PERSONAL EXPENSE TRACKER

A PROJECT REPORT

In partial fulfillment of the requirements for the award of the degree

BTECH

Under the guidance of

SOUMILI KUNDU

under

Academy of Skill Development

ASD
Skills For Employment

Submitted by

AYANTIKA BARDHAN

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY (KIIT)
Deemed to be University U/S 3 of UGC Act, 1956

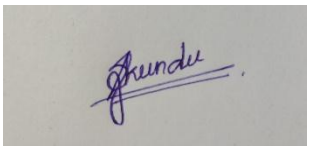
KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY,

BHUBANESWAR

Certificate from the Mentor

This is to certify that **Ayantika Bardhan** has completed the project titled **"Personal Expense Tracker"** under my supervision during the period from **May, 2025** to **July, 2025** which is in partial fulfillment of requirements for the award of the **B.Tech** and submitted to **"Academy of Skills Development"**.

ACADEMY OF SKILL DEVELOPMENT
KOLKATA



Signature of the Mentor

Date: 23 June, 2025

Acknowledgment

I take this opportunity to express my deep gratitude and sincerest thanks to my project mentor, **Soumili Kundu** for giving the most valuable suggestions, helpful guidance, and encouragement in the execution of this project work.

I would like to give a special mention to my colleagues. Last but not least I am grateful to all the faculty members of the **Academy of Skill Development** for their support.

Project Version Control History: -

Version	Author	Description of Version	Date Completed
Final	Ayantika Barshan	Project Report	15th may,2022

Contents

- Python Data Types
- Data Type Conversion
- Python Functions
- Python Modules
- Python Packages
- Commenting in Python
- Graphical User Interface
- Introduction to Tkinter

- Advantages of Tkinter
- Disadvantages of Tkinter
- Definitions Related to Tkinter GUI
- Geometry Managers in Tkinter
- Tkinter Widgets
- Tkinter Themed Widgets
- Widgets Used in this Project
- Creating Root Window and Adding Widgets in it
- Tkinter Button Widget
- Tkinter Label Widget
- Tkinter Entry Widget
- Tkinter Combobox Widget
- Tkinter Textbox Widget
- Actual Codes Used in the Project

- **The Actual Views of the Tkinter Windows**
- **Conclusion**
- **Future Scope of the Project**
- **Thank You**

Overview

Python is an interpreted object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding; make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed. Often, programmers fall in love with Python because of the increased productivity it provides. Since there is no compilation step, the edit-test-debug cycle is incredibly fast.

History of Python

The programming language Python was conceived in the late 1980s, and its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands as a successor to ABC capable of exception handling and interfacing with the Amoeba operating system. Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community, Benevolent Dictator for Life (BDFL). However, van Rossum stepped down as leader on July 12, 2018. Python was

named for the BBC TV show **Monty Python's Flying Circus**.

Python 2.0 was released on October 16, 2000, with many major new features, including a cycle-detecting garbage collector (in addition to reference counting) for memory management and support for Unicode. However, the most important change was to the development process itself, with a shift to a more transparent and community-backed process.

Python 3.0, a major, backwards-incompatible release, was released on December 3, 2008 after a long period of testing. Many of its major features have also been back ported to the backwards-compatible, while by now unsupported, Python 2.6 and 2.7.

Features of Python

Some of the most useful as well as powerful features of Python Language are given below: -

1) **High-Level Language**: - Python is a high-level language. When we write programs in python, we do not need to remember the system architecture, nor do we need to manage the memory.

2) **Interpreted Language**: - Python is an Interpreted Language because Python code is executed line by line at a time. Like other languages C, C++, Java, etc. there is no need to compile python code this makes it easier to debug our code. The source code of python is converted into an immediate form called byte code.

3) **Object-Oriented Language:** - One of the key features of python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, objects encapsulation, etc.

4) **Free and Open Source:** - Python language is freely available at the official website and you can download it from there. Since it is open-source, this means that source code is also available to the public. So you can download it as, use it as well as share it.

5) **Python is a Beginner's Language:** -Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to www browsers to games.

6) **Easy to code:** - Python is a high-level programming language. Python is very easy to learn the language as compared to other languages like C, C#, JavaScript, Java, etc. It is very easy to code in python language and anybody can learn python basics in a few hours or days. It is also a developer-friendly language.

7) **Extensible features:** - Python is an Extensible language. We can write us some Python code into C or C++ language and also we can compile that code in C/C++ language.

8) **Python is Portable language:** - Python language is also a portable language. For example, if we have python code for windows and if we want to run this code on other platforms such as Linux, UNIX, and Mac then we do not need to change it, we can run this code on any platform.

9) **GUI Programming Support:** - Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in python. PyQt5 is the most popular option for creating graphical apps with Python.

10) **Large Standard Library** - Python has a large standard library which provides a rich set of module and functions so you do not have to write your own code for every single thing. There are many libraries present in python for such as regular expressions, unit-testing, web browsers, etc.

11) **Dynamically Typed Language: -** Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable**.**

12) **Python is interactive: -** When a Python statement is entered, and is followed by the Return key, if appropriate, the result will be printed on the screen, immediately, in the next line. This is particularly advantageous in the debugging process.

Environmental Setup

Python is available on a wide variety of platforms including Linux and Mac OS X. Let's understand how to set up our Python environment.

Open a terminal window and type "python" to find out if it is already installed and which version is installed.

- UNIX (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, etc.)
- Win 9x/NT/2000
- Macintosh (Intel, PPC, 68K)
- OS/2
- DOS (multiple versions)
- PalmOS
- Nokia mobile phones
- Windows CE
- Acorn/RISC OS
- BeOS
- Amiga
- VMS/OpenVMS
- QNX
- VxWorks
- Psion
- Python has also been ported to the Java and .NET virtual machines

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python

You can download Python documentation. The documentation is available in HTML, PDF, and PostScript formats.

Installing Python

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python.

If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation.

Here is a quick overview of installing Python on various platforms –

1) UNIX and Linux Installation: -

Here are the simple steps to install Python on Unix/Linux machine.

- Open a Web browser and go to https://www.python.org/downloads/.
- Follow the link to download zipped source code available for Unix/Linux.
- Download and extract files.
- Editing the Modules/Setup file if you want to customize some options.
- run ./configure script
- make
- make install

This installs Python at standard location /usr/local/bin and its libraries at /usr/local/lib/pythonXX where XX is the version of Python.

2) Windows Installation: -

Here are the steps to install Python on Windows machine.

- Open a Web browser and go to https://www.python.org/downloads/.
- Follow the link for the Windows installer python-XYZ.msi file where XYZ is the version you need to install.
- To use this installer python-XYZ.msi, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.

- Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you are done.

3) Macintosh Installation: -

Recent Macs come with Python installed, but it may be several years out of date. See http://www.python.org/download/mac/ for instructions on getting the current version along with extra tools to support development on the Mac. For older Mac OS's before Mac OS X 10.3 (released in 2003), MacPython is available.

Jack Jansen maintains it and you can have full access to the entire documentation at his website − http://www.cwi.nl/~jack/macpython.html. You can find complete installation details for Mac OS installation.

Setting up Python Path

Programs and other executable files can be in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.

The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains information available to the command shell and other programs.

The path variable is named as PATH in UNIX or Path in Windows (UNIX is case sensitive; Windows is not).

In Mac OS, the installer handles the path details. To invoke the Python interpreter from any particular directory, you must add the Python directory to your path.

1) Setting path at Unix/Linux: -

To add the Python directory to the path for a particular session in UNIX −

- In the csh shell − type setenv PATH "$PATH:/usr/local/bin/python" and press Enter.
- In the bash shell (Linux) − type export PATH="$PATH:/usr/local/bin/python" and press Enter.
- In the sh or ksh shell − type PATH="$PATH:/usr/local/bin/python" and press Enter.

- Note − /usr/local/bin/python is the path of the Python directory.

2) Setting path at Windows: -

To add the Python directory to the path for a particular session in Windows −

At the command prompt − type path %path%;C:\Python and press Enter.

Note − C:\Python is the path of the Python directory

Python Environment Variables

Here are important environment variables, which can be recognized by Python –

Sl. No.	Variable & Description
1	**PYTHONPATH** It has a role similar to PATH. This variable tells the Python interpreter where to locate the module files imported into a program. It should include the Python source library directory and the directories containing Python source code. PYTHONPATH is sometimes preset by the Python installer.
2	**PYTHONSTARTUP** It contains the path of an initialization file containing Python source code. It is executed every time you start the interpreter. It is named as .pythonrc.py in Unix and it contains commands that load utilities or modify PYTHONPATH.
3	**PYTHONCASEOK** It is used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it.
4	**PYTHONHOME** It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy.

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter 'A' to 'Z' or 'a' to 'z' or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Salary** and **salary** are two different identifiers in Python.

Here are naming conventions for Python identifiers –

1) Identifiers can be combination of uppercase and lowercase letters, digits or an underscore (_). So myVariable, variable_1, variable_for_print all are valid python identifiers.

2) An Identifier cannot start with digit. So while variable1 is valid, 1variable is not valid.

3) We can't use special symbols like '!','#','@','%','$' etc in our Identifier.

4) Identifier can be of any length.

5) Class names start with an uppercase letter. All other identifiers start with a lowercase letter.

6) Starting an identifier with a single leading underscore indicates the identifier is private.

7) Starting an identifier with double leading underscore indicates the identifier is strictly private.

8) If the identifier starts and ends with two underscores, than means the identifier is language-defined special name.

9) While c = 10 is valid, writing count = 10 would make more sense and it would be easier to figure out what it does even when you look at your code after a long time.

10) Multiple words can be separated using an underscore, for example, this_is_a_variable.

Python Keywords

There are 35 keywords in Python. If we type help('keywords') in the Python IDLE, it will show all the available keywords in Python. These keywords are tabulated below as given in Python Documentation.

Table of all Keywords

False	None	True	and	as
assert	async	await	break	class
continue	def	del	elif	else
except	finally	for	from	global
if	import	in	is	lambda
nonlocal	not	or	pass	raise
return	try	while	with	yield

Mandatory Indentation in Python

- **Indentation** refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability purpose only, the indentation system in Python Language is very important.
- **Guido van Rossum** believes that using indentation for grouping is extremely elegant and contributes a lot to the clarity of the average Python program. Most people learn to love this feature after awhile.
- Since there are no begin/end brackets there cannot be a disagreement between grouping perceived by the parser and the human reader.
- Python uses indentation to indicate a block of code.
- In most other programming languages, indentation is used only to help make the code look pretty.
- But in Python, it is required for indicating what block of code a statement belongs to.
- The amount of indentation matters: A missing or extra space in a Python block could cause an error or unexpected behavior.
- Indentation also helps to get rid of 2 lines containing braces only per control statement, which can save several hundred of lines for any very big program.

Command Line Arguments in Python

With Python being such a popular programming language, as well as having support for most operating systems, it's become widely used to create command line tools for many purposes. These tools can range from simple CLI apps to those that are more complex, like AWS' awscli tool.

Complex tools like this are typically controlled by the user via command line arguments, which allows the user to use specific commands, set options, and more. For example, these options could tell the tool to output additional information, read data from a specified source, or send output to a certain location.

In general, arguments are passed to CLI tools differently, depending on your operating system:

- Unix-like: `-` followed by a letter, like `-h`, or `--` followed by a word, like `--help`
- Windows: `/` followed by either a letter, or word, like `/help`

These different approaches exist due to historical reasons. Many programs on Unix-like systems support both the single and double dash notation. The single dash notation is mostly used with single letter options, while double dashes present a more readable options list, which is particularly useful for complex options that need to be more explicit.

Handling Command Line Arguments with Python

Python 3 supports a number of different ways of handling command line arguments. The built-in way is to use the sys module. In terms of names, and its usage, it relates directly to the C library (libc). The second way is the getopt module, which handles both short and long options, including the evaluation of the parameter values.

Furthermore, two other common methods exist. This is the argparse module, which is derived from the optparse module available up to Python 2.7. The other method is using the docopt module, which is also available.

Many programs can be run to provide you with some basic information about how they should be run. Python enables you to do this with -h −

$ python-h
usage: python [option]...[-c cmd|-m mod | file |-][arg]...

Options and arguments (and corresponding environment variables):

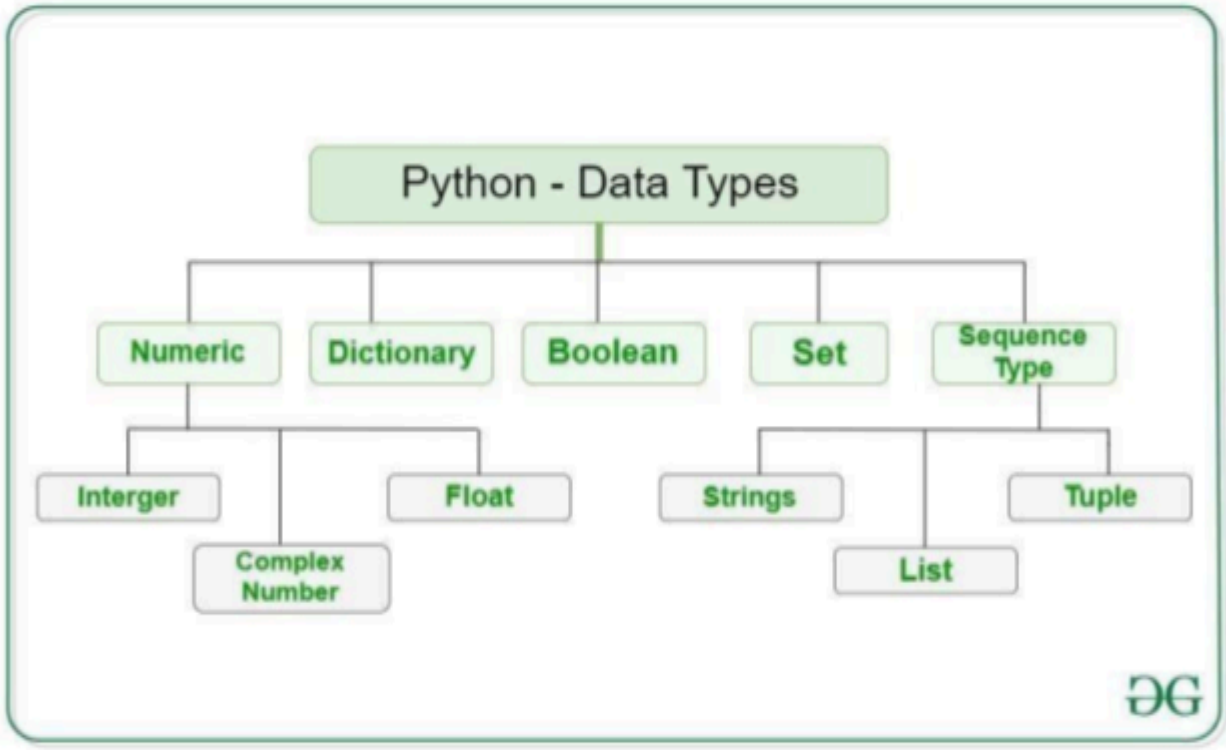
-c cmd: program passed in as string (terminates option list)

-d : debug output from parser (also PYTHONDEBUG=x)

-E : ignore environment variables (such as PYTHONPATH)

-h : print this help message and exit [etc.]

Python Data Types



3.4 Data Type Definitions

Below is a table of all used data types.

Name	Data type	Size bits	Size bytes	Range
char, __int8	signed integer	8	1	– 128 ... 127
BYTE	unsigned integer	8	1	0 ... 256
short	signed integer	16	2	– 32'768 ... 32'767
WORD	unsigned integer	16	2	0 ... 65'535
long	signed integer	32	4	– 2'147'483'648 ... 2'147'483'647
DWORD	unsigned integer	32	4	0 ... 4'294'967'295
BOOL	signed integer	32	4	TRUE = 1 FALSE = 0
HANDLE	pointer to an object	32	4	0 ... 4'294'967'295

Table 2: Data type definitions

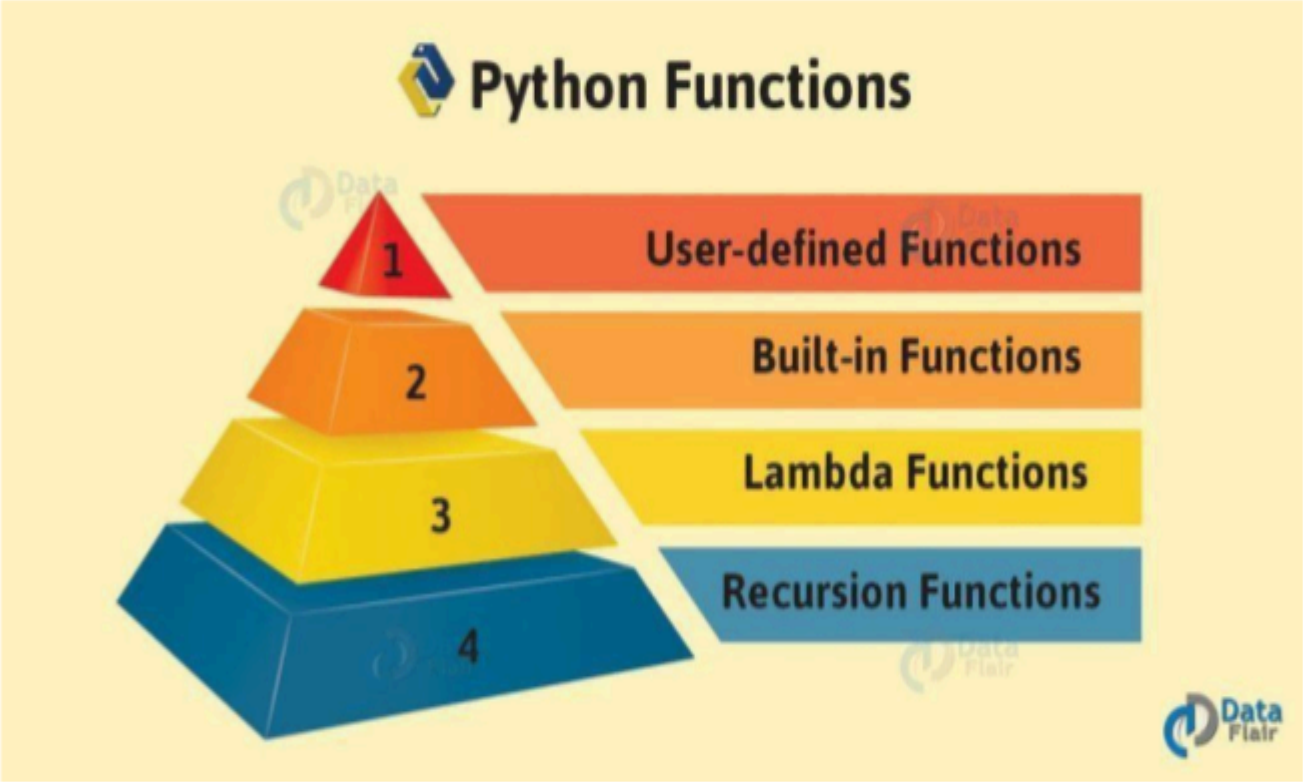
Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another.

Sr.No.	Function & Description
1	**int(x [,base])** Converts x to an integer. base specifies the base if x is a string
2	**long(x [,base])** Converts x to a long integer. base specifies the base if x is a string.
3	**float(x)** Converts x to a floating-point number.
4	**complex(real [,imag])** Creates a complex number.
5	**str(x)** Converts object x to a string representation.
6	**repr(x)** Converts object x to an expression string.
7	**eval(str)** Evaluates a string and returns an object.
8	**tuple(s)** Converts s to a tuple.
9	**list(s)** Converts s to a list.

Python Functions



A function is a set of statements that take inputs, do some specific computation and produce output. The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can call the function. Functions that readily come with Python are called built-in functions. Python provides built-in functions like print ( ) , input() etc. but we can also create your own functions. These functions are known as **user defines functions**.

1) Python User-Defined Functions

All the functions that are written by any us comes under the category of user defined functions. Below are the steps for writing user defined functions in Python.

- In Python, **"def"** keyword is used to declare user defined functions.
- An indented block of statements follows the function name and arguments which contains the body of the function.

The constructional structure of an user-defined Python function is: -

def <function_name>([<parameters>]):

    <statement(s)>

The components of the definition are explained in the table below:

Sl. No.	Components	Meaning
1	def	The keyword that informs Python that a function is being defined.
2	<function_name>	A valid Python identifier that names the function.
3	<parameters>	An optional, comma-separated list of parameters that may be passed to the function.
4	**:**	Punctuation that denotes the end of the Python function header (the name and parameter list).

5	<statement(s)>	A block of valid Python statements. It is also called the body of the function. The body is a block of statements that will be executed when the function is called. The body of a Python function is defined by indentation in accordance with the off-side rule.

There are 5 different aspects of user-defined functions as follows:

1) Parameterized Functions: - In a parameterized function, one or more of the details of what the function does are defined as parameters instead of being defined in the function; they have to be passed in by the calling code. The function may take arguments(s) also called parameters as input within the opening and closing parentheses, just after the function name followed by a colon. For example,

def myfunc(num1,num2,…numn):

    <statements to be executed>

2) Functions with Default Arguments: - Python has a different way of representing syntax and default values for function arguments. Default values indicate that the function argument will take that value if no argument value is passed during function call. The default value is assigned by using assignment (=) operator. For example,

```
def myFun(x, y = 50):
    print("x = ", x)
    print("y = ", y)
myFun(10)        # This is the driver code
```

This code will give the output like this:
x = 10
y = 50

Here, only one parameter is passed and it's assigned to x, and the value of y is assigned as its default value passed in the function declaration. Again, if we call the function like this:

```
def myFun(x, y = 50):
    print("x = ", x)
    print("y = ", y)
myFun(10,20)        # This is also the driver code
```

This code will give the output like this:
x = 10
y = 20

Because values of both the parameters are passed, so the default value is no longer required or valid.

3) Functions with Keyword Arguments: - When we call a function with some values, these values get assigned to the arguments according to their position. According to Wikipedia, in computer programming, named parameters, named argument or keyword arguments refer to a computer language's support for function calls that clearly state the name of each parameter within the function call. Python allows functions to be called using **keyword arguments**. When we call functions in this way, the order (position) of the arguments can be changed. For example,

```
def myFun(x,y):
        print("x = ", x)
        print("y = ", y)
myFun(10,20)        # This is also the driver code
```

This code will give the output like this:

```
x = 10
y = 20
```

```
def myFun(x,y):
        print("x = ", x)
        print("y = ", y)
myFun(y=20,x=10)        # This is also the driver code
```

This code will also give the same output like the previous one:

```
x = 10
y = 20
```

4) Functions with Variable Length Arguments: - We can have both normal and keyword variable number of arguments. Here, we have called the function with multiple arguments. These arguments get wrapped up into a tuple before being passed into the function. Inside the function, we use a for loop to retrieve all the arguments back.

- The special syntax *args in function definitions in Python is used to pass a variable number of arguments to a function. It is used to pass a non-keyworded, variable-length argument list.
- The special syntax **kwargs in function definitions in python is used to pass a keyworded, variable-length argument list. We use the name kwargs with the double star. The reason is because the double star allows us to pass through keyword arguments (and any number of them).

For example of *agrs, this can be illustrated by the below example: -

```
def greet(*names):          # This function greets all the person in the names tuple
   # names is a tuple with arguments
   for name in names:
      print("Hello", name)

greet("Monica", "Luke", "Steve", "John")          # This is the driving code
```

This program will produce the following output: -

```
Hello Monica
Hello Luke
Hello Steve
Hello John
```

Similarly, **kwargs can be illustrated as bellow: -

```
def intro(**data):
   print("\nData type of argument:",type(data))

   for key, value in data.items():
      print("{} is {}".format(key,value))

intro(Firstname="John",    Lastname="Wood",
Email="johnwood@nomail.com", Country="Wakanda", Age=25,
Phone=9876543210)
```

The output of the program is: -

```
Data type of argument: <class 'dict'>
Firstname is John
Lastname is Wood
Email is johnwood@nomail.com
Country is Wakanda
Age is 25
Phone is 9876543210
```

5) **Pass by Reference and Pass by Value: -** One important thing to note is, in Python every variable name is a reference. When we pass a variable to a function, a new reference to the object is created. Parameter passing in Python is same as reference passing in Java. To confirm this Python's built-in id() function is used in below example: -

```
def myFun(x):
    print("Value received:", x, "id:", id(x))

x = 12  # This is the Driver code
print("Value passed:", x, "id:", id(x))
myFun(x)
```

The output of the above code is: -

```
Value passed: 12 id: 10853984
Value received: 12 id: 10853984
```

However, if the value of the above variable is changed inside a function, then it will create a different variable as a number which is immutable. However, if a mutable list object is modified inside the function, the changes are reflected outside the function also. For example,

```
def myFun(x, arr):
        print("Inside function")
        # Changing integer will Also change the reference to the variable
        x += 10
        print("Value received", x, "Id", id(x))

        # Modifying mutable objects
        # will also be reflected outside
        # the function
        arr[0] = 0
        print("List received", arr, "Id", id(arr))

# Driver code
x = 10
arr = [1, 2, 3]

print("Before calling function")
print("Value passed", x, "Id", id(x))
print("Array passed", arr, "Id", id(arr))
print()

myFun(x, arr)

print("\nAfter calling function")
print("Value passed", x, "Id", id(x))
print("Array passed", arr, "Id", id(arr))
```

The output of the above program will be like this:-

Before calling function
Value passed 10 Id
10853920
Array passed [1, 2, 3] Id 139773681420488

Inside function
Value received 20 Id 10854240
List received [0, 2, 3] Id 139773681420488

After calling function
Value passed 10 Id 10853920
Array passed [0, 2, 3] Id 139773681420488

2) Python Built-in Functions

The Python built-in functions are defined as the functions whose functionality is pre-defined in Python. The python interpreter has several functions that are always present for use. These functions are known as Built-in Functions. The main advantages of built-in functions are they are pre-defined in Python, so we are free to use them in our program or code wherever we want and whenever we want. As they are previously defined in Python, so programmers can easily use them freely in there code, even without knowing the actual code or logic written inside the function, programmers all have to know how and where to use them, that's all. There is huge number of built-in functions defined in Python, which can be by default accessible by any programmer. Some of them can be listed as given in the below table: -

Built-in Functions in Python				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

3) Python Lambda Functions

Python allows us to create anonymous function, i.e., function having no names using a facility called the **Lambda function**. Lambda functions are small functions usually not more than a line. It can have any number of arguments just like a normal function. ... Also there is no need for any return statement in lambda function. In Python, a lambda function is a single-line function declared with no name, which can though have any number of arguments, but it can only have one expression. Such a function is capable of behaving similarly to a regular function declared using the Python's def keyword. The usefulness of lambda will be realized when we need a small piece of function that will be run one in a while or just once. Instead of writing the function in global scope or including it as part of our main program, we can toss around few lines of code when needed to a variable or another function. It has the syntax like this:

lambda arguments :

expression For an example,

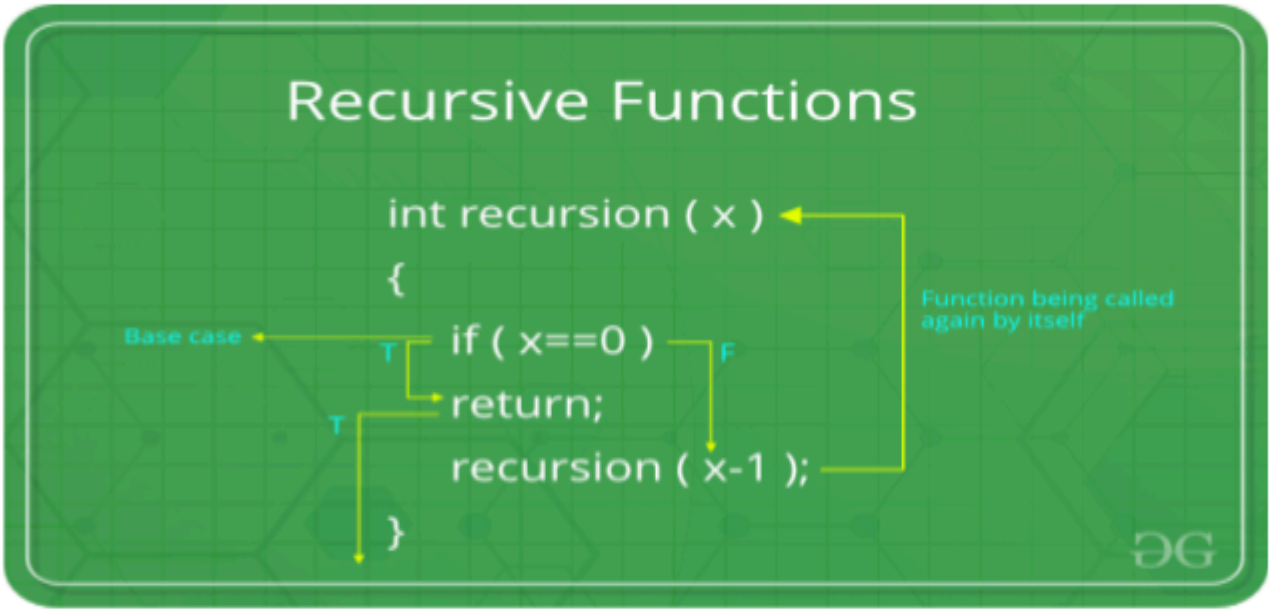
```
x = lambda a, b, c: a + b + c
print(x(5, 6, 12))
```

This is an anonymous function that evaluates the sum of all the numbers that are passed as arguments, i.e. (5+6+12) = 23

The output of this lambda expression will be: -

23

4) Python Recursive Functions



A recursive function is a function that calls itself during its execution. This enables the function to repeat itself several times, outputting the result at the end of each iteration. The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. This can be illustrated by the below example of finding factorial of a number: -

```
def fact(n):
    if n==1:
        return 1
    else:
        return(n*fact(n-1))
print('The factorial is: ',fact(int(input()))) # This is the driver code
```

The output will be like this, suppose user inputs 10:

The factorial is: 3628800

Python Modules

A module allows us to logically organize our Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

The Python code for a module named ‘aname’ normally resides in a file named ‘aname.py’. Here's an example of a simple module, support.py

```
def print_func( par ):
    print("Hello: ", par)
```

The Import Statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The import has the following syntax –

```
import module1[, module2[,… moduleN]
```

Python Packages

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and sub packages and sub-subpackages, and so on.

Consider a file Pots.py available in Phone directory. This file has following line of source code −

```
def Pots():
        print "I'm Pots Phone"
```

Similar way, we have another two files having different functions with the same name as above –

- Phone/Isdn.py file having function Isdn()
- Phone/G3.py file having function G3()

Now, create one more file__init__.py in Phone directory −

- Phone/__init__.py

To make all of your functions available when you've imported Phone, you need to put explicit import statements in__init_.py as follows −

```
from Pots import Pots
from Isdn import Isdn
from G3 import
```

Commenting in Python

When writing code in Python, it’s important to make sure that our code can be easily understood by others. Giving variables obvious names, defining explicit functions, and organizing our code are all great ways to do this. Another awesome and easy way to increase the readability of our code is by using comments!

Comments describe parts of the code where necessary to facilitate the understanding of programmers, including ourselves.

There are few methods for commenting in Python, such as: -

1) To write a single line comment in Python, we have to simply put the hash mark '#' before our desired comment. For example,

This is my comment …

2) Python ignores everything after the hash mark '#' and up to the end of the line. We can insert them anywhere in our code, even in-line with other code. For example,

def fact(n): # This fact() function calculates the factorial of n.

4) For multiline commenting, there are two possible ways:

A) Put '#' mark before all the lines, if possible, for small codes only.
B) To comment an entire block of code or simply statements, start and end the block with either triple single quote ('''...''') or triple double quotes ("""..."""").

For example,

''' This is a a multiline comment''' and """ This is also multiline comment""" both are correct.

Comments are used to increase the readability of the code for multi user as well as single user and used mainly to indicate the functionality of the parts of the code. But one point must be remembered that Comments should be made at the same indent as the code it is commenting. That is, a function definition with no indent would have a comment with no indent, and each indent level following would have comments that are aligned with the code it is commenting.

Graphical User Interface (GUI)

1) Introduction to UI: -

In the industrial design field of human-computer interaction, a **User Interface (UI)** is the space where interactions between humans and machines occur. At the most basic level, the user interface (UI) is the series of screens, pages, and visual elements—like buttons and icons—that enable a person to interact with a product or service.

2) Types of UI: -

There are five main types of user interface:

- Command Line Interface (CLI)
- Graphical User Interface (GUI)
- Menu Driven Interface (MDI)
- Form Based Interface (FBI)
- Natural Language Interface (NLI)

3) Graphical User Interface (GUI): -

A graphical user interface (GUI) is a type of User Interface through which users interact with electronic devices via visual indicator representations such as primary notation, instead of text-based user interfaces, typed command labels or text navigation. It is a process that allows us to point our mouse or cursor to a particular icon and click on it, causing a hidden list of commands to be automatically created for your computer to follow. An example of GUI is clicking on an icon on our desktop to open a particular file.

4) GUI Frameworks in Python: -

Graphical User Interfaces make human-machine interactions easier as well as intuitive. It plays a crucial role as the world is shifting towards digitalization. Python is one of the most loved languages by developers because of its large number of libraries and frameworks. It's a great and interesting task of creating a Python user interface for any application. To proceed with building python GUI we need toolkits. Python has a variety of **Toolkit** or we can say **Python GUI frameworks** that are used create the user interface for mobile applications, web applications, and desktop applications. Here is a list of top 14 GUI toolkit/ framework for Python that software developers use (Hyperlink is inserted)-

- Tkinter GUI
- PyQT GUI
- KIVY GUI
- WxPython GUI
- PySide GUI
- PySimpleGUI
- PyGUI
- Pyforms GUI
- Wax Python GUI
- Libavg
- Pygame
- Pyglet
- PyGTK

For the purpose of this very project, we've used the Tkinter GUI toolkit for the implementation of your project work.

Introduction to Tkinter

The name **Tkinter** comes from **Tk interface**. Tkinter was written by Fredrik Lundh. Tkinter is a Python binding to the Tk GUI toolkit. It is the standard Python interface to the Tk GUI toolkit and is Python's de facto standard GUI Tkinter is included with standard Linux, Microsoft Windows and Mac OS X installs of Python. Tkinter is free software released under a Python license. As with most other modern Tk bindings, Tkinter is implemented as a Python wrapper around a complete Tcl interpreter embedded in the Python interpreter. Tkinter calls are translated into Tcl commands (Tool Command Language) which are fed to this embedded interpreter, thus making it possible to mix Python and Tcl in a single application.

Advantages of Tkinter

As GUI toolkit, Tkinter is very popular for GUI programming for its advantages, some of them can be listed below: -

- Tkinter is Python's default GUI library. It is based on the Tk toolkit, originally designed for the Tool Command Language (Tcl). Due to Tk's popularity, it has been ported to a variety of other scripting languages, including Perl (Perl/Tk), Ruby (Ruby/Tk), and Python (Tkinter).
- The combination of Tk's GUI development portability and flexibility along with the simplicity of a scripting language integrated with the power of systems language gives you the tools to rapidly design and implement a wide variety of commercial-quality GUI applications.
- Python, along with Tkinter, provides a fast and exciting way to build useful applications that would have taken much longer if you had to program directly in C/C++ with the native windowing system's libraries.
- Once you have designed the application and the look and feel that goes along with your program, you will use basic building blocks known as widgets to piece together the desired.
- Once we get Tkinter up on our system, it will take less than 15 minutes to get our first GUI application running.
- Python programs using Tkinter can be very brief, partly because of the power of Python, but also due to Tk. In particular, reasonable default values are defined for many options used in creating a widget, and packing it (i.e., placing and displaying).
- Tk provides widgets on Windows, Macs, and most Unix implementations with very little platform-specific dependence. Some newer GUI frameworks are achieving a degree of platform independence, but it will be some time before they match Tk's in this respect.

Disadvantages of Tkinter

- There is some concern with the speed of Tkinter. Most calls to Tkinter are formatted as a Tcl command (a string) and interpreted by Tcl from where the actual Tk calls are made. This theoretical slowdown caused by the successive execution of two interpreted languages is rarely seen in practice and most real-world.
- Tkinter does not include advanced widgets.
- It has no similar tool as Qt Designer for Tkinter.
- It doesn't have a native look and feel.

Now the question is which one is more dominant, advantages of Tkinter or disadvantages of Tkinter? We're the only person who can answer whether it's worth it to us is ourselves. The answer depends on what our goals and needs are.

If our goal is to learn how to create GUIs, tkinter is arguably one of the best toolkits there is to reach that goal. It's simple and easy to learn, and can provide a fantastic introduction to concepts we must master in order to create graphical desktop applications.

On the other hand, if our goal is to create something like Excel or Photoshop, there are definitely better toolkits out there.

So, from the point of view of wanting to learn how to create GUIs, tkinter is a fantastic way to start. From the point of view of creating a highly polished commercial application, it's not so great.
But one point must be noted that the advantages of Tkinter are far more away than its disadvantages, at least, but not only for beginners and simple GUI applications, but also it's helpful as well useful to design quite complex GUI applications. So, Tkinter obviously worth learning!

Definitions Related to Tkinter GUI

1) Top Level Window: - A top-level form is a window that has no parent form, or whose parent form is the desktop window. Top-level windows are typically used as the main form in an application. The top-level widget is used to create and display the top-level windows which are directly managed by the window manager. The top-level widget may or may not have the parent window on the top of them. The top-level windows have the title bars, borders, and other window decorations.

2) Widgets: - In GUI programming, a top-level root windowing object contains all of the little windowing objects that will be part of your complete GUI application. These can be text labels, buttons, list boxes, etc. These individual little GUI components are known as widgets.

3) Events and Callbacks: - Usually, widgets have some associated behaviors, such as when a button is pressed, or text is filled into a text field. These types of user behaviors are called events, and the GUI's response to such events are known as callbacks.

4) Event-Driven Processing: - Events can include the actual button press (and release), mouse movement, hitting the Return or Enter key, etc. The entire system of events that occurs from the beginning until the end of a GUI application is what drives it. This is known as event-driven processing. • One example of an event with a callback is a simple mouse move. Suppose that the mouse pointer is sitting somewhere on top of your GUI application. If you move the mouse to another part of your application, something has to cause the movement of the mouse to be replicated by the cursor on your screen so that it looks as if it is moving according to the motion of your hand. These are mouse move events that the system must process portray your cursor moving across the window. When you release the mouse, there are no more events to process, so everything just remains idle on the screen again.

Geometry Managers in Tkinter

The GUI application must establish all the GUI components, then draw (a.k.a. render or paint) them to the screen. This is the responsibility of the **Geometry Manager**. When the geometry manager has completed arranging all of the widgets, including the top-level window, GUI applications enter their server-like infinite loop. Tkinter has three built-in layout/ geometry managers:

1) Place Geometry Manager
2) Pack Geometry Manager
3) Grid Geometry Manager

1) **Place Geometry Manager: -** The **Place geometry manager** is the simplest of the three general geometry managers provided in Tkinter. It allows us explicitly set the position and size of a window, either in absolute terms, or relative to another window. We can access the place manager through the place**()** method which is available for all standard widgets. It is usually not a good idea to use place**()** for ordinary window and dialog layouts; it's simply too much work to get things working as they should. Use the pack() or grid() managers for such purposes.

2) **Pack Geometry Manager: -** The Pack geometry manager packs widgets in rows or columns. We can use options like fill, expand, and side to control this geometry manager. Compared to the grid manager, the pack manager is somewhat limited, but it's much easier to use in a few, but quite common situations:

- Put a widget inside a frame (or any other container widget), and have it fill the entire frame.
- Place a number of widgets on top of each other.
- Place a number of widgets side by side.

3) **Grid Geometry Manager: -** The **Grid geometry manager** puts the widgets in a 2-dimensional table. The master widget is split into a number of rows and columns, and each "cell" in the resulting table can hold a widget. The **grid** manager is the most flexible of the geometry managers in Tkinter. If you don't want to learn how and when to use all three managers, you should at least make sure to learn this one.

TKINTER WIDGETS

There are various pre-defined widgets available in Tkinter, they are: -

Sl. No.	Widget names	Description
01.	Buttons	Similar to a Label but provides additional functionality for mouse-hovers, presses, and releases, as well as keyboard activity/events.
02.	Canvas	Provides ability to draw shapes (lines, ovals, polygons, rectangles); can contain images or bitmaps.
03.	Check Buttons	Set of boxes, of which any number can be “checked”
04.	Entry	Single-line text field with which to collect keyboard input.
05.	Frame	Pure container for other widgets.
06.	Label	Used to contain text or images.
07.	Label Frame	Combo of a label and a frame but with extra label attributes.

08.	List Box	Presents the user with a list of choices from which to choose.
09.	Menu	Actual list of choices “hanging” from a Menu button from which the user can choose.
10.	Menu Button	Provides infrastructure to contain menus (pull down, cascading, etc.).
11.	Message	Similar to a Label, but displays multiline text.
12.	Panned Window	A container widget with which you can control other widgets placed within it.
13.	Radio Button	Set of buttons, of which only one can be “pressed”.
14.	Scale	Linear “slider” widget providing an exact value at current setting; with defined starting and ending values.
15.	Scroll Bar	Provides scrolling functionality to supporting widgets, for example, Text, Canvas, List box, and Entry.
16.	Spin Box	Combination of an entry with a button letting you adjusts its value.
17.	Text	Multiline text field with which to collect (or display) text from user.
18.	Top Level	Similar to a Frame, but provides a separate window container.

Tkinter Themed Widgets

- The tkinter**.**ttk module provides access to the **Tk themed widget set**, introduced in Tk 8.5.
- If Python has not been compiled against Tk 8.5, this module can still be accessed if **Tile** has been installed.
- The former method using Tk 8.5 provides additional benefits including anti-aliased font rendering under X11 and window transparency.
- The basic idea for tkinter.ttk is to separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance.
- Tkinter.ttk is used to create modern GUI (Graphical User Interface) applications which cannot be achieved by tkinter itself.
- Ttk comes with 18 widgets, twelve of which already existed in tkinter: Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale, Scrollbar, and Spinbox. The other six are new: Combobox, Notebook, Progressbar, Separator, Sizegrip and Treeview.
- Designing with ttk widgets involves three levels of abstraction: -

  1) A theme is a complete "look and feel", customizing the appearance of all the widgets
  2) A style is the description of the appearance of one kind of widget. Each theme comes with a predefined set of styles, but you can customize the built-in styles or create your own new styles.
  3) Each style is composed of one or more elements. For example, the style of a typical button has four elements: a border around the outside; a focus element that changes color when the widget has input focus; a padding element; and the button's label (text, image, or both).

Widgets Used in This Project

In this project, 5 widgets from Tkinter and Tkinter.ttk have been used. These are listed below: -

1) Button

2) Label

3) Entry

4) Combo Box

5) Text Box

There are many more Tkinter widgets also available, but as per the requirement of this project, the above mentioned 5 basic widgets had been used.

Creating Root Window and Adding Widgets in It

At first, we've to import tkinter. Then, we have to make a top-level window, which is also known as root or parent window and it contains all other widgets as its child class and the root main loop must be run infinitely to encounter any user action and callbacks, e.g., like this: -

```
from tkinter import *
root = tk()
btn = Button(root, option= values)
lbl = Label(root, option= values)
combox = Combobox(root, option= values)
```

```
entry = Entry(root, option= values)
txtbox = Textbox(root, option= values)
root.mainloop()
```

1) Tkinter Button Widget

- **Introduction: -** The Button widget is a standard Tkinter widget, which is used for various kinds of buttons. A button is a widget which is designed for the user to interact with, i.e. if the button is pressed by mouse click some action might be started. They can also contain text and images like labels. While labels can display text in various fonts, a button can only display text in a single font. The text of a button can span not only in only one line but also even more than one line.
  A Python function or method can be associated with a button. This function or method will be executed, if the button is pressed in some way.
  One point should be noted that if we import all from tkinter and then all from tkinter.ttk, then the tkinter.ttk widgets will overwrite the widgets that were imported form tkinter previously.

- **Syntax: -** btn = Button(master, option= values)

- **Parameters: -**

  1) **master: -** This represents the parent window.

  2) **options: −** There is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas. These are listed in the table below: -

Sl. No.	Options	Description
1.	activebackground	Background color when the button is under the cursor.
2.	activeforeground	Foreground color when the button is under the cursor.
3.	bd	Border width in pixels. Default is 2.
4.	bg	Normal background color.
5.	command	Function or method to be called when the button is clicked.
6.	fg	Normal foreground (text) color
7.	font	Text font to be used for the button's label.
8.	height	Height of the button in text lines (for textual buttons) or pixels (for images).
9.	highlightcolor	The color of the focus highlight when the widget has focus.
10.	image	Image to be displayed on the button (instead of text).
11.	justify	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
12.	padx	Additional padding left and right of the text.
13.	pady	Additional padding above and below the text.
14.	relief	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE
15.	state	Set this option to DISABLED to gray out the button and make it unresponsive. Has the value ACTIVE when the mouse is over it. Default is NORMAL.
16.	underline	Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined.
17.	width	Width of the button in letters (if displaying text) or pixels (if displaying an image).
18.	wraplength	If this value is set to a positive number, the text lines will be wrapped to fit within this length.

2) Tkinter Label Widget

- **Introduction: -** This widget implements a display box where we can place text or images. The text displayed by this widget can be updated at any time we want. It is also possible to underline part of the text (like to identify a keyboard shortcut) and span the text across multiple lines.

- **Syntax: -** lbl = Label(master, option= values)

- **Parameters: -**

  1) **master: -** This represents the parent window.

  2) **options: −** There is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas. These are listed in the table below: -

Sl. No	Options	Description
1.	activebackground	Background color when the button is under the cursor.
2.	activeforeground	Foreground color when the button is under the cursor.
3.	bd	Border width in pixels. Default is 2.

4.	bg	Normal background color.
5.	command	Function or method to be called when the button is clicked.
6.	fg	Normal foreground (text) color.
7.	font	Text font to be used for the button's label.
8.	height	Height of the button in text lines (for textual buttons) or pixels (for images).
9.	highlightcolor	The color of the focus highlight when the widget has focus.
10	image	Image to be displayed on the button (instead of text).
11.	justify	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
12.	padx	Additional padding left and right of the text.
13.	pady	Additional padding above and below the text.
14.	relief	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE.
15.	state	Set this option to DISABLED to gray out the button and make it unresponsive. Has the value ACTIVE when the mouse is over it. Default is NORMAL.
16.	underline	Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined.
17.	width	Width of the button in letters (if displaying text) or pixels (if displaying an image).
18.	wraplength	If this value is set to a positive number, the text lines will be wrapped to fit within this length.

3) Tkinter Entry Widget

- **Introduction: -**The Entry widget is used to accept single-line text strings from a user.
  - If we want to display multiple lines of text that can be edited, then we should use the Text widget.
  - If we want to display one or more lines of text that cannot be modified by the user, then we should use the Label widget.

- **Syntax: -** entry = Entry(master, option= values)

- **Parameters: -**

  1) **master: -** This represents the parent window.

  2) **options: −** There is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas. These are listed in the table below: -

Sl. No.	Options	Description
1.	bg	The normal background color displayed behind the label and indicator
2.	bd	The size of the border around the indicator. Default is 2 pixels.
3.	command	A procedure to be called every time the user changes the state of this check button.

4.	cursor	If you set this option to a cursor name (arrow, dot etc.), the mouse cursor will change to that pattern when it is over the check button.

5.	font	The font used for the text.
6.	exportselection	By default, if you select text within an Entry widget, it is automatically exported to the clipboard. To avoid this exportation, use exportselection=0.
7.	fg	The color used to render the text.
8.	highlightcolor	The color of the focus highlight when the check button has the focus.
9.	justify	If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.
10.	relief	With the default value, relief=FLAT, the check button does not stand out from its background. You may set this option to any of the other styles
11.	selectbackground	The background color to use displaying selected text.
12	selectborderwidth	The width of the border to use around selected text. The default is one pixel.
13.	selectforeground	The foreground (text) color of selected text.
14.	show	Normally, the characters that the user types appear in the entry. To make a .password. entry that echoes each character as an asterisk, set show="*".
15.	state	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive. If the cursor is currently over the check button, the state is ACTIVE.
16.	textvariable	In order to be able to retrieve the current text from your entry widget, you must set this option to an instance of the StringVar class.
17.	width	The default width of a check button is determined by the size of the displayed image or text. You can set this option to a number of characters and the check button will always have room for that many characters.

18.	xscrollcommand	If you expect that users will often enter more text than the onscreen size of the widget, you can link your entry widget to a scrollbar.

- **Methods: -** There are various popular methods related to tkinter entry widgets, they are listed below: -

Sl. No.	Methods	Descriptions
1	delete (first, last=None)	Deletes characters from the widget, starting with the one at index first, up to but not including the character at position last. If the second argument is omitted, only the single character at position first is deleted.
2	get()	Returns the entry's current text as a string.
3	icursor (index)	Set the insertion cursor just before the character at the given index.
4	index (index)	Shift the contents of the entry so that the character at the given index is the left most visible character. Has no effect if the text fits entirely within the entry.
5	insert (index, s)	Inserts string s before the character at the given index.
6	select_adjust (index)	This method is used to make sure that the selection includes the character at the specified index.
7	select_clear()	Clears the selection. If there isn't currently a selection, has no effect.
8	select_from (index)	Sets the ANCHOR index position to the character selected by index, and selects that character.
9	select_present()	If there is a selection, returns true, else returns false.
10	select_range (start, end)	Sets the selection under program control. Selects the text starting at the start index, up to but not including the character at the end index. The start position must be before the end position.
11	select_to (index)	Selects all the text from the ANCHOR position up to but not including the character at the given index.
12	xview (index)	This method is useful in linking the Entry widget to a horizontal scrollbar.
13	xview_scroll (number, what)	Used to scroll the entry horizontally. The ‘what’ argument must be either UNITS, to scroll by character widths, or PAGES, to scroll by chunks the size of the entry widget. The number is positive to scroll left to right, negative to scroll right to left.

4) Tkinter Combobox Widget

- **Introduction: -** Combobox widget is a widget that combines an entry with a list of choices available to the use. It's similar to List box in Tkinter.
- **Syntax: -** entry = Entry(master, option= values)

- **Parameters: -**

1) master: - This represents the parent window.

2) options: − There is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas. These are listed in the table below: -

Sl. No.	Options	Description
1.	background	Used to set background color for widget.
2.	borderwidth	Used to draw with border in 3D effects.
3.	font	Used to set font for widget.
4.	foreground	Used to set foreground color for widget.
5.	textvariable	Variable associated with the widget. When the text of widget changes, the variable is set to text of widget.
6.	values	Arbitrary values for combo box widget.
7.	width	Sets the width for widget.
8.	justify	Sets the alignment of text, which can be left, center, or right.

9.	state	Sets the state, which can be read_only, disabled, or normal.
10.	postcommand	Procedure to be executed post action.

5) Tkinter Textbox Widget

- **Introduction: -** Text widgets provide advanced capabilities that allow us to edit a multiline text and format the way it has to be displayed, such as changing its color and font. You can also use elegant structures like tabs and marks to locate specific sections of the text, and apply changes to those areas. Moreover, you can embed windows and images in the text because this widget was designed to handle both plain and formatted text.

- **Syntax: -** entry = Entry(master, option= values)

- **Parameters: -**

  2) master: - This represents the parent window.

  2) options: − There is the list of most commonly used options for this widget. These options can be used as key-value pairs separated by commas. These are listed in the table below: -

Sl. No.	Options	Description
1.	bg	The default background color of the text widget.
2.	bd	The width of the border around the text widget. Default is 2 pixels.
3.	cursor	The cursor that will appear when the mouse is over the text widget.
4.	exportselection	Normally, text selected within a text widget is exported to be the selection in the window manager. Set exportselection=0 if you don't want that behavior.
5.	font	The default font for text inserted into the widget.
6.	fg	The color used for text (and bitmaps) within the widget. You can change the color for tagged regions; this option is just the default.
7.	height	The height of the widget in lines (not pixels!), measured according to the current font size.
8.	highlightbackground	The color of the focus highlight when the text widget does not have focus.
9.	highlightcolor	The color of the focus highlight when the text widget has the focus.
10.	highlightthickness	The thickness of the focus highlight. Default is 1. Set highlightthickness=0 to suppress display of the focus highlight.
11.	insertbackground	The color of the insertion cursor. Default is black.
12.	insertborderwidth	Size of the 3-D border around the insertion cursor. Default is 0.
13.	insertofftime	The number of milliseconds the insertion cursor is off during its blink cycle. Set this option to zero to suppress blinking. Default is 300.

14.	insertontime	The number of milliseconds the insertion cursor is on during its blink cycle. Default is 600.
15.	insertwidth	Width of the insertion cursor (its height is determined by the tallest item in its line). Default is 2 pixels.
16.	padx	The size of the internal padding added to the left and right of the text area. Default is one pixel.
17.	pady	The size of the internal padding added above and below the text area. Default is one pixel.
18.	relief	The 3-D appearance of the text widget. Default is relief=SUNKEN.
19.	selectbackground	The background color to use displaying selected text.
20.	selectborderwidth	The width of the border to use around selected text.
21.	spacing1	This option specifies how much extra vertical space is put above each line of text. If a line wraps, this space is added only before the first line it occupies on the display. Default is 0.
22.	spacing2	This option specifies how much extra vertical space to add between displayed lines of text when a logical line wraps. Default is 0.
23.	spacing3	This option specifies how much extra vertical space is added below each line of text. If a line wraps, this space is added only after the last line it occupies on the display. Default is 0.
24.	state	Normally, text widgets respond to keyboard and mouse events; set state=NORMAL to get this behavior. If you set state=DISABLED, the text widget will not respond, and you won't be able to modify its contents programmatically either.
25.	tabs	This option controls how tab characters position text.

26.	wrap	This option controls the display of lines that are too wide. Set wrap=WORD and it will break the line after the last word that will fit. With the default behavior, wrap=CHAR, any line that gets too long will be broken at any character.
27.	width	The width of the widget in characters (not pixels!), measured according to the current font size.
28.	xscrollcommand	To make the text widget horizontally scrollable, set this option to the set() method of the horizontal scrollbar.
29.	yscrollcommand	To make the text widget vertically scrollable, set this option to the set() method of the vertical scrollbar.

- **Methods: -** Some popular textbox methods are listed below: -

Sl. No.	Methods	Description
1.	delete(startindex [,endindex])	This method deletes a specific character or a range of text.
2.	get(startindex [,endindex])	This method returns a specific character or a range of text.
3.	index(index)	Returns the absolute value of an index based on the given index.
4.	insert(index [,string]...)	This method inserts strings at the specified index location.
5.	see(index)	This method returns true if the text located at the index position is visible.

Actual Codes used in the Project

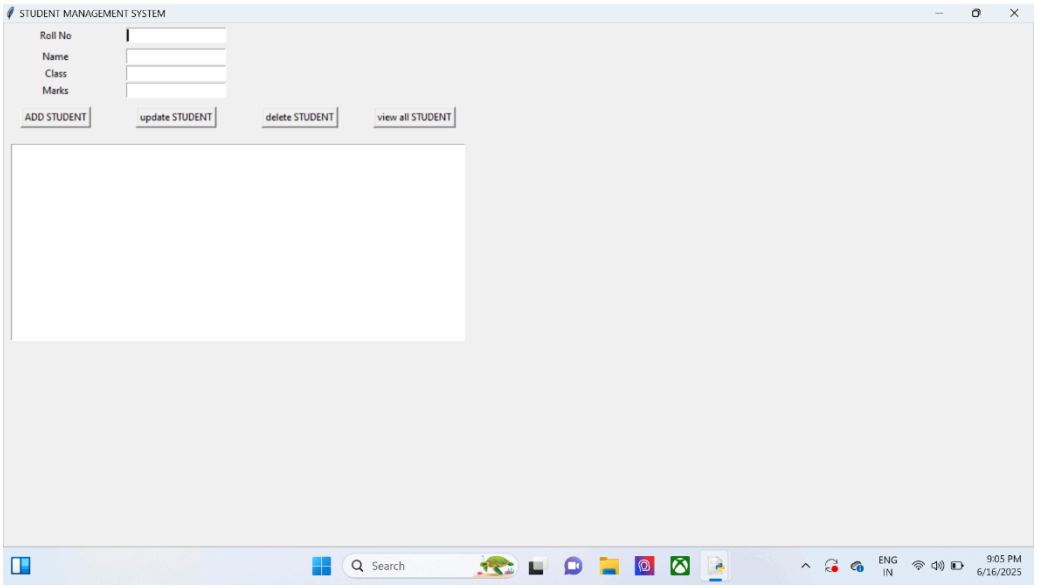
```
import tkinter as tk
from tkinter import messagebox
import sqlite3
#connent  to sqlite and create table
def connect_db():
    conn=sqlite3.connect("students.db")
    cursor=conn.cursor()
    cursor.execute('''
    CREATE TABLE IF NOT EXISTS student(
    roll INTEGER PRIMARY KEY,
    name TEXT,
    class TEXT,
    marks INTEGER
    )
    ''')

    conn.commit()
    conn.close()
def add_student():
    roll= roll_entry.get()
    name=name_entry.get()
    stu_class=class_entry.get()
    marks=marks_entry.get()
    if roll and name and stu_class and marks:
        try:
            conn=sqlite3.connect("students.db")
            cursor=conn.cursor()
            cursor.execute("INSERT INTO student VALUES(?,?,?,?)",(roll,name,stu_class,marks))
            conn.commit()
            conn.close()
            messagebox.showinfo("Success","Student Added")
            view_student()
        except sqlite3.IntegrityError:
            messagebox.showerror("Error","roll number already exist!")
        else:
            messagebox.showwarning("Input Error","All fields required")
def view_student():
    output_text.delete("1.0",tk.END)
    conn=sqlite3.connect("students.db")
    cursor=conn.cursor()
    cursor.execute("SELECT * FROM student")
    rows=cursor.fetchall()
```

```
    conn.close()
    for row in rows:

  output_text.insert(tk.END,f"ROLL:{row[0]},Name: {row[1]},Class:

{row[2]},Marks:{row[3]}\n")
def update_student():
    roll= roll_entry.get()
    name=name_entry.get()
    stu_class=class_entry.get()
    marks=marks_entry.get()
    conn=sqlite3.connect("students.db")
    cursor=conn.cursor()
    cursor.execute("UPDATE student SET name=?, class=?,marks=? WHERE
roll=?",(name,stu_class,marks,roll))
    conn.commit()
    conn.close()
    messagebox.showinfo("success","student updated!")
    view_student()
def delete_student():
    roll= roll_entry.get()
    conn=sqlite3.connect("students.db")
    cursor=conn.cursor()
    cursor.execute("DELETE FROM student WHERE roll=?",(roll,))
    conn.commit()
    conn.close()
    messagebox.showinfo("DELETED","student RECORD DELETED!")
    view_student()
#GUI LAYOUT
root= tk.Tk()
root.title("STUDENT MANAGEMENT SYSTEM")
root.geometry("700x500")
#roll
tk.Label(root,text="Roll No").grid(row=0,column=0,padx=10,pady=5)
roll_entry=tk.Entry(root)
roll_entry.grid(row=0,column=1)
#name
tk.Label(root,text="Name").grid(row=1,column=0)
name_entry=tk.Entry(root)
name_entry.grid(row=1,column=1)
#class
tk.Label(root,text="Class").grid(row=2,column=0)
class_entry=tk.Entry(root)
class_entry.grid(row=2,column=1)
#marks
tk.Label(root,text="Marks").grid(row=3,column=0)
marks_entry=tk.Entry(root)
```

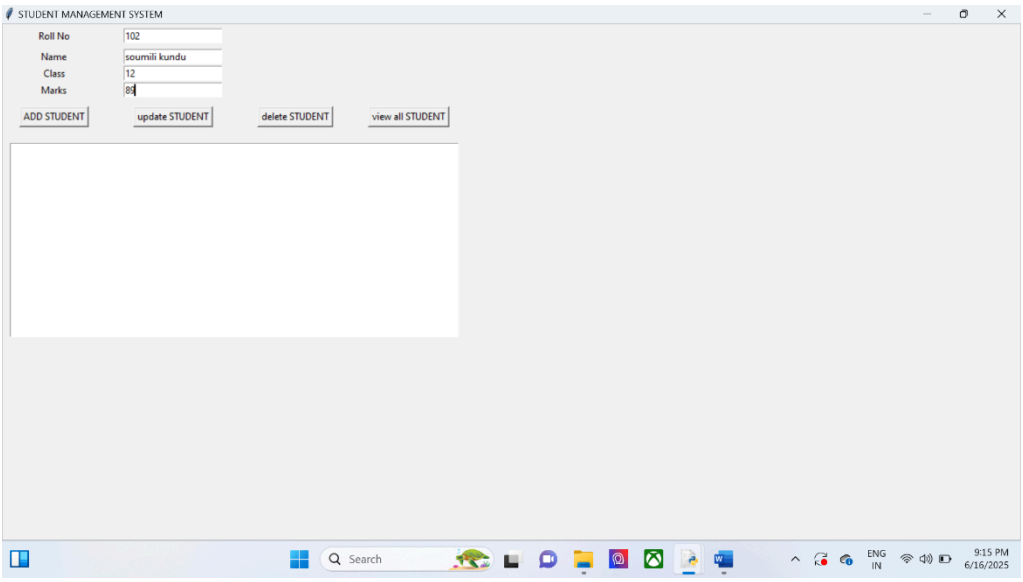
```
marks_entry.grid(row=3,column=1)
#add student button
tk.Button(root,text="ADD STUDENT",command=add_student).grid(row=4,column=0,pady=10)



#update
tk.Button(root,text="update STUDENT",command=update_student).grid(row=4,column=1)
#delete
tk.Button(root,text="delete STUDENT",command=delete_student).grid(row=4,column=2)
#view all
tk.Button(root,text="view all STUDENT",command=view_student).grid(row=4,column=3)
#output display
output_text=tk.Text(root,height=15,width=70)
output_text.grid(row=5,column=0,columnspan=4, padx=10,pady=10)
connect_db()
view_student()
root.mainloop()
```

GUI VIEW

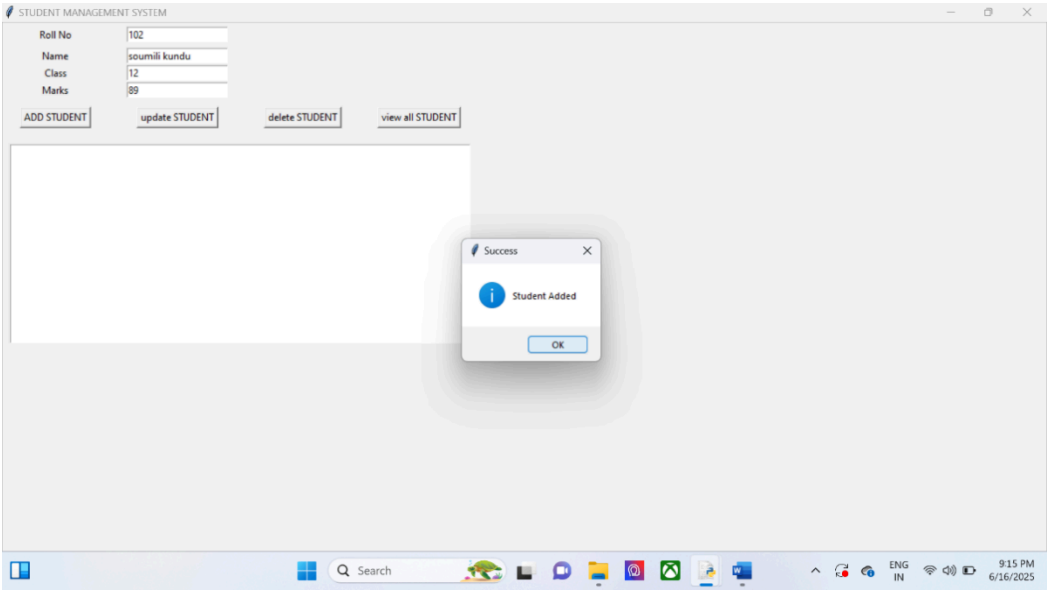
Before inserting any record



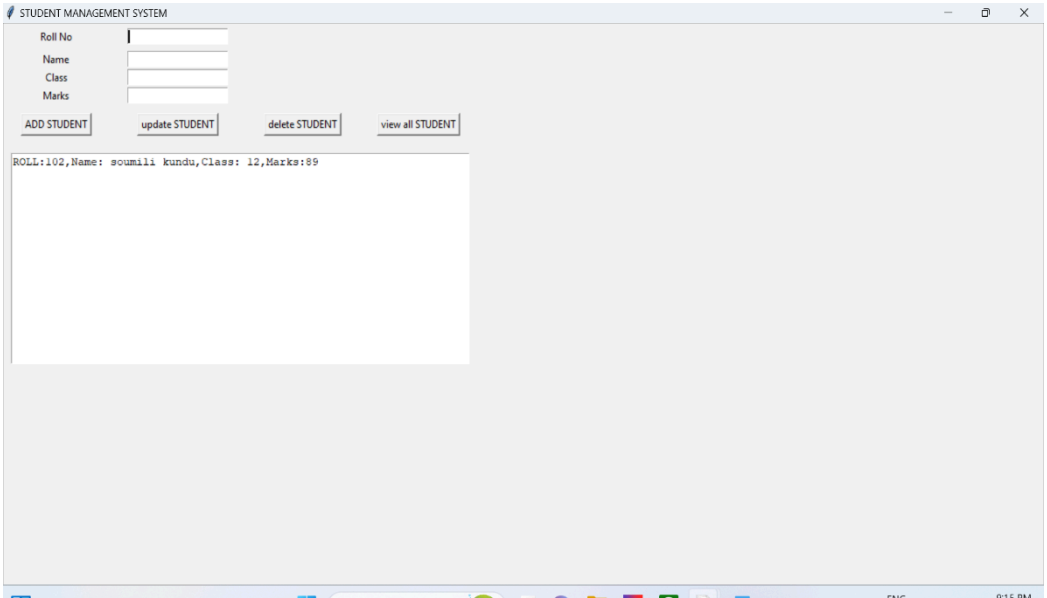
Data entry



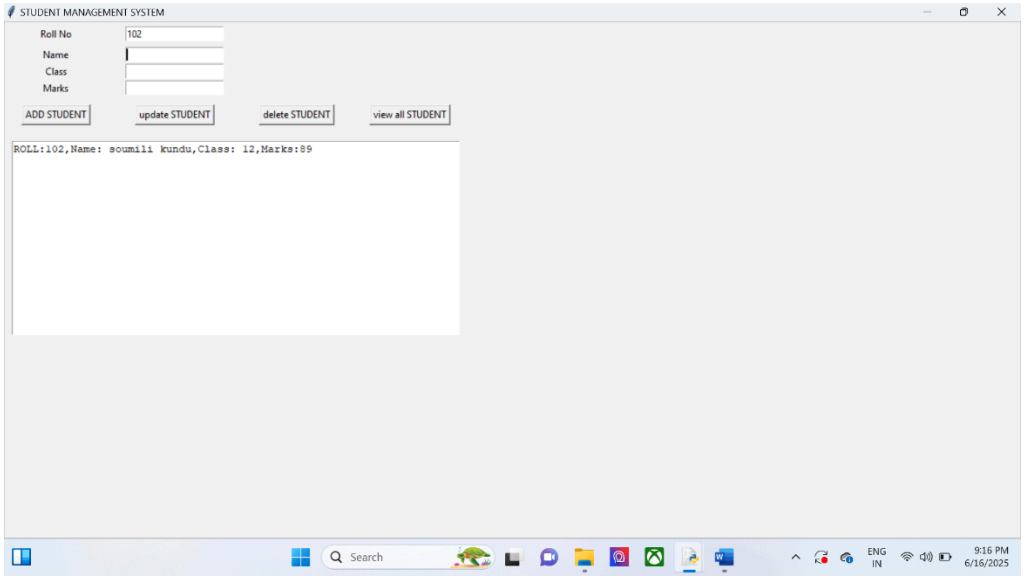
After added student record



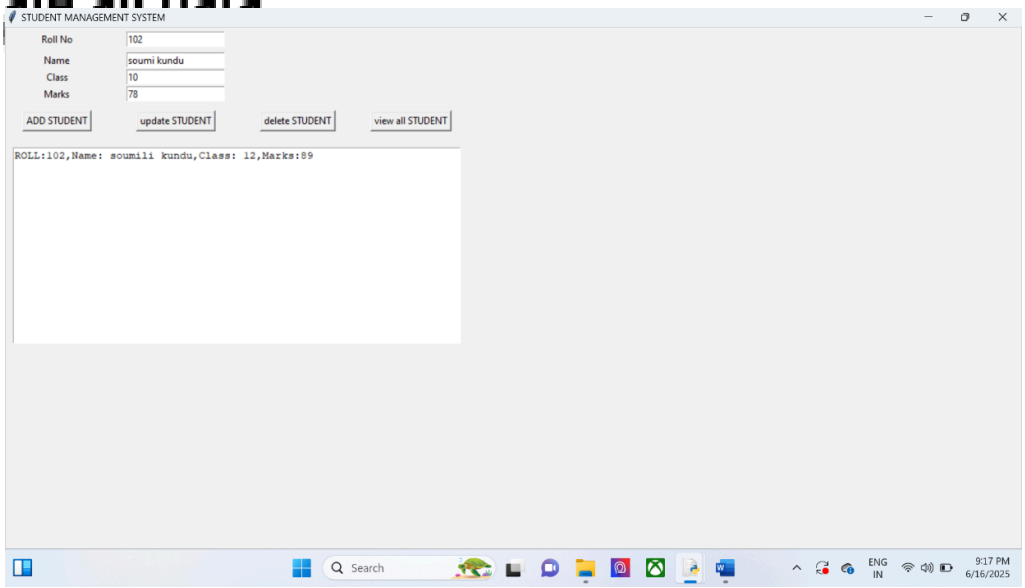
View student record



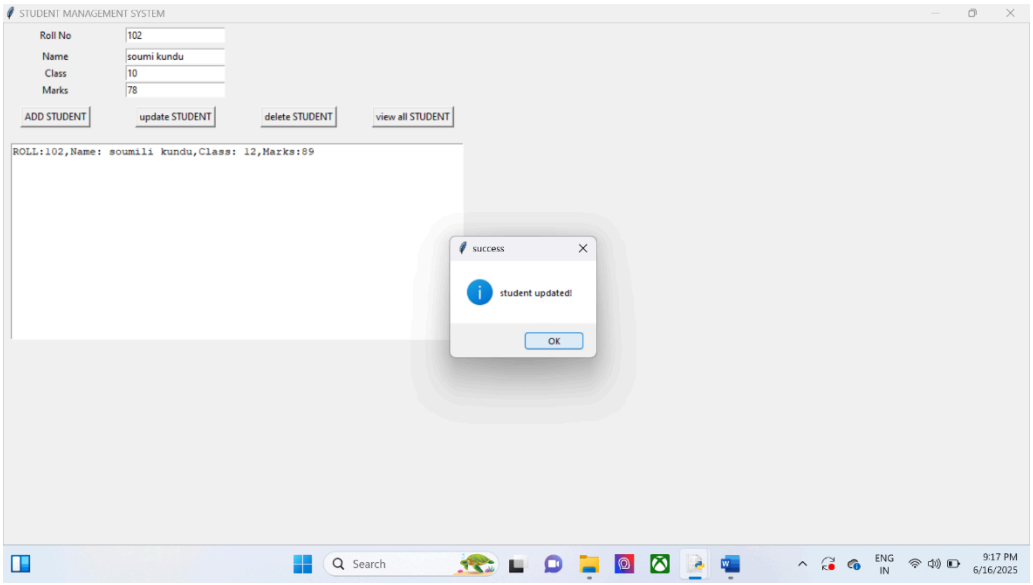
Updated student record by roll number



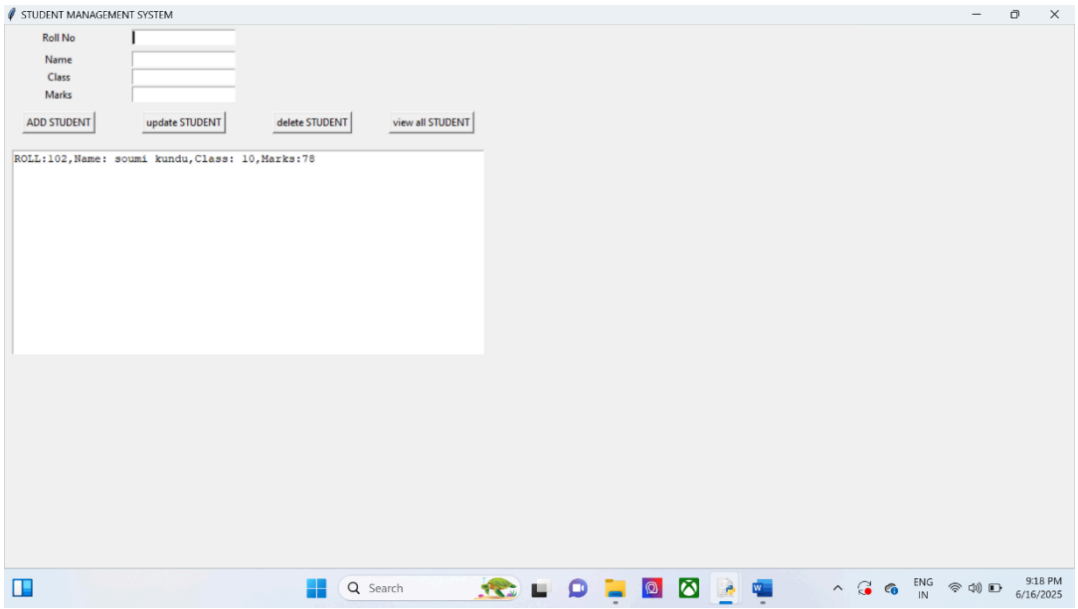
Update all data



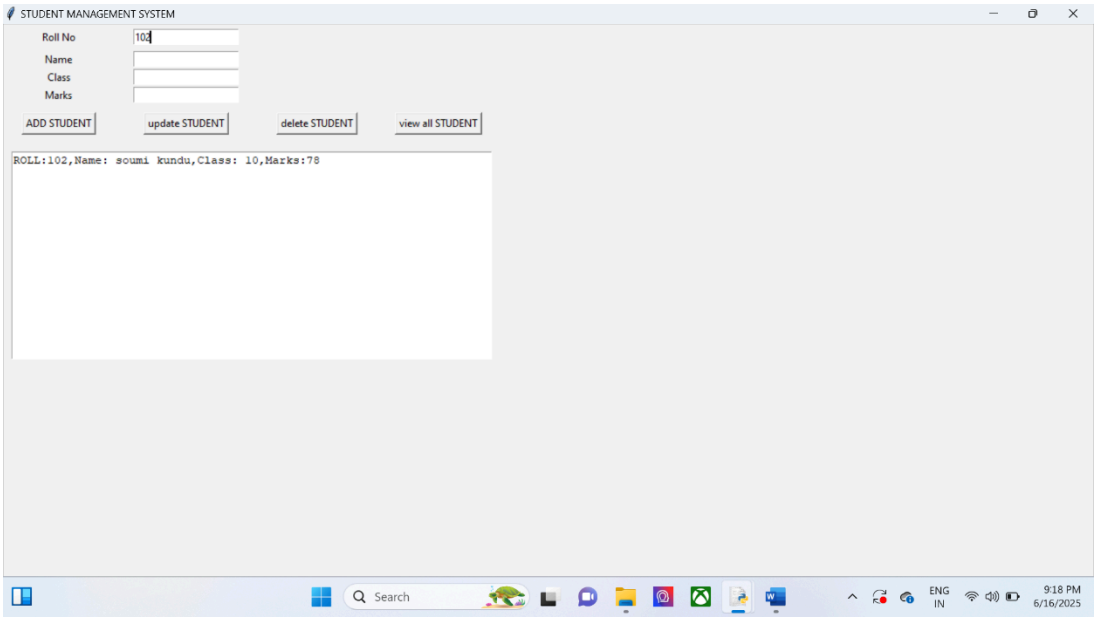
After student records updated



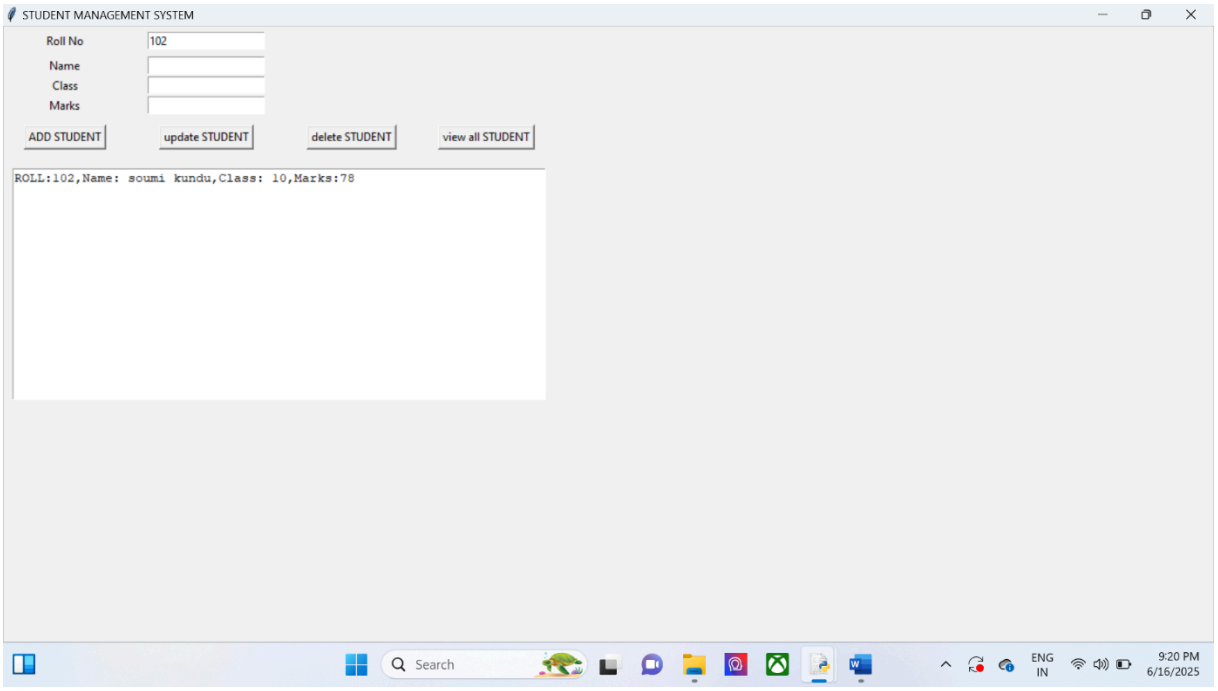
After update all records



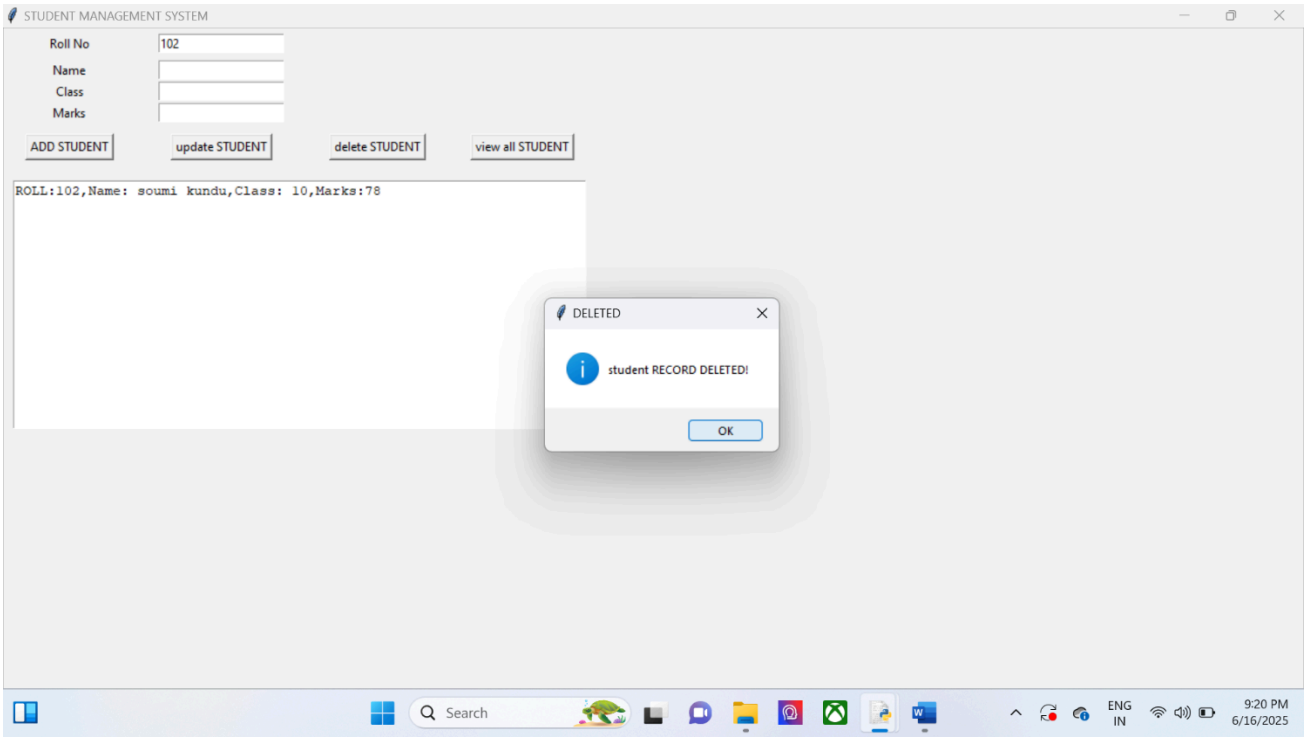
Delete student data



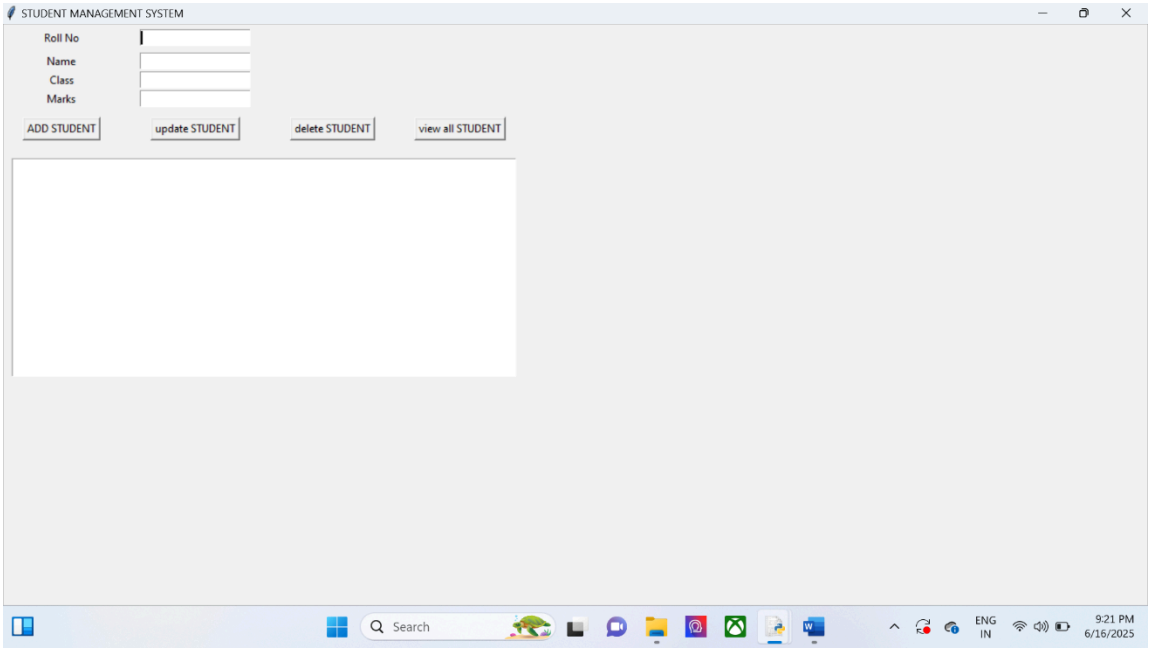
For delete student data find with roll number



After delete the student data



After deleting all students record



FUTURE SCOPE OF PROJECT

Enhanced Widgets

- **Advanced Input Forms:** Use of enhanced input widgets for better data entry (like combo boxes, calendar pickers, and auto-suggestions for fields such as class, subject, and grade).
- **Dynamic Tables:** Integration of dynamic, sortable tables for displaying student data with real-time filtering and searching options.
- **Interactive Dashboards:** Develop interactive dashboards to visualize statistics like performance trends, attendance summaries, and more.

Theming and Styles

- **Custom Themes:** Implementation of multiple themes (light/dark mode) to suit different user preferences.
- **Modern UI Designs:** Use modern fonts, icons, and layout styles to make the application visually appealing and user-friendly.

Integration with Modern Frameworks

- **Hybrid Development:** Combine Tkinter with web-based technologies (like Flask for web admin panel) to make a hybrid application.
- **Database Abstraction:** Use ORMs (like SQLAlchemy) instead of raw SQL for smoother database operations and scalability.

Cross-Platform Compatibility

- Ensure that the application runs flawlessly on **Windows, Linux, and macOS**, with consistent appearance and behavior.
- Package the application as a standalone executable using PyInstaller or similar tools for easy distribution.

Performance Improvements

- **Optimized Data Loading:** Implement pagination and lazy loading for large student datasets.
- **Memory Management:** Efficient handling of resources to avoid memory leaks and ensure smooth performance.

Community Contributions

- **Plugin Architecture:** Allow third-party plugins (e.g., SMS report generator, graphical attendance viewer).
- **Open-Source Repository:** Encourage community development and extensions via GitHub contributions.

Documentation and Tutorials

- Provide **in-depth documentation**, tooltips within the application, and step-by-step tutorials for:
  - Installation
  - Data entry
  - Generating reports
  - User management

Support for Asynchronous Programming

- Use asyncio for operations like data backup, syncing with the cloud, or email sending, so the UI remains responsive during long-running tasks.

AI and Machine Learning Integration

Predictive Analytics

- **Dropout Prediction:** Analyze attendance, grades, and participation to predict students at risk of dropping out.
- **Grade Forecasting:** Use past performance data to predict final exam scores and give suggestions to improve.

Personalized Feedback

- Generate automatic performance summaries with AI-based suggestions like "Needs improvement in Math" or "Attendance is

below average."

Enhanced User Experience

Customizable Access Levels

- Admin, Teacher, Student, and Parent logins with role-based dashboards.

Student Profiles and Tracking

- Track metrics such as:
  - Grades
  - Attendance
  - Activities
  - Disciplinary actions
- Generate visual performance graphs.

Adaptive Notifications

- Send intelligent alerts (e.g., low attendance, approaching fee deadlines, or exam schedules) via email/SMS.

Real-Time Alerts

- Pop-up messages or audio alerts for events like:
  - Duplicate roll numbers
  - Missing student data
  - Successful form submission.

Performance Analysis and Feedback

Advanced Reporting

- Generate detailed printable reports such as:
  - Grade cards
  - Attendance sheets
  - Subject-wise performance
- Export in **PDF/Excel/CSV** formats.

Cross-Platform Compatibility

Web and Mobile Support

- Build companion web and mobile apps for:
  - Students to view marks/attendance
  - Parents to get updates
  - Teachers to enter marks from phones

Cloud Sync and Backup

- Store data on cloud services (e.g., Firebase, AWS) to ensure data accessibility and safety across devices.

Enhanced Graphics and Sound

User-Centric Interface

- Graphical charts (pie, bar, line) for attendance and performance.
- Use avatars or photos in student profiles.

Audio Feedback

- Optional sound notifications for success/failure of form submissions or alerts.

Educational and Training Tools

Tutorials and Help Sections

- Step-by-step guidance within the application to train teachers and staff.
- Integrated help bot or FAQ section.

Integration with School ERP

- Seamless data sharing with existing educational systems such as:
  - Timetable management
  - Fee collection portals
  - Library system

CONCLUSION

The future scope of the **Student Management System Python project** is both broad and impactful. With enhanced UI, AI-powered insights, multi-device access, and advanced analytics, this project can evolve into a full-fledged **Educational ERP Solution**. Whether used in schools, colleges, or coaching centers, continuous innovation will make it an indispensable tool for educational institutions.

STUDENT SCORES PREDICTION USING DECISION TREE REGRESSION

INTRODUCTION

This project aims to analyze student performance based on the number of hours they study. We use a decision tree regression model to predict student scores and visualize the dataset using various matplotlib plots including bar, line, pie, and scatter plots. The dataset used is student_scores.csv, which contains two columns: Hours and Scores.

OBJECTIVES

- Understand and apply Decision Tree Regression for prediction.
- Perform data visualization using matplotlib.
- Evaluate model performance using R^2 and MSE.
- Use Python libraries and modules to build a complete data analysis workflow.

CONCEPTS COVERED IN THIS PROJECT

This project demonstrates a variety of crucial data science and machine learning concepts, such as:

- Supervised Learning: A method where a model is trained on labeled data; in this case, it learns the relationship between hours studied and the scores achieved.
- Regression Modeling: Used to predict continuous outcomes, specifically through decision tree regression in this study.
- Exploratory Data Analysis (EDA): Initial investigation of data using summary statistics and visual techniques to understand patterns.
- Data Splitting Strategy: The dataset is divided into training and testing subsets using an 80/20 split for validation.
- Model Evaluation Metrics: Quantitative metrics like R^2 score and Mean Squared Error (MSE) are applied to measure prediction accuracy.
- Decision Tree Learning: A tree-structured approach used to model and predict outcomes from input data.
- Data Visualization Techniques: The data is represented visually using bar, line, scatter, and pie charts for easier interpretation.
- Use of sklearn and matplotlib: The project uses sklearn for modeling and matplotlib for visualizing data and results.
- Data Cleaning and Preprocessing: Though the dataset is clean, handling missing values and formatting are always crucial in practice
- Feature Selection: Separating independent (Hours) and dependent (Scores) variables to train the model.
- Understanding R^2 and MSE: Interpreting the statistical metrics to evaluate model fit and error.

- Random State for Reproducibility: Ensures consistent splitting of the dataset for repeatable experiments.
- Comparing Actual vs Predicted Values: Critical for assessing model performance and guiding future improvements.

TECHNOLOGIES USED

- Language: Python 3
- Libraries:
    - pandas: For data manipulation
    - sklearn (scikit-learn): For regression models and evaluation
    - matplotlib: For plotting and visualization

IMPORTING LIBRARIES

```
import pandas as pd
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt
```

DATA LOADING AND INSPECTION

```
df = pd.read_csv("student_scores.csv")
print("📋 Full Dataset:\n", df)
```

OUTPUT:

```
📋 Full Dataset:
    Hours  Scores
0     2.5      21
1     5.1      47
2     3.2      27
3     8.5      75
4     3.5      30
5     1.5      20
6     9.2      88
7     5.5      60
8     8.3      81
9     2.7      25
10    7.7      85
11    5.9      62
12    4.5      41
13    3.3      42
14    1.1      17
15    8.9      95
16    2.5      30
17    1.9      24
18    6.1      67
19    7.4      69
20    2.7      30
21    4.8      54
22    3.8      35
23    6.9      76
24    7.8      86
```

DATA SPLITTING USING ILOC

```python
# Step 3: Extract input (X) and target (y) using iloc
X = df.iloc[:, 0:1]   # Feature: 'Hours'
y = df.iloc[:, 1]     # Target: 'Scores'

print("\n📓 Input Feature (X - Hours):\n", X)
print("\n🎯 Output Target (y - Scores):\n", y)
```

SPLIT INTO TRAINING AND TESTING SETS

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=1)

print("\n Training Input (X_train):\n", X_train)
print("\n Training Output (y_train):\n", y_train)
print("\n Testing Input (X_test):\n", X_test)
print("\n Testing Output (y_test):\n", y_test)
```

DECISION TREE REGRESSION

```
tree_model = DecisionTreeRegressor(random_state=1)
tree_model.fit(X_train, y_train)
y_pred_tree = tree_model.predict(X_test)
```

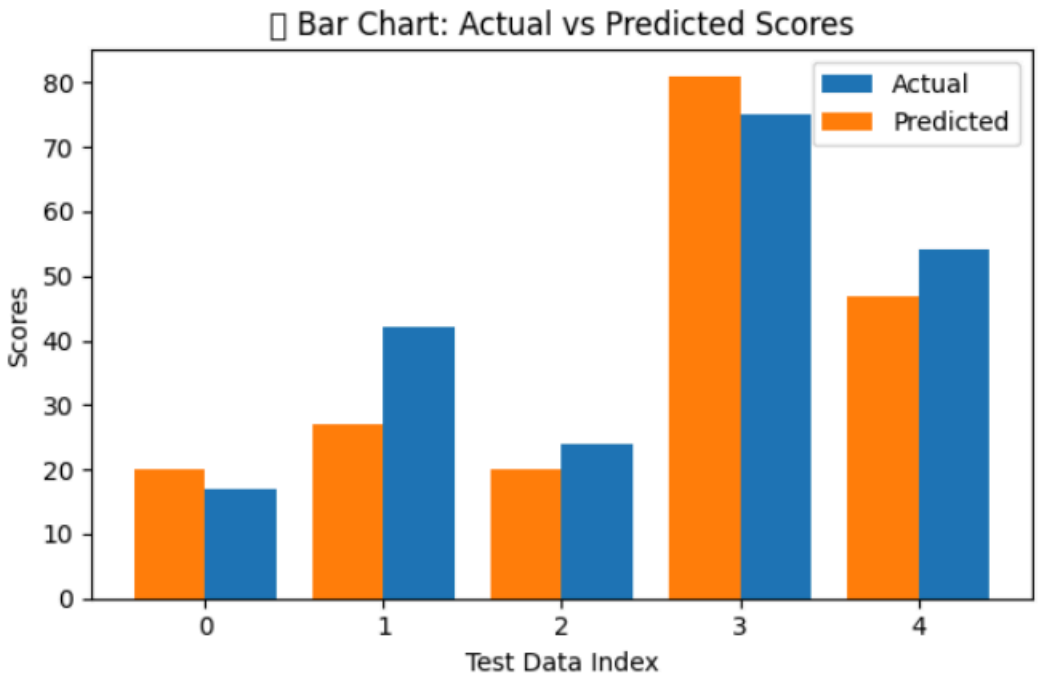
MODEL EVALUATION

```
mse_tree = mean_squared_error(y_test, y_pred_tree)
r2_tree = r2_score(y_test, y_pred_tree)

print("\n📊 Decision Tree Evaluation:")
print("Mean Squared Error (MSE):", mse_tree)
print("R² Score:", r2_tree)
```

DATA VISUALIZATION

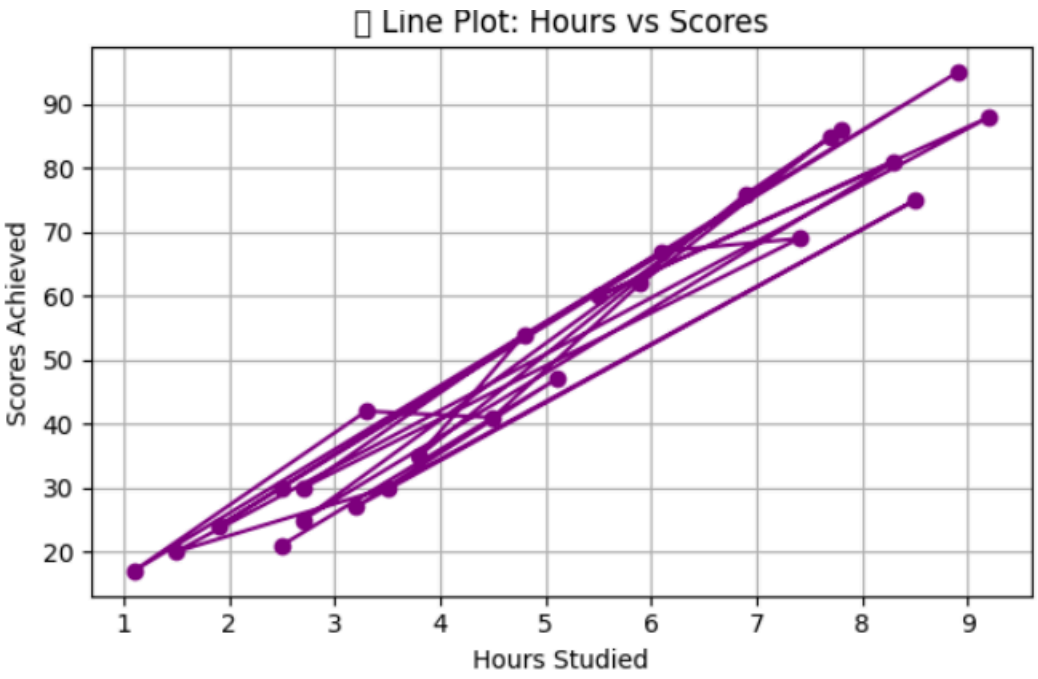
Bar Chart

```
plt.figure(figsize=(6,4))
plt.bar(range(len(y_test)), y_test, width=0.4, label='Actual', align='edge')
plt.bar(range(len(y_pred_tree)), y_pred_tree, width=-0.4, label='Predicted',
align='edge')
plt.xlabel('Test Data Index')
plt.ylabel('Scores')
plt.title('📊 Bar Chart: Actual vs Predicted Scores')
plt.legend()
plt.tight_layout()
plt.show()
```



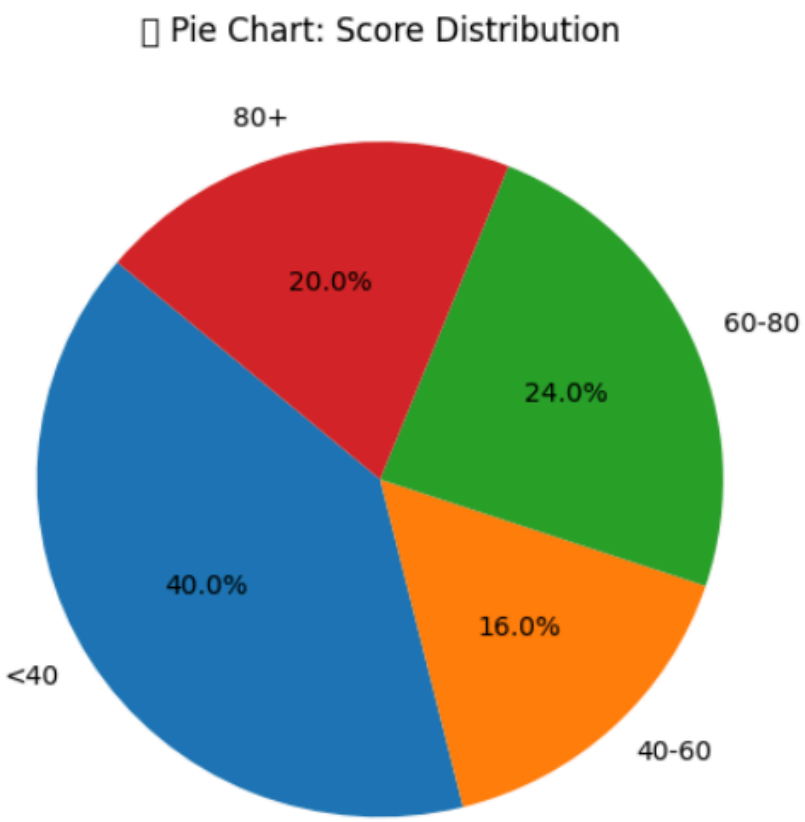
Line Plot

```
plt.figure(figsize=(6,4))
plt.plot(df['Hours'], df['Scores'], marker='o', color='purple')
plt.title('📈 Line Plot: Hours vs Scores')
plt.xlabel('Hours Studied')
plt.ylabel('Scores Achieved')
plt.grid(True)
plt.tight_layout()
plt.show()
```



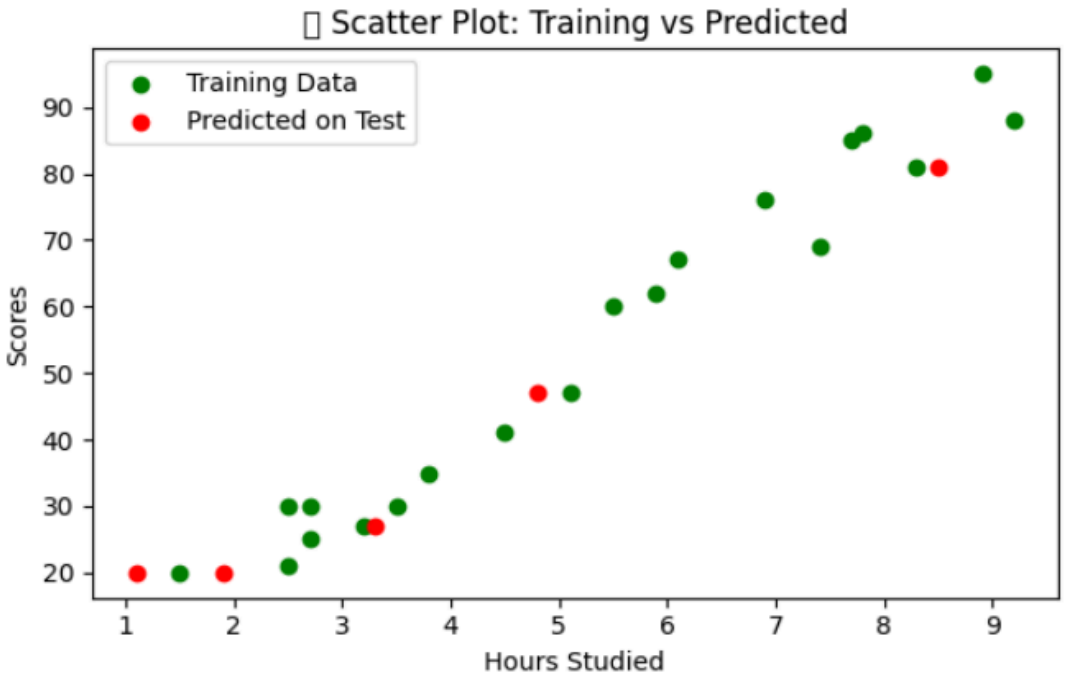
Pie Chart

```
score_ranges = ['<40', '40-60', '60-80', '80+']
counts = [
    len(df[df['Scores'] < 40]),
    len(df[(df['Scores'] >= 40) & (df['Scores'] < 60)]),
    len(df[(df['Scores'] >= 60) & (df['Scores'] < 80)]),
    len(df[df['Scores'] >= 80])
]
plt.figure(figsize=(5,5))
plt.pie(counts, labels=score_ranges, autopct='%1.1f%%', startangle=140)
plt.title('🍪 Pie Chart: Score Distribution')
plt.tight_layout()
plt.show()
```



Scatter Plot

```
plt.figure(figsize=(6,4))
plt.scatter(X_train, y_train, color='green', label='Training Data')
plt.scatter(X_test, y_pred_tree, color='red', label='Predicted on Test')
plt.xlabel('Hours Studied')
plt.ylabel('Scores')
plt.title('📍 Scatter Plot: Training vs Predicted')
plt.legend()
plt.tight_layout()
plt.show()
```


Scatter Plot: Training vs Predicted
Training Data
Predicted on Test
Scores
Hours Studied
20
30
40
50
60
70
80
90
1
2
3
4
5
6
7
8
9

FUTURE SCOPE

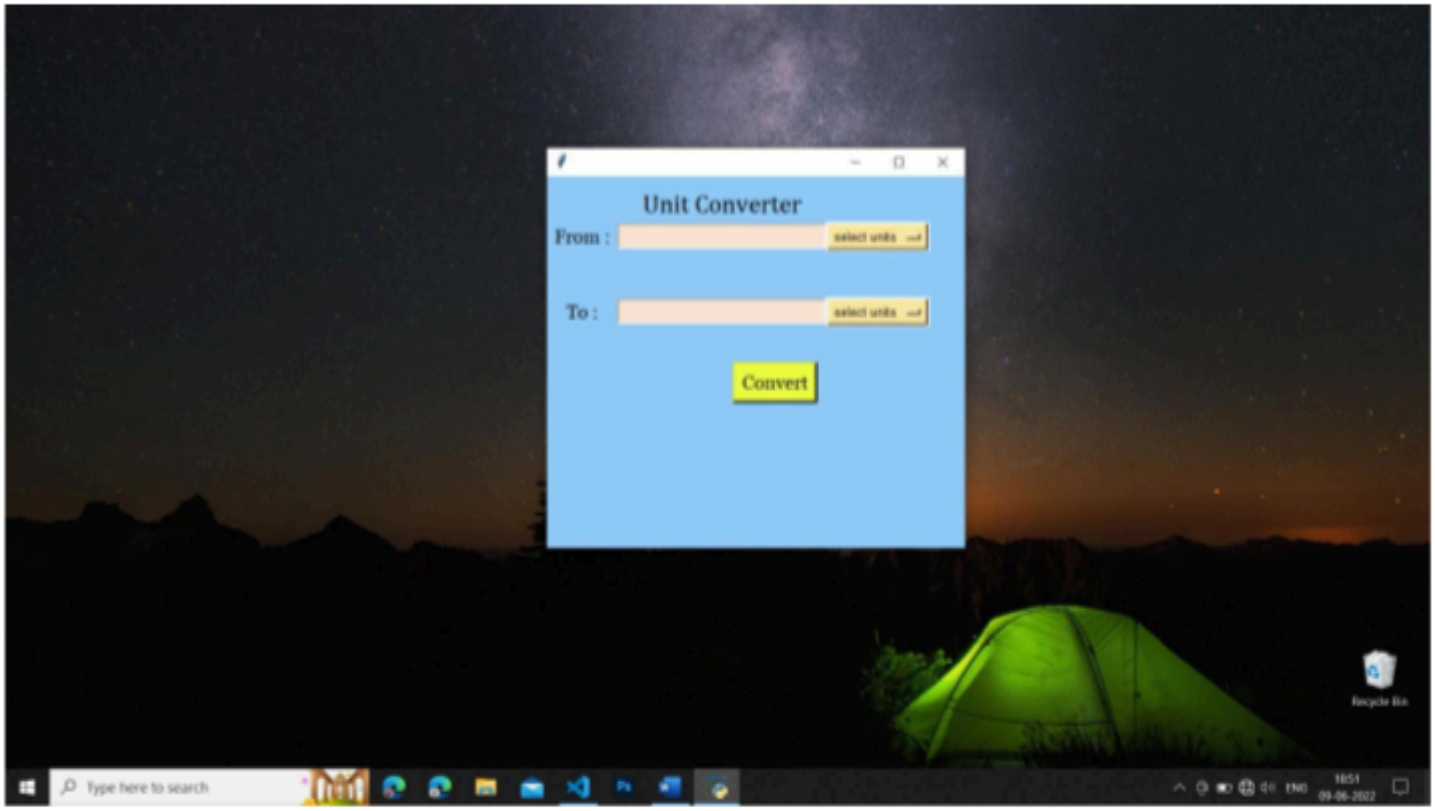
- Use larger datasets with more features like attendance, test preparation, etc.
- Compare multiple regression models (Linear, Random Forest).
- Deploy the model with a web-based GUI.
- Add interactive charts using Plotly or Dash.

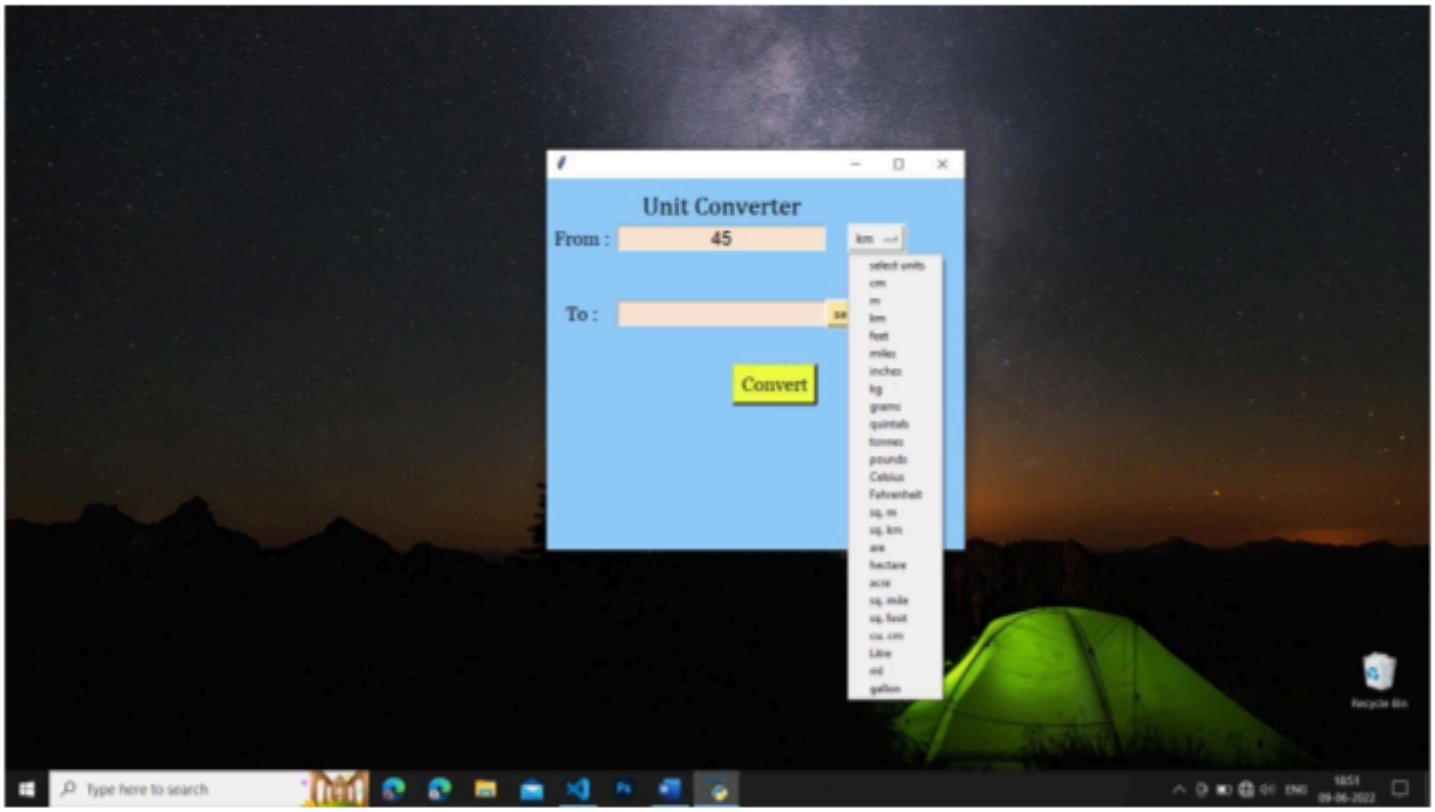
CONCLUSION

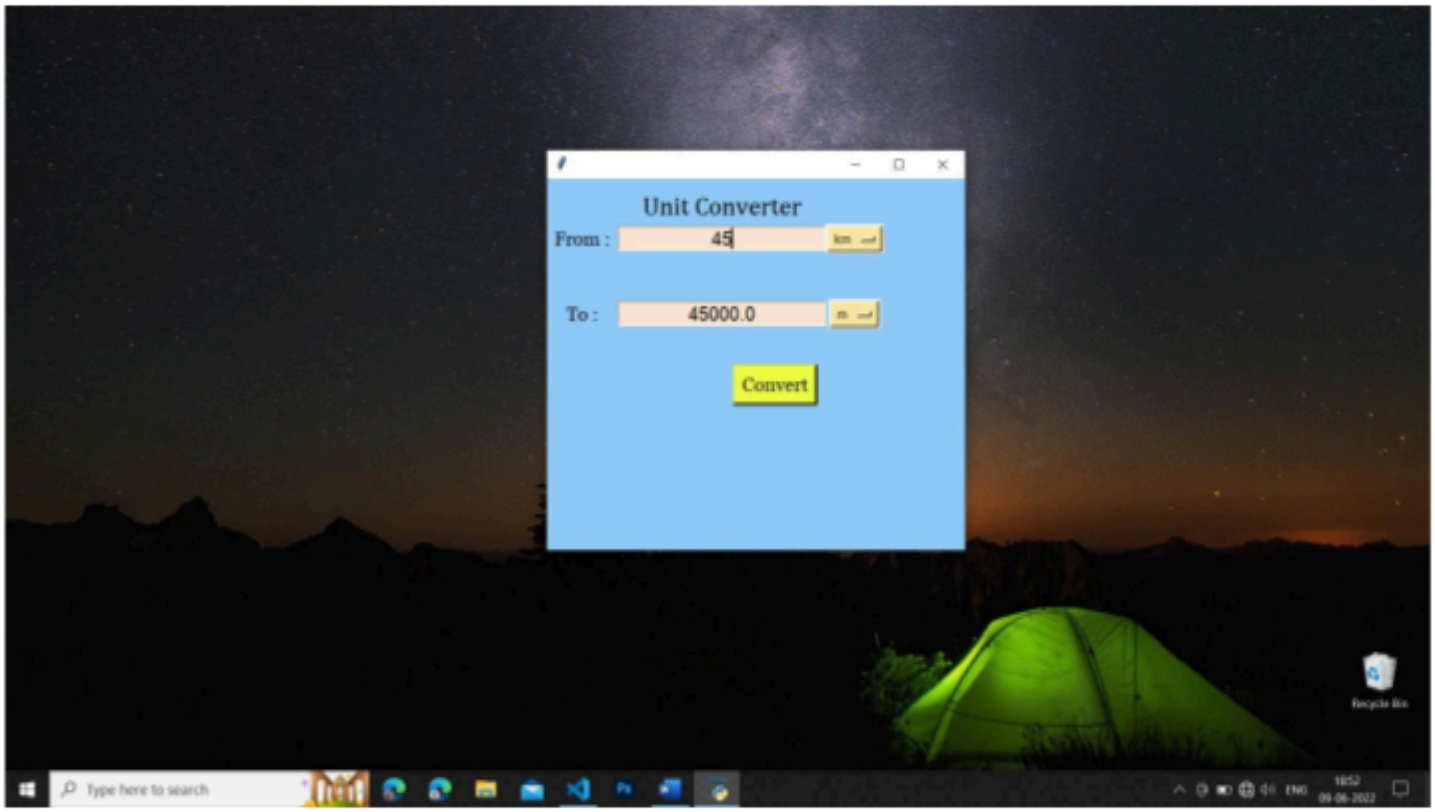
The decision tree model effectively predicts student scores based on study hours with reasonable accuracy. Visualization helps interpret trends and data distribution. This project demonstrates a clear workflow from loading data to modeling and visual insights.

REFERENCES

- scikit-learn documentation
- matplotlib documentation
- kaggle.com (for datasets)


Unit Converter
From :
select units
To :
select units
Convert
Recycle Bin
Type here to search
ENG
18:51
09-06-2022


Unit Converter
From :
45
km
select units
cm
m
km
feet
miles
inches
kg
grams
quintals
tonnes
pounds
Celcius
Fahrenheit
sq. m
sq. km
are
hectare
acre
sq. mile
sq. foot
cu. cm
Litre
ml
gallon
To :
Convert
Recycle Bin
Type here to search
ENG
1851
09-06-2022


Unit Converter
From :
45
km
To :
45000.0
m
Convert
Recycle Bin
Type here to search
ENG
1852
09-06-2022

Conclusion

The decision tree model effectively predicts student scores based on study hours with reasonable accuracy. Visualization helps interpret trends and data distribution. This project demonstrates a clear workflow from loading data to modelling and visual insights.

Future Scope of the Project

- Use larger datasets with more features like attendance, test preparation, etc.
- Compare multiple regression models (Linear, Random Forest).
- Deploy the model with a web-based GUI.
- Add interactive charts using Dash.

Thank You