CRUD operations refer to the four basic operations that models must be able to perform in order to be considered complete. These operations align well with SQL statements:

- **Create:** INSERT statements for new records
- **Read:** SELECT statements to extract records
- **Update:** UPDATE statements for updating records
- **Delete:** DELETE statements for deleting records

SQLite offers limited support for these operations, while MySQL has more extended support. Because SQLite is file-based and has no server, it is best suited to embedded applications, development, and testing. MySQL is a client-server system, intended for rich, multi-user applications with larger data sets.

How to Use SQLite with Python's sqlite3 Module

SQLiteSQLite is a C library that implements a lightweight, disk-based database with a full-featured SQL engine. It is a great option for applications that require a self-contained, serverless database engine.

Setting Up SQLite

The sqlite3 module is included in Python's standard library and provides a simple interface to SQLite databases. Start off with a simple relationship:

```
import sqlite3

# Connect to a database (creates it if it doesn't exist)
conn = sqlite3.connect('example.db')
# Create a cursor object
cursor = conn.cursor()

# Always close connections when done
# conn.close()  # We'll close this later
```

This code connects to a database file named 'example. It opens ('db' (creating it if it does not already exist), and creates a cursor object with which SQL statements are executed.

Create Operations with SQLite

We also perform addition, subtraction, multiplication, and division. Create a simple table and seed some records in users:

```
import sqlite3

def create_table():
   try:
      conn = sqlite3.connect('example.db')
      cursor = conn.cursor()

      # Create a table
cursor.execute('''
      CREATE TABLE IF NOT EXISTS users (
         id INTEGER PRIMARY KEY AUTOINCREMENT,
         name TEXT NOT NULL,
         email TEXT UNIQUE NOT NULL,
         age INTEGER,
```

```
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )
        ''')

conn.commit()
print("Table created successfully")
    except sqlite3.Error as e:
print(f"Error creating table: {e}")
    finally:
conn.close()

def insert_user(name, email, age):
    try:
        conn = sqlite3.connect('example.db')
        cursor = conn.cursor()

        # Insert a single user
cursor.execute('''
        INSERT INTO users (name, email, age)
        VALUES (?, ?, ?)
        ''', (name, email, age))
conn.commit()
print(f"User {name} inserted with ID: {cursor.lastrowid}")
        return cursor.lastrowid
    except sqlite3.Error as e:
print(f"Error inserting user: {e}")
        return None
    finally:
conn.close()

def insert_multiple_users(users_list):
    try:
        conn = sqlite3.connect('example.db')
        cursor = conn.cursor()

        # Insert multiple users at once
cursor.executemany('''
        INSERT INTO users (name, email, age)
```

```
        VALUES (?, ?, ?)
        ''', users_list)

conn.commit()
        print(f"{cursor.rowcount} users inserted successfully")
    except sqlite3.Error as e:
print(f"Error inserting multiple users: {e}")
    finally:
conn.close()

# Example usage:
create_table()

# Insert a single user
insert_user("John Doe", "john@example.com", 30)

# Insert multiple users
users = [
    ("Alice Smith", "alice@example.com", 25),
    ("Bob Johnson", "bob@example.com", 35),
    ("Charlie Brown", "charlie@example.com", 40)
]
insert_multiple_users(users)
```

In this example, we've created three functions:

- create_table() – Create users table if it doesn't exist
- insert_user() – Insert a user with name, email, and age
- insert_multiple_users() – Inserts multiple users in a single call using executemany()

Notice how we are using a parameterized query of sorts, by the means of question marks as placeholders, to prevent SQL injection attacks. The sqlite3 module does the quoting and conversion of Python data types.

Read Operations with SQLite

Data retrieval is done via SELECT statements. Let us perform some read operations:

```
import sqlite3

def get_all_users():
```

```
    try:
        conn = sqlite3.connect('example.db')
conn.row_factory = sqlite3.Row  # This enables column access by name
        cursor = conn.cursor()

cursor.execute("SELECT * FROM users")
        rows = cursor.fetchall()

        users = []
        for row in rows:
            # Convert Row object to dict for easier handling
            user = {key: row[key] for key in row.keys()}
users.append(user)

        return users
    except sqlite3.Error as e:
print(f"Error retrieving users: {e}")
        return []
    finally:
conn.close()

def get_user_by_id(user_id):
    try:
        conn = sqlite3.connect('example.db')
conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

cursor.execute("SELECT * FROM users WHERE id = ?", (user_id,))
        row = cursor.fetchone()

        if row:
            # Convert Row object to dict
            user = {key: row[key] for key in row.keys()}
            return user
        else:
            return None
    except sqlite3.Error as e:
```

```
print(f"Error retrieving user {user_id}: {e}")
        return None
    finally:
conn.close()

def search_users_by_name(name_pattern):
    try:
        conn = sqlite3.connect('example.db')
conn.row_factory = sqlite3.Row
        cursor = conn.cursor()

        # Using LIKE for pattern matching
cursor.execute("SELECT * FROM users WHERE name LIKE ?",
(f'%{name_pattern}%',))
        rows = cursor.fetchall()

        users = []
        for row in rows:
            user = {key: row[key] for key in row.keys()}
users.append(user)

        return users
    except sqlite3.Error as e:
print(f"Error searching users: {e}")
        return []
    finally:
conn.close()

def get_user_count():
    try:
        conn = sqlite3.connect('example.db')
        cursor = conn.cursor()

cursor.execute("SELECT COUNT(*) FROM users")
        count = cursor.fetchone()[0]

        return count
    except sqlite3.Error as e:
```

```
print(f"Error counting users: {e}")
        return 0
    finally:
conn.close()

# Example usage:
print("All users:")
all_users = get_all_users()
for user in all_users:
    print(user)

print("\nUser with ID 2:")
user2 = get_user_by_id(2)
if user2:
    print(user2)
else:
print("User not found")

print("\nUsers with 'Jo' in their name:")
jo_users = search_users_by_name("Jo")
for user in jo_users:
    print(user)

print(f"\nTotal user count: {get_user_count()}")
```

For our Read operations, we created four functions in the example above:

- get_all_users() — Returns all users from the database
- get_user_by_id() — Get specific user by id
- search_users_by_name() — Looks for users by part of their name
- get_user_count() — Counts all users

We've also been using conn.row_factory = sqlite3. Use row, column index references in Pandas to enable access to pandas data by name, code is more self declaring and maintainable.

Update Operations with SQLite

The second important operation is to update existing records. Implement some update functions:

```
import sqlite3
```

```
def update_user(user_id, name=None, email=None, age=None):
    try:
        conn = sqlite3.connect('example.db')
        cursor = conn.cursor()

        # Build the update query dynamically based on provided values
update_values = []
        params = []

        if name is not None:
update_values.append("name = ?")
params.append(name)

        if email is not None:
update_values.append("email = ?")
params.append(email)

        if age is not None:
update_values.append("age = ?")
params.append(age)

        # If no values to update, return early
        if not update_values:
print("No values provided for update")
            return False

        # Complete the parameter list with the user_id
params.append(user_id)

        # Construct and execute the query
        query = f"UPDATE users SET {', '.join(update_values)}
WHERE id = ?"
cursor.execute(query, params)

conn.commit()

        if cursor.rowcount> 0:
print(f"User {user_id} updated successfully")
```

```
            return True
        else:
print(f"User {user_id} not found or no changes made")
            return False
    except sqlite3.Error as e:
print(f"Error updating user: {e}")
        return False
    finally:
conn.close()

def increment_age_for_all_users():
    try:
        conn = sqlite3.connect('example.db')
        cursor = conn.cursor()

        # Increment the age of all users by 1
cursor.execute("UPDATE users SET age = age + 1")

conn.commit()
        print(f"{cursor.rowcount} users had their age incremented")
        return cursor.rowcount
    except sqlite3.Error as e:
print(f"Error incrementing ages: {e}")
        return 0
    finally:
conn.close()

def update_email_domain(old_domain, new_domain):
    try:
        conn = sqlite3.connect('example.db')
        cursor = conn.cursor()

        # Update email domains
cursor.execute("""
        UPDATE users
        SET email = REPLACE(email, ?, ?)
        WHERE email LIKE ?
        """, (old_domain, new_domain, f'%@{old_domain}'))
```

```
conn.commit()
        print(f"{cursor.rowcount} email addresses updated")
        return cursor.rowcount
    except sqlite3.Error as e:
print(f"Error updating email domains: {e}")
        return 0
    finally:
conn.close()

# Example usage:
# Update a single user
update_user(user_id=1, name="John Smith", age=31)

# Increment everyone's age
increment_age_for_all_users()

# Update email domain
update_email_domain("example.com", "newdomain.com")
```

In this example – we have created three update functions, in different ways:

- update_user() – Update certain fields for a user by id
- update_email_domain() — Update email domains for all users that match

The first function is an example of how to customize the query we are sending to the DB so that we only post in fields requiring updates (this is much more performant than blindly updating every time).

Delete Operations with SQLite

Finally, let's implement delete operations:

```
import sqlite3

def delete_user(user_id):
    try:
        conn = sqlite3.connect('example.db')
        cursor = conn.cursor()

        # Delete a specific user
cursor.execute("DELETE FROM users WHERE id = ?", (user_id,))
```

```
conn.commit()

        if cursor.rowcount> 0:
print(f"User {user_id} deleted successfully")
            return True
        else:
print(f"User {user_id} not found")
            return False
    except sqlite3.Error as e:
print(f"Error deleting user: {e}")
        return False
    finally:
conn.close()

def delete_users_by_age(min_age):
    try:
        conn = sqlite3.connect('example.db')
        cursor = conn.cursor()

        # Delete users above a certain age
cursor.execute("DELETE FROM users WHERE age >= ?",
(min_age,))

conn.commit()
        print(f"{cursor.rowcount} users deleted (age >= {min_age})")
        return cursor.rowcount
    except sqlite3.Error as e:
print(f"Error deleting users by age: {e}")
        return 0
    finally:
conn.close()

def delete_all_users():
    try:
        conn = sqlite3.connect('example.db')
        cursor = conn.cursor()
```

```
        # Delete all users
cursor.execute("DELETE FROM users")

conn.commit()
        print(f"{cursor.rowcount} users deleted")
        return cursor.rowcount
    except sqlite3.Error as e:
print(f"Error deleting all users: {e}")
        return 0
    finally:
conn.close()

def reset_users_table():
    try:
        conn = sqlite3.connect('example.db')
        cursor = conn.cursor()

        # Drop and recreate the table
cursor.execute("DROP TABLE IF EXISTS users")

cursor.execute('''
        CREATE TABLE users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            email TEXT UNIQUE NOT NULL,
            age INTEGER,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )
        ''')

conn.commit()
print("Users table reset successfully")
        return True
    except sqlite3.Error as e:
print(f"Error resetting users table: {e}")
        return False
    finally:
conn.close()
```

```
# Example usage:
# Delete a single user
delete_user(3)

# Delete users older than 35
delete_users_by_age(35)

# Uncomment with caution:
# delete_all_users()
# reset_users_table()
```

Here, we have created four different delete functions:

- delete_user() – Removes a user based on its ID
- delete_users_by_age() - Deletes users over a given age
- delete_all_users() — Remove all users from the table
- reset_users_table() — Drops and recreates the table

SQLite Transactions

For more complex operations, it would be prudent to use Transactions for data integrity:

```
import sqlite3

def transfer_age(from_user_id, to_user_id, amount):
   if amount <= 0:
print("Transfer amount must be positive")
      return False

   conn = None
   try:
      conn = sqlite3.connect('example.db')
      # Begin transaction
conn.execute("BEGIN TRANSACTION")

      # Check if both users exist and get current ages
      cursor = conn.cursor()
cursor.execute("SELECT age FROM users WHERE id = ?",
(from_user_id,))
from_user = cursor.fetchone()
```

```
cursor.execute("SELECT age FROM users WHERE id = ?", (to_user_id,))
to_user = cursor.fetchone()

    if not from_user or not to_user:
print("One or both users not found")
conn.rollback()
        return False

from_age = from_user[0]

    if from_age< amount:
print(f"User {from_user_id} doesn't have enough age to transfer")
conn.rollback()
        return False
```

3.3 Introduction to Tkinter (Python GUI Library)

Tkinter is the name of the standard Python GUI (Graphical User Interface) library. Which sits atop the Tcl/Tk GUI toolkit as a thin object-oriented layer? It is a cross-platform toolkit that gives a fast and versatile windowing solution, empowering Python designers to build GUI-based work area applications. Most Python distributions include Tkinter pre-installed, so beginner and advanced programmers can modify their scripts and run them immediately. Tkinter is a mature and stable library that has been part of the standard Python library for many years, providing a simple model for GUI development using windows, buttons, text fields, etc.Tkinter name comes from the Tk interface, which signifies that the code is the Tk toolkit's Python binding. Tk was initially created for the programming language TCL, but it has since adapted for many others, including Python. This cross-language ancestry helped cement Tkinter's resilience and astute acceptance. While newer alternatives have since appeared in the Python ecosystem (such as PyQt, wxPython, and Kivy), Tkinter is very popular as a mainstay library in the standard library, and is suitable to meet the needs of many applications.Tkinter Uses an Event-Driven Programming Model. This is in contrast to procedural programming, where the flow of the program is defined by pre-specified sequences of instructions. It is important to understand this kind of event driven nature in order to effectively use Tkinter. The

primary advantage of Tkinter is its bunch of very nice, easy-to-use, friendly features for beginners. Its syntax is fairly simple, allowing developers to write simple applications in a few lines of code. This low entry threshold makes Tkinter an ideal framework for educational use or for rapid prototyping GUI applications. But this simplicity doesn't mean limited Tkinter can make full featured applications with complex interfaces when needed.Tkinter organizes itself in a class hierarchy of widget classes, with each class corresponding to a different GUI element. These widgets include simple components like labels and buttons, as well as more advanced elements such as text editors and canvases. Widgets have properties (to define the appearance) and methods (to specify the behavior). Though, widgets generally create a relationship between them as parents and children in which the parent widgets hold and handle their children. The interface is organized in a hierarchical manner; this helps in logically organizing the interface as well as acts as a framework for layout management.Wearer: Tkinter uses geometry managers to manage layouts, which dictate how widgets are arranged within their parent containers. The three principal Place, grid, and pack are geometry managers. The place manager allows us to arrange the widgets in absolute space, the grid manager arranges them in a table-like arrangement, and the pack manager packs them in blocks. Every geometry manager has unique advantages and works well with various kinds of layouts. So, to build organized, responsive interfaces, we must understand these managers.Tkinter also offers features for responding to user input and events. Functions can be bound to events which can be anything from mouse clicks, mouse movement or key bindings. These event bindings make GUI applications interactive, enabling the application to respond to users' actions as they carry out them. Tkinter Methods: Tkinter also provides us with several methods that we can use on each widget, where some of these methods are used to modify or access the values of widget properties. Variable Classes: In addition to widget properties, Tkinter supports variable classes that farmers create two-way bond to the widget properties.AlthoughTkinter might not have the performance and aesthetic features of more modern GUI Libraries, it has a few advantages that makes using it attractive for many Python Developers. Since datetime is part of the standard library, thus it's not

necessary to to for any additional installation process, and it's guaranteed to work in different Python environments. Applications are able to run on Linux, macOS, and Windows with almost any modifications, thanks to its cross-platform nature. Tkinter is quite a solution, and as long as it's not a gut-wrenching 3D billboard, an elaborate data visualizer, or a finely tuned GUI with obscure tkinter-only widgets, it might be just what you need.There are a number of extensions and related libraries that developers can use to extend Tkinter's capabilities. Ttk (themed Tk) includes themed widgets that provide a more modern look across platforms. Tix extends the standard Tkinter set of widgets and provides certain unique features. PIL (Python Imaging Library) and it’s Fork Pillow; They help in advanced image processing along with Tkinter. Tkinter's limitations can be improved with the use of various extensions that facilitate development on top of Tkinter.We shall go over Tkinter's fundamental idea in this extensive guide.s, the types of widgets that are part of the toolkit, layout management, event handling, and advanced features. The exploration of Tkinter in this guide through practical examples and explanations will instill a more detailed understanding of the park of Tkinter and its effective use for GUI development in Python. Tkinter is the most commonly used library to create GUIs in Python. It comes pre-installed with Python, making it a popular option for developers of all skill levels when it comes to building simple utility applications and functional desktop software with graphical interfaces.

Getting Started with Tkinter

The first thing you need to do in order to start using Tkinter is to import the Tkinter module into your Python script. Generally, if When you install Python, Tkinter is also installed the standard library, so it is not additional installed. The import statement is simple: import tkinter as tk. This is standard practice since it facilitates reading the code when you are working with Tkinter elements. For more complex projects, you may also need to import certain modules or classes from Tkinter, which you can, do like this: from tkinter import ttk is great for themed widgets.After importing Tkinter, In each Tkinter application, the initial step is to generate a rudimentary window (known as the "root" window). We achieve this by instantiating the Tk class: root = tk. Tk().title() sets the window title, and geometry()

defines the window's size.After you have created and configured the root window you will normally start to add widgets to it and configure event bindings, finishing by starting the main event loop with root. Mainloop (). This command initiates the loop of Tkinter events, maintaining the window open to listen for and process events like user input until the window is closed. Event Loop: The event loop is vital for GUI applications to keep them responsive, it continuously checks for events and invokes the callback functions when an action is performed.

Here's our first example, a simple "Hello World" application in Tkinter:

Tkinter is imported as tk

```
# Create the root window
root = tk.Tk()
root.title("Hello World")
root.geometry("300x200")

# Add a label widget
label = tk.Label(root, text="Hello, Tkinter World!")
label.pack(pady=20)

# Start the event loop
root.mainloop()
```

Here is the simplest example that generates a window with a single text label saying “Hello, Tkinter World!”. Where pack() method aligns the label on the window, pady=20 gives some vertical padding. When you run this script, a new window will open that displays the greeting text, and it will stay open until you close it, showing you the basic skeleton of a Tkinterapplication.Core concepts, which include widgets, geometry managers, and event handling, are crucial to understanding if you want to work with Tkinter successfully. Widgets are GUI-building blocksA widget is anything that is part of a GUI: buttons, labels, text fields, etc. Geometry managers (pack, grid, place) control how those widgets are arranged in their containers. Your software may react to user actions, such as keyboard or mouse clicks, thanks to event handlinginputs.

```
import tkinter as tk
```

```
def button_click():
label.config(text="Button clicked!")

root = tk.Tk()
root.title("Button Example")
root.geometry("300x200")

label = tk.Label(root, text="Click the button below")
label.pack(pady=20)

button = tk.Button(root, text="Click Me", command=button_click)
button.pack(pady=10)
root.mainloop()
```

In this case, we've attached a button that, with click, it'll modify the label's text. The Button widget takes a command parameter that indicates what function to call when this button is clicked. Tkinter is event-driven and runs your button_click function when you click the button.For more advanced applications, you may want to structure your graphical user interface (GUI) using frames, which are essentially containers for other widgets. It is easier to manage layouts and better organize complex user interfaces through the use of frames. Here is an example using frames:

```
import tkinter as tk

root = tk.Tk()
root.title("Frame Example")
root.geometry("400x300")

# Create a frame for the top section
top_frame = tk.Frame(root, bg="lightblue")
top_frame.pack(fill=tk.X, padx=10, pady=10)

# Add widgets to the top frame
label1 = tk.Label(top_frame, text="Top Frame Content",
bg="lightblue")
label1.pack(pady=5)

# Create a frame for the bottom section
```

```
bottom_frame = tk.Frame(root, bg="lightgreen")
bottom_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

# Add widgets to the bottom frame
label2 = tk.Label(bottom_frame, text="Bottom Frame Content", bg="lightgreen")
label2.pack(pady=5)

button = tk.Button(bottom_frame, text="Click Me")
button.pack(pady=5)

root.mainloop()
```

This example creates 2 frames, one at the top goes light blue, the one at the bottom goes light green. The top frame uses fill=tk. top frame: use X to go expand horizontally fill=tk. BOTH, expand=True to fill available space horizontally and vertically. Instead, they implemented the hierarchical structure using frames to achieve more structured and flexible layouts.However, once you get more familiar with Tkinter you'll realize that it has a lot more set of widgets and features. The various widgets range from simple buttons and labels to ever more complex options like text editors, canvases, and listboxes, and make up a rich toolset for creating all manner of GUI apps. Thus, when paired with the ease of use and readability that Python provides, Tkinter opens the door wide for functional and interactive desktop applications.

Core Tkinter Widgets

A wide variety of widgets are available in Tkinter for creating graphical user interfaces. Widgets are the fundamental interactive building blocks for an application, its strengths and weaknesses are important to be considered while GUI development. Here are some of the basic widgets that the foundation of Tkinterapplications.Label widget is one of the simplest and most used widgets in Tkinter. It is mainly used for rendering text or images to the user. The default type of controls is non-interactive controls, which do nothing when clicked or interacted with. They can also be styled with different fonts, colors, and borders to improve the aesthetics of your application and be made dynamic (show current state of your application).

```
import tkinter as tk

root = tk.Tk()
root.title("Label Examples")

# Simple text label
label1 = tk.Label(root, text="Hello, Tkinter!")
label1.pack(pady=10)

# Label with custom font and color
label2 = tk.Label(root, text="Styled Label",
                  font=("Arial", 16, "bold"),
fg="blue",
bg="lightgray")
label2.pack(pady=10)

# Label with an image
# First, create a PhotoImage object
image = tk.PhotoImage(file="python_logo.png")
# Then create a label with the image
image_label = tk.Label(root, image=image)
image_label.image = image  # Keep a reference to prevent garbage collection
image_label.pack(pady=10)

root.mainloop()
```

We made three different kinds of labels in this example: a simple text label, a label with custom styling, and an image label. The font parameter indicates the font family, size, and style. The fg (foreground) and bg (background) options determine the colors of the text and background, respectively. In the image label, we utilizing the PhotoImage class, create an object that contains a picture file to be linked to the label's image parameter. Notably, we maintain a reference to the image so it doesn't get garbage collected.The Button widget is an interactive thing that is invoked when clicked. It is an essential building block of any GUI application for user interaction. Buttons can present textual content, image content, or both, and work with minimum a couple of visible properties such as background,

border, coloration, and shadow. For Buttons, the most important parameter is command, which tells the Button what function it should call when it is clicked.

```
import tkinter as tk

def button1_clicked():
result_label.config(text="Button 1 was clicked!")

def button2_clicked():
result_label.config(text="Button 2 was clicked!")

root = tk.Tk()
root.title("Button Examples")

# Simple button
button1 = tk.Button(root, text="Click Me",
command=button1_clicked)
button1.pack(pady=10)

# Button with custom styling
button2 = tk.Button(root, text="Styled Button",
        font=("Arial", 12),
bg="lightblue",
fg="navy",
padx=10,
pady=5,
        command=button2_clicked)
button2.pack(pady=10)

# Label to display result
result_label = tk.Label(root, text="Click a button")
result_label.pack(pady=20)

root.mainloop()
```

This is an example of having two buttons, one of which has a different style and is associated with a different function. The buttons update a label, indicating which button was clicked. The padx and pady arguments are for padding around the buttons; they just make

the buttons prettier.import all of the contents from tkinter root = Tk() # constructing a new Tk instance Entry(root).pack() # add Entry widget to Tk instance and pack it root.mainloop() # create a mainloopthat waits for user interaction The Entry widget allows users to enter content in a simple one-line format. The element is very much similar to the input element, it's used for any potential use case where you need to get information in text form like forms, search bars, etc. You support different operations on the Entry widget, like inserting text, deleting text and getting the text, etc.

```
import tkinter as tk
def submit():
input_text = entry.get()
result_label.config(text=f"You entered: {input_text}")

root = tk.Tk()
root.title("Entry Widget Example")

# Label for the entry
label = tk.Label(root, text="Enter your name:")
label.pack(pady=(10, 0))

# Entry widget
entry = tk.Entry(root, width=30)
entry.pack(pady=5)
entry.focus()  # Set focus to the entry widget

# Submit button
submit_button = tk.Button(root, text="Submit", command=submit)
submit_button.pack(pady=5)

# Label to display result
result_label = tk.Label(root, text="")
result_label.pack(pady=10)

root.mainloop()
```

An Entry widget is then displayed for entering the name, a label and a button for submission. Following When a user presses the submit button after typing text, the text is entered gets retrieved using the

get() method and displayed in the label for the outcome. The input is added using the focus() method with typing focus to Entry widget.A Text widget is used to create a multi-line text area to display or edit text. This is an example of the Text widget which will allow you to display text that may become much larger and that can also be formatted, unlike the Entry widget which is limited to one line. It can be beneficial for text editors, loggers or any other application that should show or edit large models of text content.

```
import tkinter as tk

def save_text():
    content = text_area.get("1.0", tk.END)
result_label.config(text=f"Saved text ({len(content)-1} characters)")

root = tk.Tk()
root.title("Text Widget Example")

# Label for the text area
label = tk.Label(root, text="Enter your notes:")
label.pack(pady=(10, 0))

# Text widget
text_area = tk.Text(root, width=40, height=10)
text_area.pack(pady=5)

# Save button
save_button = tk.Button(root, text="Save", command=save_text)
save_button.pack(pady=5)

# Label to display result
result_label = tk.Label(root, text="")
result_label.pack(pady=10)

root.mainloop()
```

We make a Text widget in this example to take notes and a save button to handle the text that is entered. The get() method of Text widgets requires start and end indices. The first line, character 0 in the above example is denoted as "1.0" where tk. END (End of

text)[2] The -1 in the character count computation is for the new line character included by get().Checkbutton widget: This tool is utilized to select a boolean value, it can be selected (on) or unselected (off). For choices that can be turned on or off individually, we use Checkbuttons. They may be associated with Tkinter variables, which offer a very handy way to monitor their status.

```
import tkinter as tk

def show_selection():
    result = ""
    if var1.get():
        result += "Option 1 selected\n"
    if var2.get():
        result += "Option 2 selected\n"
    if var3.get():
        result += "Option 3 selected\n"

result_label.config(text=result if result else "No options selected")

root = tk.Tk()
root.title("Checkbutton Example")

# IntVar objects to store the state of each checkbutton
var1 = tk.IntVar()
var2 = tk.IntVar()
var3 = tk
```

3.4 Basic Widgets (Button, Label, Entry, Frame, Menu)

Widgets are simple elements that you can build into GUIs in your code. They act as interactive components that enable users to enter data, view information, and traverse through applications. Some widgets are frequently used include Buttons, Labels, Entries, Frames, and Menus. This is the case for nearly every GUI application, regardless of the programming language or framework.In this article, we will look into these five types of basic widgets viz, their properties, methods and where do we use them. We will show examples in several languages and frameworks to give you a comprehensive overview of how these widgets work across different platforms. These foundational building blocks form the backbone of

whatever you create, be it desktop applications, web interfaces, or mobile apps.

Understanding GUI Widgets

Before diving into specific widgets, we need to understand what we mean by widgets in GUI contexts. Widgets, also known as controls (or components), are interface element that a user works with to complete something in an application. They combine graphical look and action, offering any common interface for subjecting records to programmatic treatment.The five widgets that we will be looking towards in this guide are the most basic building blocks of any GUI that one makes:

1. Buttons - Interactive elements that trigger actions when clicked
2. Labels - These are static text displays used to present information to the user
3. Entries - The best prescriptive prompt for this new agent would be something like the following: — User Interface Components
4. Frames — Elements for containing widgets to better organize them
5. Menus – A system of hierarchical navigation that gives access to commands and options

These widgets are found in virtually every GUI framework and provide the minimal functionality needed to create an interactive application. These five fundamental aspects of GUI development will form a strong foundation that has applications for more advanced interface design.So, let's dive deep into each widget type, their properties, methods, event handling, and practical applications. The examples will feature in popular GUI frameworks such as Tkinter (Python), JavaFX, HTML/CSS/JavaScript, Qt and others to illustrate the universal principles behind these widgets.

Buttons

Buttons are most likely among the most basic interactive widgets in GUI programming. It stands for clickables that can invoke an action when you click on it. Buttons abound in software UIs, from basic forms to complex apps..

Button Properties

However, across frameworks, most button implementations have certain properties in common:

- Text/Label — The text on the button
- Size – Width dimensions and height
- STATE — Enabled/disabled status
- Style – Visual properties (colours, borders, etc.)
- Command / Action – Function / Method called on button click

Button Events

Generally buttons listen to the below events.

- On Click/Press – When the button gets clicked/pressed
- Hover - When the cursor is pointed to the button
- Focus – When the button gains the keyboard focus

Example 1: Basic Button in Python with Tkinter

```
import tkinter as tk

# Create main window
root = tk.Tk()
root.title("Basic Button Example")
root.geometry("300x200")

# Function to handle button click
def button_clicked():
print("Button was clicked!")
result_label.config(text="Button clicked!")

# Create a button
my_button = tk.Button(root, text="Click Me!",
                      command=button_clicked,
bg="lightblue",
fg="navy",
padx=20,
pady=10)
my_button.pack(pady=30)

# Create a label to show result
result_label = tk.Label(root, text="")
result_label.pack()
```

```
# Start the main loop
root.mainloop()
```

We directly changed label text when a button is pressed in this simple example of Tkinter. The button has customized colors and padding to make it look better.

Example 2: Button in HTML/JavaScript

```
<!DOCTYPE html>
<html>
<head>
<title>Basic Button Example</title>
<style>
.custom-button {
        background-color: #4CAF50;
color: white;
        padding: 10px 20px;
        border: none;
        border-radius: 4px;
        cursor: pointer;
        font-size: 16px;
        transition: background-color 0.3s;
    }

.custom-button:hover {
        background-color: #45a049;
    }

.result {
        margin-top: 20px;
        font-family: Arial, sans-serif;
    }
</style>
</head>
<body>
<h2>Button Example</h2>

<button class="custom-button" id="myButton">Click Me!</button>
```

```
<div id="result" class="result"></div>

<script>
document.getElementById("myButton").addEventListener("click",
function() {
document.getElementById("result").textContent = "Button was
clicked!";
console.log("Button clicked event triggered");
    });
</script>
</body>
</html>
```

This HTML/JavaScript example demonstrates a styled button that replaces the text inside a div element after being clicked. Shows also hover effects with pure CSS.

Example 3: Button in Java with JavaFX

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class ButtonExample extends Application {

  @Override
  public void start(Stage primaryStage) {
    // Create button
    Button button = new Button("Click Me!");
button.setStyle("-fx-background-color: #6495ED; -fx-text-fill:
white;");

    // Create label for result
    Label resultLabel = new Label("");

    // Add action for button click
button.setOnAction(event -> {
```

```
resultLabel.setText("Button was clicked!");
System.out.println("Button click detected");
        });

        // Create layout
VBox root = new VBox(20);
root.setPadding(new Insets(30));
root.getChildren().addAll(button, resultLabel);

        // Create scene
        Scene scene = new Scene(root, 300, 200);

        // Configure and show stage
primaryStage.setTitle("JavaFX Button Example");
primaryStage.setScene(scene);
primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Not only do we use a custom style for the button, but we also set up an action event handler using a lambda expression.

Button Best Practices

1. Ascertainably Clear Paneling – Apply short and action-oriented content on buttons (for instance, use “Save” as opposed to “Click here to save”)
2. Visual Feedback - Add visual feedback if buttons are pressed or hover over
3. Consistent Styling – Keep the look and feel of your buttons similar in the entire app
4. Appropriate sizing — make buttons large enough to be click-able
5. Disabled States – Make it clear when buttons aren’t available

Buttons are by far the most important widget, since they are the primary means by which users cause actions in applications.

Labels

Labels are non-editable widgets that are used to show text or images. They deliver context and information to users, without requiring input. Functionality labels are critical for developing concise, comprehensible user interfaces that assist users in navigating through the applications.

Label Properties

Some common properties of label widgets are:

- Text - The text content shown by the label
- Font – Typeface, size, and style
- Text alignment – Horizontal and vertical alignment
- Icon/Image – Optional icon/image
- Wrapping – Text wrapping behavior

Unlike buttons, labels do not usually respond to user interactions, although certain frameworks provide limited interactivity with labels.

Example 4: Multi-styled Labels in Python/Tkinter

```
import tkinter as tk
from tkinter import font

# Create main window
root = tk.Tk()
root.title("Label Examples")
root.geometry("400x300")
root.configure(bg="#f0f0f0")

# Create a custom font
heading_font = font.Font(family="Helvetica", size=16, weight="bold")

# Header label
header_label = tk.Label(
   root,
   text="Customer Information",
   font=heading_font,
bg="#4285F4",
fg="white",
padx=10,
pady=10,
   width=30
```

```
)
header_label.pack(pady=(20, 30))

# Name label with left alignment
name_label = tk.Label(
    root,
    text="Name: John Smith",
    font=("Arial", 12),
bg="white",
padx=15,
pady=8,
    anchor="w",
    width=25,
    relief=tk.RIDGE
)
name_label.pack(pady=5)

# Email label with left alignment
email_label = tk.Label(
    root,
    text="Email: john.smith@example.com",
    font=("Arial", 12),
bg="white",
padx=15,
pady=8,
    anchor="w",
    width=25,
    relief=tk.RIDGE
)
email_label.pack(pady=5)

# Status label with custom colors
status_label = tk.Label(
    root,
    text="Status: Active",
    font=("Arial", 12, "bold"),
bg="#4CAF50",
fg="white",
```

```
padx=15,
pady=8,
   width=25
)
status_label.pack(pady=20)

root.mainloop()
```

This example demonstrates various label styles in Tkinter, including different fonts, colors, alignments, and border effects.

Example 5: HTML/CSS Labels

```
<!DOCTYPE html>
<html>
<head>
<title>Label Examples</title>
<style>
      body {
         font-family: Arial, sans-serif;
         margin: 40px;
         background-color: #f5f5f5;
      }

      .container {
         max-width: 500px;
         margin: 0 auto;
         background-color: white;
         padding: 30px;
         border-radius: 8px;
         box-shadow: 0 2px 10px rgba(0,0,0,0.1);
      }

.header-label {
         background-color: #3498db;
color: white;
         padding: 12px 20px;
border-radius: 4px;
         font-size: 18px;
text-align: center;
         margin-bottom: 25px;
```

```
    }

    .info-label {
      display: block;
      padding: 10px 15px;
      margin-bottom: 12px;
      background-color: #f9f9f9;
      border-left: 4px solid #2ecc71;
    }

.warning-label {
      display: block;
      padding: 10px 15px;
      margin-top: 20px;
      background-color: #fff3cd;
      border-left: 4px solid #ffc107;
color: #856404;
    }
</style>
</head>
<body>
<div class="container">
<div class="header-label">Product Information</div>

<span class="info-label">Product Name: Premium Widget X200</span>
<span class="info-label">Price: $199.99</span>
<span class="info-label">Availability: In Stock</span>

<div class="warning-label">Limited time offer: 20% discount until end of month</div>
</div>
</body>
</html>
```

This HTML/CSS example shows how various label-like elements can be styled on a web page to present information clearly and attractively.

Example 6: SwingUI Labels with Images (Java)

```
import javax.swing.*;
import java.awt.*;

public class LabelExample extends JFrame {

   public LabelExample() {
setTitle("Label Examples");
setSize(400, 300);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // Create panel with padding
JPanel panel = new JPanel();
panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
panel.setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));

      // Simple text label
JLabeltextLabel = new JLabel("Welcome to the Application");
textLabel.setFont(new Font("Arial", Font.BOLD, 16));
textLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
panel.add(textLabel);
panel.add(Box.createRigidArea(new Dimension(0, 20)));

      // HTML formatted label
JLabelhtmlLabel = new JLabel("<html><div style='text-align:
center;'>" +
                        "<span style='color: blue;'>Blue text</span>, "
+
                        "<span style='color: red;'>red text</span> and
" +
                        "<u>underlined text</u>.</div></html>");
htmlLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
panel.add(htmlLabel);
panel.add(Box.createRigidArea(new Dimension(0, 20)));

      // Label with icon
ImageIcon icon = new ImageIcon("icon.png"); // Replace with actual
image path
```

```
JLabeliconLabel = new JLabel("Label with Icon", icon, JLabel.CENTER);
iconLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
panel.add(iconLabel);

      add(panel);
setLocationRelativeTo(null);
   }

   public static void main(String[] args) {
SwingUtilities.invokeLater(() -> {
         new LabelExample().setVisible(true);
      });
   }
}
```

So here is a fancy example for how text labels can be used by doing some formatting and adding image icon.

Label Best Practices

1. Legible Fonts / Font Size – Keep it readable
2. Less is More — Give users right amount of information without overloading them
3. Modular styling - Keep the label styles consistent for the similar types of information
4. Correct Alignment – Arrange labels in manner that establish visual of flow and readability
5. Hello Pupp: Accessibility – Ensure reading labels to all users possible, Infosense everything.

Labels are easy to understand, however, their proper usage could make a huge difference in how the application will be used. Intuitive labels help users navigate interfaces without confusion, minimizing cognitive load and frustration.

Entry Widgets

Entry widgets (alternatively known, among other things, as text fields, input fields, or text boxes) are interactive text fields in which users can input text data. They are a core piece of functionality for things like forms, search, or anywhere that user text input is needed.

Entry Properties

Entry widget propertiesCommon properties of entry widgets include:

- Entry Body – The present value of the entry field
- Width/Size - Number of characters that the field shows
- Placeholder Text – A hint that will appear when the field is empty
- Validation – Rules for acceptable input
- State – Linkable/Unlinkable
- Read-only Mode — If the text is editable

Entry Events

These events are usually triggered by entry widgets:

- Change/Input – When the content changes
- Focus — When the field is gaining/losing focus
- Submit – On Enter/Return press.
- Validation — When an input is validated against validation rules

Example 7: Entry Field with Validation in Tkinter

```
import tkinter as tk
from tkinter import messagebox
import re

# Create main window
root = tk.Tk()
root.title("Entry Field Example")
root.geometry("400x300")
root.configure(bg="#f5f5f5")

# Create a frame for form elements
frame = tk.Frame(root, bg="#f5f5f5", padx=20, pady=20)
frame.pack(expand=True)

# Email validation function
def validate_email(email):
    pattern = r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$"
    return re.match(pattern, email)
```

```
# Form submission function
def submit_form():
   email = email_entry.get()
   if not email:
      messagebox.sh
```

3.5 Event Handling in Tkinter

In graphical user interface (GUI) programming, event handling is a crucial subject that enables applications to respond to users' interactions, including mouse clicks, keyboard presses, and other inputs. Using Tkinter and events to develop graphical user interfaces Simple GUI apps can be made with Tkinter in Python. To Get Better at How Events Work in Tkinter: Tkinter Event HandlingIn response to the events that are triggered in Tkinter, it uses an event-driven programming paradigm, meaning that the application expects an event to occur, and it then implements related handler functions. They would range from clicking a button, resizing a window, scrolling the screen, etc. The magic of this system is that it is customizable, so you can bind different functions to different events, depending on what The user communicates with!The event system in Tkinter is based on a hierarchy of widgets that are able to register for different types of user input and handle them. An event gets fired, which moves its way around the application and eventually calls the associated callback function. It avoids direct calls to the controller or view from the model and provides a separation of concerns that makes the code more modular, maintainable, and testable.Certainly at its most fundamental level, event handling in Tkinter involves binding methods to events through the bind() method. This associates some event pattern with a callback function which gets called when that specific event happens. Its syntax is even more compact, representing which type of event we want to capture, as a string.The bind takes two parameters: An event descriptor string starting with and a callback function. The event descriptor is made up of specific fields according to a standard format that enumerate the event type, and that might include optional modifiers and details. For instance, " means clicking the left mouse button, as opposed to pressing 'Ctrl+A' on your keyboard, which is described as "binding can get tedious for events common to Tkinter widgets, so higher-level interfaces are provided. Other widgets, such

as buttons, menus, and scales, come with built-in options (command) that will automatically handle some types of events for that specific widget. It makes it easier to create interactive applications without manually binding the function to event.InTkinter programming; we need to understand event propagation as well. Widgets can also be made to bubble up in the widget hierarchy, so parent widgets can handle events that occur on their children. This propagation can be stopped using event methods, such as stopPropagation(), to prevent an event from moving up the hierarchy any further.It will cover the event types, binding ways, and event handling used in Tkinter as an implementation for responsive and interactive Python applications.This section will cover the implementation of basic event binding in Tkinter.It all revolves around binding functions to events. We can do this with the bind() method which associates the callback function to an event type. When the event happens, your callback function executes automatically.All of the widgets in Tkinter have the bind() method with 2 main parameters: event descriptor and a call back function. The event descriptor takes the form of a character string enclosed in angle brackets that identifies the event to react to. The Python program you wish to execute The callback function is invoked when the event occurs.The callback An event object is supplied to the function along with the event details (eg, mouse click coordinates, keyboard press character). The callback calls the event object that gives details about the context of the event so the callback function can make decisions depending on specifics of the event.The simplest example of event binding is binding a function to mouse clicking in a widget. Now, let's make a simple application that listens a left mouse button click on a label:

Tkinter is imported as tk.

```
def handle_click(event):
print(f"Label clicked at coordinates: ({event.x}, {event.y})")

root = tk.Tk()
root.title("Basic Event Binding")
root.geometry("300x200")

label = tk.Label(root, text="Click me!", bg="lightblue", padx=20, pady=20)
```

```
label.pack(expand=True)

label.bind("<Button-1>", handle_click)

root.mainloop()
```

The left mouse button is used to bind the handle_click method to the click event () in this example, which involves creating a basic label widget. We now have a labelled button-like widget where our feature comes in. When someone clicks on the label, the function prints out the coordinates of where a click occurred relative to the widget.Additionally, this is Tkinter's basic event handling pattern. In this manner, we can listen for a variety of events, and tie them to a variety of functions, allowing us to enrich the interactivity of our applications. We will see the beauty of this system when we start dealing with multiple events or creating complex interactions.

Common Event Types in Tkinter

Tkinter handles many types of events; therefore, applications can respond differently to different user interactions. Each of these event types has its own significance, so knowing what they are will help us build responsive applications. Below are some common event types:

1. Mouse Events:

- Click on The mouse button on the left
- Middle mouse button click
- Right mouse button click
- Release left mouse button
- Double left click
- Mouse pointer enters the widget
- When mouse pointer leaves widget
- Mouse movement

2. Keyboard Events:

- Any key press
- Release of any key
- Enter key press
- Space key press
- Escape key press
- Control-Key combination

3. Widget Events:

- Widget size or position change

- Widget received focus
- Widget loses focus
- Change in widget visibility
- Window Events
- Window is being destroyed
- Indicates that window has been mapped (is now visible)
- Window gets unmapped (hidden)

Let's create an example that demonstrates multiple event types:

```
import tkinter as tk
def handle_mouse_enter(event):
event.widget.config(bg="yellow")
status_label.config(text="Mouse entered the button")

def handle_mouse_leave(event):
event.widget.config(bg="lightgray")
status_label.config(text="Mouse left the button")

def handle_left_click(event):
status_label.config(text="Left-clicked the button")

def handle_right_click(event):
status_label.config(text="Right-clicked the button")

def handle_key_press(event):
status_label.config(text=f"Key pressed: {event.char}")

root = tk.Tk()
root.title("Event Types Demo")
root.geometry("400x300")

button = tk.Button(root, text="Interact with me", bg="lightgray", padx=20, pady=10)
button.pack(pady=50)

status_label = tk.Label(root, text="Interaction status will appear here", bd=1, relief=tk.SUNKEN, anchor=tk.W)
status_label.pack(side=tk.BOTTOM, fill=tk.X)
```

```
# Binding multiple events to the button
button.bind("<Enter>", handle_mouse_enter)
button.bind("<Leave>", handle_mouse_leave)
button.bind("<Button-1>", handle_left_click)
button.bind("<Button-3>", handle_right_click)

# Binding keyboard events to the root window
root.bind("<Key>", handle_key_press)

root.mainloop()
```

This illustration shows how to bind multiple events to a widget, how to bind to different buttons on the keyboard, and how to handle keyboard events at the window level. The button has a mouse enter and mouse leave listeners defined, that alter the color of the background of the button and update a status message. The status message is updated on left and right clicks on the button. The key that was pressed is also updated in the status message for any key press on the keyboard.

Event Object Properties

In Tkinter, when an event happens, the callback function is provided with an event object that holds event-related information. This item has many properties that give the context about what caused the event to occur.

The following are some of the most frequently used attributes of the event object:

1. event. widget: The widget where the event was fired
2. event. x and event. y: Widget relative coordinates
3. event. x_root and event. y_root: Co-ordi-nates rel-a-tive to the screen
4. event. char: The character for a keyboard event
5. event. keysym: The key symbol of a keyboard event
6. event. keycode: A keyboard event's key code
7. event. state: State of modifier keys (Shift, Control, etc.)
8. event. width and event. height: new events configure dimensions
9. event. event: The event that occurred—either a trigger or a hook.

Now, lets make an example to show you how to use these event object properties:

```
Tkinter is imported as tk.
def show_event_details(event):
    details = f"""
    Event Type: {event.type}
    Widget: {event.widget}
    Position (widget): ({event.x}, {event.y})
    Position (screen): ({event.x_root}, {event.y_root})
    """

    if hasattr(event, 'char') and event.char:
        details += f"Character: {event.char}\n"

    if hasattr(event, 'keysym') and event.keysym:
        details += f"Key Symbol: {event.keysym}\n"

    if hasattr(event, 'state'):
        details += f"State: {event.state}\n"

    if hasattr(event, 'width') and hasattr(event, 'height'):
        details += f"Size: {event.width} x {event.height}\n"

event_details_label.config(text=details)

root = tk.Tk()
root.title("Event Object Properties")
root.geometry("500x400")

canvas = tk.Canvas(root, bg="lightblue", width=300, height=200)
canvas.pack(pady=20)

event_details_label = tk.Label(root, text="Interact with the canvas to see event details",
                               justify=tk.LEFT, bd=2, relief=tk.GROOVE, padx=10, pady=10)
event_details_label.pack(fill=tk.X, padx=10)
```

```
# Binding different events to the canvas
canvas.bind("<Button-1>", show_event_details)
canvas.bind("<Motion>", show_event_details)
canvas.bind("<Configure>", show_event_details)
canvas.bind("<Key>", show_event_details)

# Make canvas focusable to receive keyboard events
canvas.config(highlightthickness=1)
canvas.focus_set()

root.mainloop()
```

Here we created a canvas widget and bound some events to the widget. When one of these events happen, a show_event_detailsfunction is called that pulls specific pieces from the event object and renders them in a label. In the real-time, we can observe the properties of different event types with this property.

Command Callbacks vs. Bind Method

There are two built-in methods in Tkinter that enable it to handle events by using a command option or a bind() method. Though both serve the purpose of responding to user interactions, they differ in terms of capabilities and use cases.The command option is available only on some widgets (buttons, checkbuttons, menu items, etc.) It takes (a function, without argument) that gets executed when the default trigger action of the widget happens (like; clicking a button). This technique is easier and tends to be more simple in case of simple use-cases.Thebind() method is more flexible though and can be applied to any widget with any event type. The bind callback function works with an event object that holds further details about the occasion. If you are doing more complex interactions then this makes bind more powerful.

Here's an example comparing both approaches:

```
Tkinter is imported as tk
def button_command():
status_label.config(text="Button clicked using command option")

def button_bind(event):
status_label.config(text=f"Button clicked using bind method at ({event.x}, {event.y})")
```

```
root = tk.Tk()
root.title("Command vs. Bind")
root.geometry("400x200")

# Button using command option
cmd_button = tk.Button(root, text="Command Button", command=button_command)
cmd_button.pack(pady=10)

# Button using bind method
bind_button = tk.Button(root, text="Bind Button")
bind_button.pack(pady=10)
bind_button.bind("<Button-1>", button_bind)

status_label = tk.Label(root, text="Click either button", bd=1, relief=tk.SUNKEN, anchor=tk.W)
status_label.pack(side=tk.BOTTOM, fill=tk.X)

root.mainloop()
```

We have made two buttons in this example, one with the command choice and the other with the bind() method. Both buttons update the status label on click, but the bind method provides more detail on where the click took place.

Differences these approaches differ in key ways:

1. Callback parameters: Command callbacks have no parameters, while bind callbacks receive an event object.
2. Specificity on an event: command to the default action of a widget; bind any event type.
3. Simplicity Command is more simple for basic use cases bind is then more flexible and informative.

In stable, simple interactions such as button clicks, using the command option is enough and is more direct. If you want more complex interactions or need details about the event, however, the bind method is your best bet.

Data set Event Modifiers and Virtual Events

Tkinter can let the user be more specify on the event bindings by using modifiers and virtual events. Modifiers are special keys (you

know the likes of Shift, Control, Alt) that can be used in conjunction with other events. Virtual events are events you create to make it easy to handle complex events.

Event Modifiers

Event patterns with modifiers to provide specific bindings. Common modifiers include:

- Control: The key for control
- Alt: Alt key
- Shift: Shift key
- Double: Double-click
- Triple: Triple-click
- Any: Any modifier key

Hyphens separate modifiers from an event name. As an example, means pressing and holding the Control key when clicking the left mouse.button.

Virtual Events

irtual events are special events you define, usually by composing other events together. They have names surrounded by double angle brackets, such as >. This make your code and your logic is easily readable and maintainable because other complex event patterns are abstracted into something meaningful.

Let's create an example demonstrating modifiers and virtual events:

```
Tkinter is imported as tk
def handle_copy(event):
status_label.config(text="Copy action triggered (Ctrl+C)")

def handle_paste(event):
status_label.config(text="Paste action triggered (Ctrl+V)")

def handle_custom_event(event):
status_label.config(text="Custom event triggered")

root = tk.Tk()
root.title("Event Modifiers and Virtual Events")
root.geometry("400x300")

text_entry = tk.Entry(root, width=30)
text_entry.pack(pady=20)
```

```
trigger_button = tk.Button(root, text="Trigger Custom Event")
trigger_button.pack(pady=10)

status_label = tk.Label(root, text="Use Ctrl+C, Ctrl+V, or the button",
bd=1, relief=tk.SUNKEN, anchor=tk.W)
status_label.pack(side=tk.BOTTOM, fill=tk.X)

# Binding events with modifiers
text_entry.bind("<Control-c>", handle_copy)
text_entry.bind("<Control-v>", handle_paste)

# Creating a virtual event
root.event_add("<<CustomEvent>>", "<Control-t>", "<Button-3>")

# Binding the virtual event
text_entry.bind("<<CustomEvent>>", handle_custom_event)
trigger_button.bind("<Button-1>", lambda e:
root.event_generate("<<CustomEvent>>"))
root.mainloop()
```

In this case, we needed to create bindings for Ctrl+C and Ctrl+V modifier combinations on a text entry widget. We have created a virtual event called > which can be triggered either by using Ctrl+T or right-clicking. Note: You can trigger this custom event programmatically using the event_generate() method when the button is clicked.This is especially useful if you want to expose event listeners for triggering the same action in multiple ways, such as providing keyboard shortcuts as well as menu commands for common operations.

Event Binding Levels

Tkinter enables event binding on multiple levels such as on a widget level on a class level or even at an application-wide level. This allows for a more hierarchical approach to handling events across the application.

1. Specific to widgets binding: It binds an event to a widget instance.
2. Class type: Binds an event to all widgets of a particular class.

3. Application-wide binding: Binds an event to a root window and applies it to the entire application.

Event bindings at higher levels can be over

MCQs:

1. **Which library is used in Python for MySQL database connectivity?**
   a) sqlite3
   b) mysql.connector
   c) pandas
   d) numpy
2. **Which of the following is NOT a valid SQL operation?**
   a) CREATE
   b) READ
   c) UPDATE
   d) DELETE
3. **Which command is used to create a new table in SQL?**
   a) NEW TABLE
   b) CREATE TABLE
   c) ADD TABLE
   d) INSERT TABLE
4. **Which module is used for SQLite database connectivity in Python?**
   a) sqlite
   b) sqlite3
   c) pysql
   d) sqlalchemy
5. **Which of the following is a valid Tkinter widget?**
   a) Button
   b) Frame
   c) Label
   d) All of the above
6. **Which function is used to start the Tkinter main event loop?**
   a) window.start()
   b) root.mainloop()
   c) tk.start()
   d) run.loop()

7. **Which method is used to insert data into a database table using Python?**
   a) execute("INSERT INTO ...")
   b) run("INSERT INTO ...")
   c) commit("INSERT INTO ...")
   d) push("INSERT INTO ...")
8. **Which widget is used in Tkinter for a single-line text input field?**
   a) Label
   b) Text
   c) Entry
   d) Frame
9. **What is event handling in a GUI application?**
   a) Displaying only text
   b) Managing user interactions like button clicks
   c) Running SQL queries
   d) None of the above
10. **Which Tkinter function is used to close a window?**
    a) root.quit()
    b) window.close()
    c) app.exit()
    d) tk.stop()

Short Questions:

1. What is SQLite, and how is it different from MySQL?
2. Explain CRUD operations in a database with examples.
3. How do you connect Python to MySQL using mysql.connector?
4. Write a Python program to create a database and a table using SQLite.
5. What is Tkinter, and why is it used?
6. Explain the difference between Entry and Label widgets in Tkinter.
7. How do you create and display a button in a Tkinter window?
8. What is event handling in a GUI application, and why is it important?
9. Write a Python program to create a simple login form using Tkinter.

10. How do you fetch data from a database and display it in a GUI application?

Long Questions:

1. Explain how to connect Python with MySQL and SQLite with examples.
2. Write a Python program to perform CRUD operations in SQLite.
3. Discuss the difference between SQLite and MySQL in terms of usage and performance.
4. Write a Tkinter program to create a simple calculator using buttons and labels.
5. Explain different Tkinter widgets (Button, Label, Entry, Frame, Menu) with examples.
6. Write a Python program to fetch data from a database and display it in a Tkinter window.
7. Discuss the importance of GUI applications in Python and their real-world applications.
8. Explain the event handling mechanism in Tkinter with an example program.
9. Write a Python program to create a student management system with a database and GUI.
10. Discuss the advantages and disadvantages of using Tkinter for GUI development.

References

Chapter 1: Python Basics

1. Lutz, M. (2023). Learning Python (6th ed.). O'Reilly Media.
2. Matthes, E. (2022). Python Crash Course (3rd ed.). No Starch Press.
3. Sweigart, A. (2023). Automate the Boring Stuff with Python (3rd ed.). No Starch Press.
4. Beazley, D., & Jones, B. K. (2022). Python Cookbook (4th ed.). O'Reilly Media.
5. Ramalho, L. (2023). Fluent Python (3rd ed.). O'Reilly Media.

Chapter 2: Data Handling & Libraries

1. McKinney, W. (2023). Python for Data Analysis (3rd ed.). O'Reilly Media.
2. VanderPlas, J. (2022). Python Data Science Handbook (2nd ed.). O'Reilly Media.
3. Harris, C. R., Millman, K. J., & van der Walt, S. J. (2023). NumPy: The Complete Manual. Packt Publishing.
4. Rougier, N. P. (2022). From Python to NumPy. Zenodo.
5. Hunter, J. D., & Dale, D. (2023). Matplotlib: Visualization with Python (2nd ed.). O'Reilly Media.

Chapter 3: Database and GUI

1. Allen, G. (2022). SQLite Python: Database for Beginners. Apress.
2. Harrison, M. (2023). Tkinter GUI Programming by Example. Packt Publishing.
3. Phillips, D. (2022). Python 3 Object-Oriented Programming (4th ed.). Packt Publishing.
4. Vasiliev, A. (2023). Python Database Programming. O'Reilly Media.
5. Chaudhary, B. (2022). Tkinter GUI Application Development Cookbook. Packt Publishing.

MATS UNIVERSITY

MATS CENTER FOR OPEN & DISTANCE EDUCATION

UNIVERSITY CAMPUS : Aarang Kharora Highway, Aarang, Raipur, CG, 493 441

RAIPUR CAMPUS: MATS Tower, Pandri, Raipur, CG, 492 002

T : 0771 4078994, 95, 96, 98 M : 9109951184, 9755199381 Toll Free : 1800 123 819999

eMail : admissions@matsuniversity.ac.in Website : www.matsodl.com