BASICS OF PYTHON PROGRAMMING

Course Contents/Syllabus

UNIT-I

Foundations of Python Programming: From History to Setup and Syntax ,Overview of Python: History and Features, Python 2 vs Python 3, Setting up the Python Environment: Installing Python, IDEs and Text Editors, Basic Syntax and Data Types: Variables and Data Types, Operators, Input/output.

What is the Python programming language?

Python is an interpreted, object-oriented, general-purpose programming language that's a popular choice for software and web development. Python is modular, meaning it's easily integrated with other technologies.

It's also an open-source language – there's a well-established development community, with the Python Software Foundation overseeing quality.

Philosophy and Design

One of the key philosophies behind Python's development is **"The Zen of Python"**, a collection of aphorisms that emphasize simplicity, readability, and explicitness in code. This can be accessed by typing import this in a Python shell.

Python's development is guided by the **PEP (Python Enhancement Proposals)** process, which allows developers to propose changes and improvements to the language. The most famous PEP is **PEP 20** (The Zen of Python), which emphasizes principles .

The **Python Software Foundation (PSF)**, established in 2001, plays a significant role in managing Python's development and organizing events like **PyCon**.

Why the name "Python"? Van Rossum named Python after the British comedy series **"Monty Python's Flying Circus"**, not after the snake. He was a fan of the show and wanted a name that was short, unique, and slightly mysterious.

History of Python Programming Language

Python has an interesting history that has shaped it into one of the most popular programming languages today. Here's a timeline of its evolution:

1. Early Beginnings (1980s - 1990)

- **1980s**: The origins of Python can be traced back to the 1980s, when **Guido van Rossum**, a Dutch programmer, was working on the **ABC programming language** at the Centrum Wiskunde & Informatica (CWI) in the Netherlands.
- ABC was a teaching language designed to be easy for beginners, but it had limitations. Van Rossum wanted to create a **more powerful, flexible** language while maintaining simplicity.
- **Late 1980s**: Van Rossum started working on what would become Python as a successor to ABC.

2. Creation of Python (1991)

- **February 20, 1991**: Guido van Rossum released the **first version of Python (0.9.0) as an open-source project**. This version included key features like exception handling, functions, and core data types such as lists and dictionaries.

3. Python 1.0 (1994)

- Python 1.0 was released in January 1994. By this time, the language had already gained a small following, with improvements such as **more built-in modules and more robust exception handling**.
- Some key features included:
    - **The introduction of lambda functions** for functional programming.
    - **Basic support for object-oriented programming**.
    -

4. Python 2.x Series (2000 - 2010)

- **2000**: Python 2.0 was released, and it included significant improvements like garbage collection, list comprehensions, and Unicode support.
- The 2.x series introduced several features and optimizations, but also broke backward compatibility with some earlier versions.
- **Python 2.7** was the last major release in the 2.x series and became widely used for many years. It was actively maintained until January 1, 2020, when it officially reached the end of its life.

5. Python 3.x Series (2008 - Present)

- **2008**: Python 3.0 was released, marking a major transition from Python 2.x. It aimed to fix the design flaws of Python 2 but introduced some backward compatibility issues, meaning that code written in Python 2 might not work in Python 3 without modification.

    - **Key features of Python 3.x:**
        - **Print function**: The print statement was replaced by the print() function.
        - **Integer Division**: In Python 3, division between integers results in a float (e.g., 5 / 2 == 2.5), whereas in Python 2, it would result in integer truncation (5 / 2 == 2).
        - **Unicode by default**: Python 3 treats strings as Unicode by default, which improved handling of non-ASCII characters.
- **Python 3.6 and 3.7**: These versions introduced additional features like **f-strings** for easier string formatting, and improvements in type hints, making Python more suitable for large-scale applications.

6. Python Today

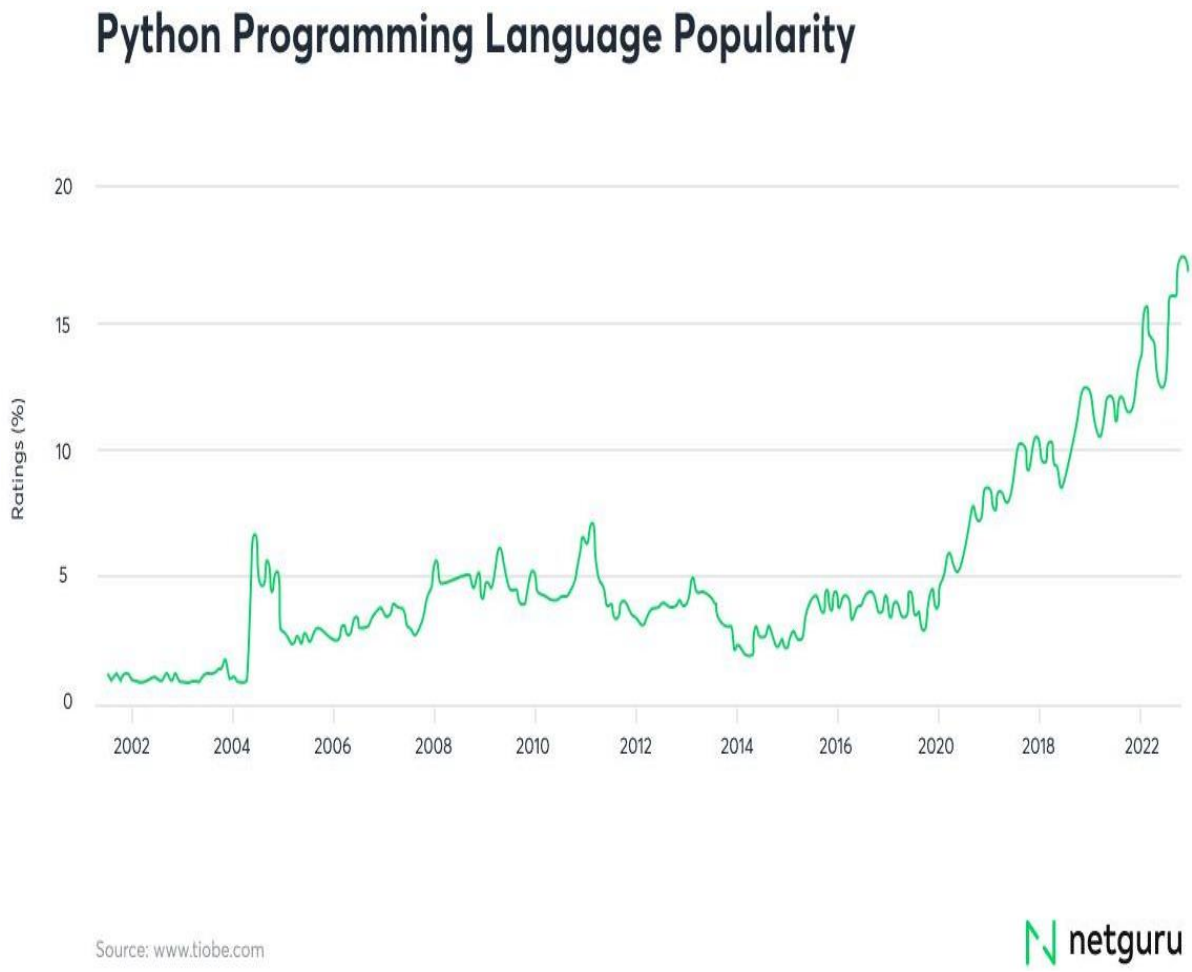
- **2020**: Python 2 officially reached the end of its life. While many older projects continued to rely on Python 2, the Python 3.x series became the standard.
- Python has become one of the most widely used programming languages globally, especially in fields like:
    - **Data science and machine learning** (thanks to libraries like NumPy, pandas, and TensorFlow)
    - **Web development** (with frameworks like Django and Flask)
    - **Scripting and automation**

Key Milestones:

- **2000s**: Python became a favorite among startups and large companies alike, particularly for web development and scientific computing.
- **2010s**: With the rise of data science and machine learning, Python gained immense popularity in those fields.
- **2020s**: Python continues to evolve with ongoing updates (e.g., Python 3.8, 3.9, 3.10, 3.11), focusing on performance, ease of use, and new language features.

Why is Python one of the most popular programming languages for software development?

Python is a **dynamic and versatile** programming language that's been used for software development for over **34 years**. Indeed, it's currently ranked number one in the **TIOBE Index**, measuring the popularity of programming languages. Moreover, TIOBE ranked Python language of the year in 2007, 2010, 2018, 2020, and 2021.



Source: TIOBE

Features of Python Language:

High-Level Language

Python is a high-level language, meaning the code is closer to human language than machine language. This makes it easier to read and

understand. The developers can write less complex code. The high-level nature of Python also allows developers to write code more quickly.

Object-Oriented Programming Language

Python is a powerful object-oriented programming language that supports OOPs concepts like inheritance, polymorphism, and encapsulation.

It allows users to create reusable and modular code, making it easier to manage large codebases. Examples: NumPy, matplotlib, Django, and Panda.

Interpreted

Python is an interpreted language executed line by line by an interpreter. This feature makes it easier to debug and test code. The errors can be identified and corrected quickly. The interpreter provides immediate feedback, which can be helpful in development.

Platform Independent

It is a platform-independent programming language that can run on different operating systems and hardware architectures without modification.

Easy integrations

Python is called a **"glue language**" because it's easy to integrate with other components such as languages, web frameworks, external services, and existing infrastructure elements.

Stable and secure.

Because of its simple syntax and readability, Python has developed a reputation as a secure technology, making it a top choice for financial applications dealing with sensitive data.

Large and vibrant community

Python is one of the most widespread coding languages today, with a **community of around 15.7 million**, second only to JavaScript. Developers create and share tools and knowledge for others to benefit from.

Ready-to-use solutions

The programming language offers over **137,000 open-source libraries** and an array of powerful Python frameworks, so there's access to much functionality, speeding up the development process.

Third-Party Libraries

Python has a great and active community that has developed many third-party libraries for various tasks. These libraries include tools for **scientific computing, web development, machine learning, and data manipulation**. The availability of third-party libraries has made Python a popular language.

Python builds simple and complex apps easily

Any kind of business can use Python, which is why it's used by startups and multi-billion dollar corporations alike.

GUI Programming Support

Python provides support for Graphical User Interface (GUI) programming. It means developers can easily create desktop applications using libraries like **Tkinter, PyQt, and PyGTK.**

Frontend and backend development

Python is a popular language for frontend and **backend web development** due to its simplicity and versatility. For frontend development, Python offers web frameworks **like** Flask and Django. Python **provides libraries** for database access, server-side scripting, and networking for backend development.

Easy to Debug

Python provides excellent debugging capabilities, with built-in tools like pdb (Python Debugger) and IDEs like PyCharm, which make it easy to find and fix bugs in the code.

Limitations of Python

- Python is **slower** compared to compiled languages like C++ due to its interpreted nature, which may not be suitable for performance-intensive applications.
- While Python can be used for web and mobile development, it lacks native mobile development frameworks compared to Java or Swift.

What Is an IDE?

- An IDE enables programmers to combine the different aspects of writing a computer program.
- IDEs increase programmer productivity by introducing features like editing source code, building executables, and debugging.

Python IDEs

Now that you know about the integrated Development Environment, let's look at a few popular Python IDEs. Note that we won't be ranking these IDEs just for the sake of it because we believe that different IDEs are meant for various purposes.

1. IDLE

- IDLE (Integrated Development and Learning Environment) is a default editor that accompanies Python
- This IDE is suitable for beginner-level developers
- The IDLE tool can be used on Mac OS, Windows, and Linux
- Price: Free

The most notable features of IDLE include:

- Interactive interpreter with syntax highlighting, and error and i/o messages
- Smart indenting, along with basic text editor features
- A very capable debugger
- Its a great Python IDE for Windows

Image Source: Stack overflow

What is a Text Editor?

A text editor is a type of program that allows you to edit plain text—essentially the most basic form of data on a computer. But don't let its simplicity fool you; for programmers, it's a powerful workspace for

writing and editing code. Unlike word processors, text editors don't add formatting to text, which is crucial when you're dealing with programming languages.

What is it Used For?

Text editors serve a multitude of purposes in programming. They enable developers to craft code from scratch, debug existing code, and sometimes manage files and directories. Modern editors also offer syntax highlighting, which makes it easier to distinguish elements of the code, and other helpful tools like line numbering, text searching, and code formatting.

About Text Editor:

A notepad is a text-only editor that only works with .txt files but can still read and edit file formats that can be edited. This project is a very simple one to make, but you are still going to have a lot of fun creating it.

Python Variables

Variable is a name that is used to refer to memory location. Python variable is also known as an identifier and used to hold value.

In Python, we don't need to specify the type of variable because Python is a infer language and smart enough to get variable type.

Variable names can be a group of both the letters and digits, but they have to begin with a letter or an underscore.It is recommended to use lowercase letters for the variable name. Rahul and rahul both are two different variables.

Rules for Identifier Naming

Variables are the example of identifiers. An Identifier is used to identify the literals used in the program. The rules to name an identifier are given below.

- The first character of the variable must be an alphabet or underscore (_).
- All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore, or digit (0-9).
- Identifier name must not contain any white-space, or special character (!, @, #, %, ^, &, *).
- .
- Identifier names are case sensitive; for example, my name, and MyName is not the same.
- Examples of valid identifiers: a123, _n, n_9, etc.
- Examples of invalid identifiers: 1a, n%4, n 9, etc.

Declaring Variable and Assigning Values

Python does not bind us to declare a variable before using it in the application. It allows us to create a variable at the required time.We don't need to declare explicitly variable in Python. When we assign any value to the variable, that variable is declared automatically.

The equal (=) operator is used to assign value to a variable.

Consider the following valid variables name.

```
name = "Devansh"
age = 20
marks = 80.50
print(name)
print(age)
print(marks)
```

Output:

Devansh
20
80.5

valid variables name

1. name = "A"
2. Name = "B"
3. naMe = "C"
4. NAME = "D"
5. n_a_m_e = "E"
6. _name = "F"
7. name_ = "G"
8. _name_ = "H"
9. na56me = "I"

The multi-word keywords can be created by the following method.

- **Camel Case -** In the camel case, each word or abbreviation in the middle of begins with a capital letter. There is no intervention of whitespace. For example - nameOfStudent, valueOfVaraible, etc.
- **Pascal Case -** It is the same as the Camel Case, but here the first word is also capital. For example - NameOfStudent, etc.
- **Snake Case** - In the snake case, Words are separated by the underscore. For example - name_of_student, etc.

Multiple Assignments

Python allows us to assign a value to multiple variables in a single statement, which is also known as multiple assignments.

We can apply multiple assignments in two ways, either by assigning a single value to multiple variables or assigning multiple values to multiple variables. Consider the following example.

1. Assigning single value to multiple variables

Eg:

```
1.  x=y=z=50
2.  print(x)
3.  print(y)
4.  print(z)
```

Output:

```
50
50
50
```

2. Assigning multiple values to multiple variables:

Eg:

```
1.  a,b,c=5,10,15
2.  print (a)
3.  print (b)
4.  print (c)
```

Output:

```
5
10
15
```

Delete a variable

We can delete the variable using the **del** keyword. The syntax is given below.

Syntax –

1. **del** <variable_name>

In the following example, we create a variable x and assign value to it. We deleted variable x, and print it, we get the error **"variable x is not defined"**. The variable x will no longer use in future.

Example -

```
# Assigning a value to x
x = 6
print(x)
# deleting a variable.
del x
print(x)
```

Output:

```
6
Traceback (most recent call last):
  File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/multiprocessing.py", line 389, in
print(x)
NameError: name 'x' is not define
```

Python Data Types

Every **value has a data type, and variables can hold values**. Python is a powerfully composed language; consequently, we don't have to characterize the sort of variable while announcing it. **The interpreter binds the value implicitly to its type.**

a = 5

We did not specify the type of the variable a, which has the value five from an integer. The Python interpreter will automatically interpret the variable as an integer.

We can verify the type of the program-used variable thanks to Python. The type() function in Python returns the type of the passed variable.

Consider the following illustration when defining and verifying the values of various data types.

```
a=10
b="Hi Python"
c = 10.5
print(type(a))
print(type(b))
print(type(c))
```

Output:

```
<type 'int'>
<type 'str'>
<type 'float'>
```

Python has the following data types built-in by default, in these categories:

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Set Types:	set, frozenset
Boolean Type:	bool
Binary Types:	bytes, bytearray, memoryview
None Type:	NoneType

Example:

```
# DataType Output: str
x = "Hello World"

# DataType Output: int
x = 50

# DataType Output: float
x = 60.5

# DataType Output: complex
x = 3j

# DataType Output: list
x = ["geeks", "for", "geeks"]

# DataType Output: tuple
```

```
x = ("geeks", "for", "geeks")

# DataType Output: range
x = range(10)

# DataType Output: dict
x = {"name": "Suraj", "age": 24}

# DataType Output: set
x = {"geeks", "for", "geeks"}

# DataType Output: frozenset
x = frozenset({"geeks", "for", "geeks"})

# DataType Output: bool
x = True

# DataType Output: bytes
x = b"Geeks"

# DataType Output: bytearray
x = bytearray(4)

# DataType Output: memoryview
x = memoryview(bytes(6))

# DataType Output: NoneType
x = None
```

Python Comments

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

Creating a Comment

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

Example:

```
#This is a comment
print("Hello, World!")
```

Example :

```
print("Hello, World!") #This is a comment
```

Multiline Comments

Python does not really have a syntax for multiline comments.

To add a multiline comment you could insert a # for each line:

Example:

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

Multi-line comments using triple quote

Example:

```
"""
This is a comment
written in
more than just one line
"""
print("Hello, World!")
```

print() Syntax in Python

The full syntax of the print() function, along with the default values of the parameters it takes, are shown below.

This is what print() looks like underneath the hood:

print(*object,sep=' ',end='\n',file=sys.stdout,flush= False)

Let's break it down:

- ***object**

  can be none, one, or many data values to be printed, and it can be of any data type.
- **sep**

  is an optional parameter that specifies how more than one object are separated. The default is ' ' – a space.
- **end**

is an optional paramater that specifies with what the line will end. By default the print call ends with a newline, with \n being the newline character.

- **file**

is an optional paramater which is an object with a write method – it can write and append (add) the output to a file. The default is sys.stdout (or system standard output) and the output is displayed on the screen.

flush

- is a boolean parameter that specifies whether the stream will be forcibly flushed or buffered. Flushed means whether the print call will immediately take affect. The default value is False (or buffered).

Python print()

Let us understand Python print() by looking at a very basic example.

Python print() function prints the message to the screen or any other standard output device. In this article, we will cover about print() function in Python as well as it's various operations.

```
print('Hello World!')

# tells us that print() is a function and not a statement
print(type(print))
```

Output

```
Hello World!
<class 'builtin_function_or_method'>
```

Example : How print() works in Python?

Example: Passing single argument to the print() function

```
# Python program to illustrate print()
# Passing single argument
print(10)

print(12.35)

print('Python Programming')

print(True)

print(1 + 5j)

my_list = [1, 2, 3, 4]
print(my_list)

my_dict = {1: 'One', 2: 'Two', 3: 'Three'}
print(my_dict)
```

Output

```
10
12.35
Python Programming
True
(1+5j)
[1, 2, 3, 4]
{1: 'One', 2: 'Two', 3: 'Three'}
```

Example: Passing multiple arguments to the print() function

```
# Python program to illustrate print()
# Passing multiple arguments
name = 'Sam'
print('My name is', name)

age = 30
print('My age is', age)

a = 'Amy'
b = 'Tom'
print(a, 'and', b, 'are coming to the party')

x = 5
y = 10
print('The sum of', x, 'and', y, 'is:', x+y)
```

Output

```
My name is Sam
My age is 30
Amy and Tom are coming to the party
The sum of 5 and 10 is: 15
```

Example: Printing different data types in Python

```
num = 12

# Integer
print('Number: %d' %num)

#Float
print('Number: %f' %num)

# Exponential
print('Number: %e' %num)

# Octal
print('Number: %o' %num)

# Hexadecimal
print('Number: %x' %num)
```

Output

```
Number: 12
Number: 12.000000
Number: 1.200000e+01
Number: 14
Number: c
```

The separator

‘ ‘ is used between multiple objects. Take note of the gap between the two objects in the output.

The character **‘\n’** (newline) is used as the **end** parameter. Each print command, as you can see, displays the output in a new line.The default value used for the **file** parameter is **sys.stdout**. The result is displayed on the screen.

The **flush** default value is **False**. The stream is not flushed forcibly.

```
print(100)

print('Welcome to School')

salary = 10000
print('Your salary is:', salary)

print('Car', 'Bike', 'Train', 'Plane', sep = ' | ', end = '\n\n')

print('Hello', 'World', end = '\n')
```

Output

```
100
Welcome to School
Your salary is: 10000
Car | Bike | Train | Plane

Hello World
```

How do you print a new line in Python?

To print on a new line, the **'\n'** newline character is used in the print() function. This newline character can also be applied with the **sep** and **end** parameters of the Python print() function.

Example

```
# new line in the objects parameter
print('Hey buddy! \n How are you?')

# new line using sep parameter
print('Tom', 'Sam', 'Kim', 'Kylie', sep = '\n')

# new line using end parameter
print('Welcome', end = '\n')
print('to the Museum')
```

Output

```
Hey buddy!
How are you?
Tom
Sam
Kim
Kylie
Welcome
to the Museum
```

Print with string formatting

String formatting is a powerful technique that allows you to insert values into a string. In Python, you can use the **'%' operator** to **format a string**. **This operator** takes a string on **the left side** and **one or more values on t**he **right side, separated by commas**.

```
# Example 1: Printing a formatted string
name = "Alice"
age = 25
print("%s is %d years old." % (name, age))
# Output: Alice is 25 years old.

# Example 2: Printing a formatted string with a floating-point value
price = 19.9999
print("The price is $%.2f." % price)
# Output: The price is $19.99.
```

Print with the 'str.format()' method

is a newer, more versatile way of formatting strings in Python. This method allows you **to insert values into a string** using curly braces '{}', and **then pass** those **values as arguments to the 'format()' method.**

```
# Example 1: Printing a formatted string using 'str.format()'
name = "Bob"
age = 30
print("{} is {} years old.".format(name, age))

# Output: Bob is 30 years old.
```

```
name = "Bob"
age = 30.4545
print("{} is {:.2f} years old.".format(name, age))
```

Print with the 'f-string' syntax

The 'f-string' syntax is a newer, even more, concise way of formatting strings in Python. This syntax allows you to embed expressions directly into string literals, by prefixing the string with the letter 'f'.

```
# Example 1: Printing an f-string
name = "Carol"
age = 35
print(f"{name} is {age} years old.")
# Output: Carol is 35 years old.

name = "Carol"
age = 35.5656
print(f"{name} is {age:.2f} years old.")

# Example 2: Printing an f-string with a formatted number
price = 29.95
tax_rate = 0.08
print(f"The total price is ${price * (1 + tax_rate):.2f}.")
# Output: The total price is $32.34.
```

Print to a file or a stream

In addition to printing to the console, you can also print to a file or a stream in Python. This can be useful when you want to save the output for later analysis or to share with others. To do this, you can use the 'print' function with the 'file' keyword argument.

```
# Example 1: Printing to a file
with open('output.txt', 'w') as f:
    # Writing to the file using the print function
    print("This line will be written to a file.", file=f)

# Example 2: Printing to a stream
import sys
print("This line will be written to a stream.", file=sys.stderr)
```

In the first example, we open a file named 'output.txt' in write mode using the 'open' function and the 'with' statement.

Then, we pass the 'file' argument to the 'print' function, which writes the output to the file. In the second example, we use the 'sys.stderr' stream to print an error message, which can be redirected to a file or a different output destination.

Python input() Function

is used to take user input. By default, it returns the user input in form of a string.

Syntax:

```
input(prompt)
```

prompt [optional]: *any string value to display as input message*

Ex*: input("What is your name? ")*

Returns: *Return a string value as input by the user.*

By default input() function helps in taking user input as string.
If any user wants to take input as int or float, we just need to typecast it.

Python input() Function Example

```
# Taking input as string
color = input("What color is rose?: ")
print(color)

# Taking input as int
# Typecasting to int
```

```
n = int(input("How many roses?: "))
print(n)

# Taking input as float
# Typecasting to float
price = float(input("Price of each rose?: "))
print(price)
```

Output:

```
What color is rose?: red
red
How many roses?: 10
10
Price of each rose?: 15.50
15.5
```

Example 1: Taking **the** Name and Age of the user as input and printing it

By default, input returns a string. So the name and age will be stored as strings.

```
# Taking name of the user as input
# and storing it name variable
name = input("Please Enter Your Name: ")

# taking age of the user as input and
# storing in into variable age
age = input("Please Enter Your Age: ")

print("Name & Age: ", name, age)
```

Output:

```
Please Enter Your Name: Rohit
Please Enter Your Age: 16
Name & Age:  Rohit 16
```

Example 2: Taking two integers from users and adding them.

In this example, we will be looking at how to take integer input from users. To take integer input we will be using int() along with **Python input()**

Source code: Python program for the addition of two numbers with user input.

```
#Addition of two numbers inputted by a user
num1=int(input("Enter first no. :"))
num2=int(input("Enter second no. :"))
addition=(num1+num2)

#displaying the user input values
print("First number is :",num1)
print("Second number is :",num2)

#displaying addition of user input
print("Addition is : ",addition)
```

Output:

```
Enter first no. :5
Enter second no. :10
First number is: 5
Second number is: 10
Addition is: 15
```

Python Program to Find the Square Root

```
COPY CODE
numb = float(input('Enter a number: '))


numsqrt = numb ** 0.5

print('square root of %0.3f is %0.3f'%(numb ,numsqrt))


Output:

Enter a number

The square root of 8.000 is 2.828
```

Python Program to Calculate the Area of a Triangle

If a, b and c are three sides of a triangle. Then,

```
s = (a+b+c)/2

area = √(s(s-a)*(s-b)*(s-c))
```

Source Code

```
# Python Program to find the area of triangle
```

```
a = 5
b = 6
c = 7

# Uncomment below to take inputs from the user
# a = float(input('Enter first side: '))
# b = float(input('Enter second side: '))
# c = float(input('Enter third side: '))

# calculate the semi-perimeter
s = (a + b + c) / 2

# calculate the area
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
print('The area of the triangle is %0.2f' %area)
```

Python Program to Swap Two Variables

```
# Python program to swap two variables

x = 5
y = 10

# To take inputs from the user
#x = input('Enter value of x: ')
#y = input('Enter value of y: ')

# create a temporary variable and swap the values
temp = x
x = y
y = temp

print('The value of x after swapping: {}'.format(x))
```

```
print('The value of y after swapping: {}'.format(y))
```
Run Code

Output

```
The value of x after swapping: 10
The value of y after swapping: 5
```

In this program, we use the `temp` variable to hold the value of `x` temporarily. We then put the value of `y` in `x` and later `temp` in `y`. In this way, the values get exchanged.

Python Program to Convert Celsius To Fahrenheit In the program below, we take a temperature in degree Celsius and convert it into degree Fahrenheit. They are related by the formula:

```
fahrenheit = celsius * 1.8 + 32
```

Source Code

```
# Python Program to convert temperature in celsius to fahrenheit

# change this value for a different result
celsius = 37.5

# calculate fahrenheit
```

```
fahrenheit = (celsius * 1.8) + 32
print('%0.1f degree Celsius is equal to %0.1f degree Fahrenheit'
%(celsius,fahrenheit))
```

Output

```
37.5 degree Celsius is equal to 99.5 degree Fahrenheit
```

We encourage you to create a Python program to convert Fahrenheit to Celsius on your own using the following formula

```
celsius = (fahrenheit - 32) / 1.8
```

Python - Operators

Python operators are the constructs which can manipulate the value of operands. These are symbols used for the purpose of logical, arithmetic and various other operations.

Consider the expression 4 + 5 = 9. Here, 4 and 5 are called **operands** and + is called **operator**.

Types of Python Operators

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators

Python Arithmetic Operators

Operator	Name	Example
+	Addition	10 + 20 = 30

-	Subtraction	20 – 10 = 10
*	Multiplication	10 * 20 = 200
/	Division	20 / 10 = 2
%	Modulus	22 % 10 = 2
**	Exponent	4**2 = 16
//	Floor Division	9//2 = 4

Example:

Following is an example which shows all the above operations:

```
a =21
b =10
# Addition
print("a + b : ", a + b)

# Subtraction
print("a - b : ", a - b)

# Multiplication
print("a * b : ", a * b)
```

```
# Division
print("a / b : ", a / b)

# Modulus
print("a % b : ", a % b)

# Exponent
print("a ** b : ", a ** b)

# Floor Division
print("a // b : ", a // b)
```

This produce the following result –

```
a + b :  31
a - b :  11
a * b :  210
a / b :  2.1
a % b :  1
a ** b :  16679880978201
a // b :  2
```

Python Comparison Operators

Python comparison operators compare the values on either sides of them and decide the relation among them. They are also called relational operators. These operators are equal, not equal, greater than, less than, greater than or equal to and less than or equal to.

Operator	Name	Example

==	Equal	4 == 5 is not true.
!=	Not Equal	4 != 5 is true.
>	Greater Than	4 > 5 is not true.
<	Less Than	4 < 5 is true.
>=	Greater than or Equal to	4 >= 5 is not true.
<=	Less than or Equal to	4 <= 5 is true.

Following is an example which shows all the above comparison operations:

```
a =4
b =5

# Equal
print("a == b : ", a == b)

# Not Equal
print("a != b : ", a != b)

# Greater Than
```

```
print("a > b : ", a > b)

# Less Than
print("a < b : ", a < b)

# Greater Than or Equal to
print("a >= b : ", a >= b)
# Less Than or Equal to
print("a <= b : ", a <= b)
```

This produce the following result –

```
a == b :  False
a != b :  True
a >b :  False
a <b :  True
a >= b :  False
a <= b :  True
```

Python Assignment Operators

Operator	Name	Example
=	Assignment Operator	a = 10
+=	Addition Assignment	a += 5 (Same as a = a + 5)
-=	Subtraction Assignment	a -= 5 (Same as a = a - 5)

*=	Multiplication Assignment	a *= 5 (Same as a = a * 5)
/=	Division Assignment	a /= 5 (Same as a = a / 5)
%=	Remainder Assignment	a %= 5 (Same as a = a % 5)
**=	Exponent Assignment	a **= 2 (Same as a = a ** 2)
//	Floor Division Assignment	a //= 3 (Same as a = a // 3)

Following is an example which shows all the above assignment operations:

```
# Assignment Operator
a =10
# Addition Assignment
a +=5
print("a += 5 : ", a)
# Subtraction Assignment
a -=5
print("a -= 5 : ", a)
# Multiplication Assignment
a*=5
```

```
print("a *= 5 : ", a)

# Division Assignment
a/=5
print("a /= 5 : ",a)
# Remainder Assignment
a%=3
print("a %= 3 : ", a)
# Exponent Assignment
a**=2
print("a **= 2 : ", a)

# Floor Division Assignment
a//=3
print("a //= 3 : ", a)
```

This produce the following result −

```
a += 5 :  105
a -= 5 :  100
a *= 5 :  500
a /= 5 :  100.0
a %= 3 :  1.0
a **= 2 :  1.0
a //= 3 :  0.0
```

Logical Operators

Logical operators are used to combine conditions.

Symbol/ Keyword	Description	Example
and	Return True if both conditions are True. Return False if any of the condition is False.	print(10<20 and 20<30) Output:True Output is True as both conditions return True. print(10>20 and 20<30) Output: False Output is False as the first condition return False. print(10<20 and 20>30) Output: False Output is False as the second condition return False.
or	Return True if any of the conditions are True. Return False when both conditions are False.	print(10<20 or 20<30) Output: True Output is True as both conditions return True. print(10>20 or 20<30)

		Output: True Output is True as second condition return True. print(10>20 or 20<30) Output: False Output is False as both conditions return False.
not	Reverses the Boolean value. Return True if given condition returns False, return False if the given condition is True.	print(not 10<20) Output: False Output is False because given condition returns True and not reverses the True into False. print(not 10>20) Output: True The output is True because given condition returns False and not reverses the False into True.

Membership Operator

Membership operator checks whether the value is present in the given collection or not. As we have studied, a collection can be a list, tuple, etc.

Symbol	Description	Example
in	Returns True if the value on the left side is present in the collection given on the right side.	Alist = [1, 2, 3, 4] print(2 in Alist) Output: True Alist = [1, 2, 3, 4]

		print(10 in Alist) Output: False
not in	Returns True if the value on the left side is not present in the collection given on the right side. Basically, this is a combination of not and in the operator.	Alist = [1, 2, 3, 4] print(2 not in Alist) Output: False Alist = [1, 2, 3, 4] print(10 not in Alist) Output: True

Python Bitwise Operators

Python bitwise operators are normally used to perform bitwise operations on integer-type objects. However, instead of treating the object as a whole, it is treated as a string of bits. Different operations

NAME	OPERATOR	DESCRIPTION	SYNTAX
Bitwise AND Operator	&	Returns 1 if both operand bits are 1, else, returns 0.	x & y

Bitwise OR Operator	|	Returns 1 if at least one of the operand bits is 1; else, returns 0.	x | y
Bitwise NOT Operator	~	Inverts all bits.	~x
Bitwise XOR Operator	^	Returns 1 if one of the operand bits is 1 but not both; else, returns 0.	x ^ y

AND Operator

The AND operator returns 1 for each bit position where both operands have 1. Otherwise, it returns 0. This operator is often used to clear specific bits in an integer. Here's an example of using the AND operator in Python:

Example

```
a = 60  # 60 = 0011 1100
b = 13  # 13 = 0000 1101
c = a & b  # 12 = 0000 1100
print("Result of AND operation:", c)
```

Output

Result of AND operation: 12

OR Operator

The OR operator returns 1 for each bit position where at least one operand has 1. It returns 0 only if both operands have 0. This operator is commonly used to set specific bits in an integer. Here's an example of using the OR operator in Python:

Example

```
a = 60  # 60 = 0011 1100
b = 13  # 13 = 0000 1101
c = a | b  # 61 = 0011 1101
print("Result of OR operation:", c)
```

Output

Result of OR operation: 61

XOR Operator

The XOR operator returns 1 for each bit position where the operands have different bits. It returns 0 if both bits are the same. This operator is useful for flipping the bits of an integer. Here's an example of using the XOR operator in Python:

Example

```
a = 60  # 60 = 0011 1100
b = 13  # 13 = 0000 1101
c = a ^ b  # 49 = 0011 0001
print("Result of XOR operation:", c)
```

Output

```
Result of XOR operation: 49
```

NOT Operator

The NOT operator returns the complement of the operand, i.e., it changes each 1 to 0 and each 0 to 1. This operator is often used to reverse the bits of an integer. Here's an example of using the NOT operator in Python:

Example

```
a = 60  # 60 = 0011 1100
c = ~a  # -61 = 1100 0011
print("Result of NOT operation:", c)
```

Output

```
Result of NOT operation: -61
```

Left Shift Operator

The left shift operator moves the bits of the operand to the left by a specified number of positions. This effectively multiplies the operand by 2, raised to the power of the shift count. Here's an example of using the left shift operator in Python:**Example**

```
a = 10  # 0b1010
b = a << 2  # 40 = 0b101000
print("Result of left shift operation:", b)
```

Output

```
Result of left shift operation: 240
```

Right Shift Operator

The right shift operator moves the operand bits to the right by a specified number of positions. This effectively divides the operand by 2, raised to the power of the shift count. Here's an example of using the right shift operator in Python:

Example

```
a = 10   # 0b1010
b = a >> 2   # 2 = 0b0010
print("Result of right shift operation:", b)
```

Output

```
Result of right shift operation: 2
```

Python Identity Operators

The identity operators compare the objects to determine whether they share the same memory and refer to the same object type (data type).

Python provided two identity operators; we have listed them as follows:

- '**is**' Operator
- '**is not**' Operator

- The '**is**' operator evaluates to True if both the operand objects share the same memory location. The memory location of the object can
- be obtained by the "id()" function. If the "id()" of both variables is same, the "is" operator returns True.

- Example of Python Identity 'is' Operator

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

# Comparing and printing return values
print(a is c)
print(a is b)

# Printing IDs of a, b, and c
```

```
print("id(a) : ", id(a))
print("id(b) : ", id(b))
print("id(c) : ", id(c))
```

It will produce the following output –

```
True
False
id(a) :  140114091859456
id(b) :  140114091906944
id(c) :  140114091859456
```

Python 'is not' Operator

The '**is not**' operator evaluates to True if both the operand objects do not share the same memory location or both operands are not the same objects.

Example of Python Identity 'is not' Operator

Open Compiler

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

# Comparing and printing return values
print(a is not c)
print(a is not b)
```

```
# Printing IDs of a, b, and c
print("id(a) : ", id(a))
print("id(b) : ", id(b))
print("id(c) : ", id(c))
```

It will produce the following output −

```
False
True
id(a) :  140559927442176
id(b) :  140559925598080
id(c) :  140559927442176
```

Precedence of Operators

Precedence	Operator	Description
1	**	Exponential
2	~, +, -	Compliment, Unary plus, unary minus
3	*, /, %, //	Multiplication, division, modulo, floor division
4	+, -	Addition, subtraction
5	>>, <<	Right shift, left shift
6	&	Bitwise and

7	^, |	Bitwise XOR, or
8	<, <=, >, >=	Less than, less than equal to, greater than, greater to equal to
9	==, !=	Equality check, not equal to
10	%=, /=, //=, -=, +=, *=	Assignment operators
11	in, not in	Membership operators
12	not, and, or	Logical not, and, or

Python Reserve Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	as	assert
break	class	continue
def	del	elif
else	except	False
finally	for	from

global	if	import
in	is	lambda
None	nonlocal	not
or	pass	raise
return	True	try
while	with	yield

Python Lines and Indentation

Follow this Link:

https://www.almabetter.com/bytes/tutorials/python/indentation-in-python

UNIT-II	Control Flow and Functions	8hours

Control Flow: Conditional Statements(if,elif,else),Looping Constructs(for, while),
Exception Handling, **Functions:** Defining Functions, Function Parameters and Return Values, Scope and Lifetime of Variables, **File Handling**: Reading and Writing to Files, File Modes and Operations

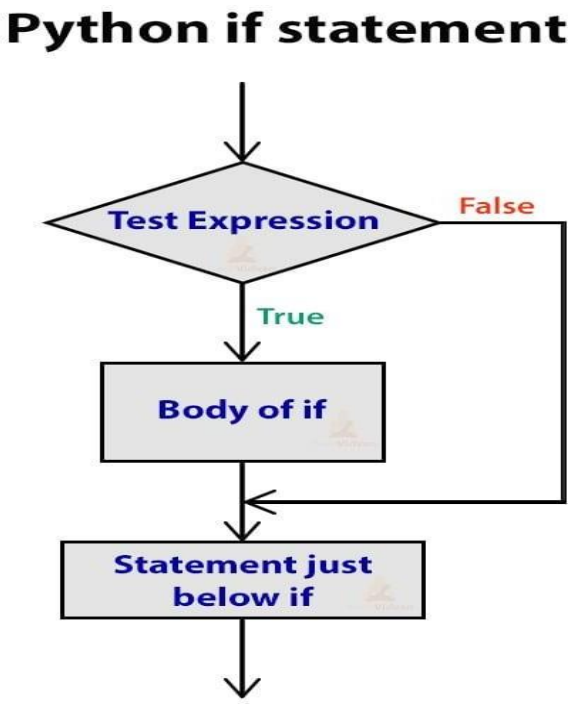
Python if statement

if statement is the most **simple form** of **decision-making** statement. It takes an **expression** and **checks** if the expression evaluates to **True** then the block of code in **if statement** will be **executed**.

If the expression evaluates to **False**, then the **block of code** is **skipped**.

Syntax:

```
if ( expression ):
        Statement 1
        Statement 2
.
Statement n
```


Python if statement
Test Expression
False
True
Body of if
Statement just
below if

Example 1:

```
a = 20 ; b = 20
if ( a == b ):
        print( “a and b are equal”)
print(“If block ended”)
```

Output:

```
a and b are equal
If block ended
```

Example 2:

```
num = 5
if ( num >= 10):
        print(“num is greater than 10”)
print(“if block ended”)
```

Output:

```
If block ended
```

Python if else statement

The if statement tells us that if a condition is true, a block of statements will be executed; if the condition is false, the block of statements will not be executed.

But what if the condition is false and we want to do something different? When the condition is false, we can use the else statement in conjunction with the if statement to run a code block.

Python if else syntax

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

EX.

```
num = int(input()) #take integer as an input

if num >= 0:
    print("Positive number")
else:
    print("Negative number")
```

When you run the program with input -1, the output will be:

```
Negative number
```

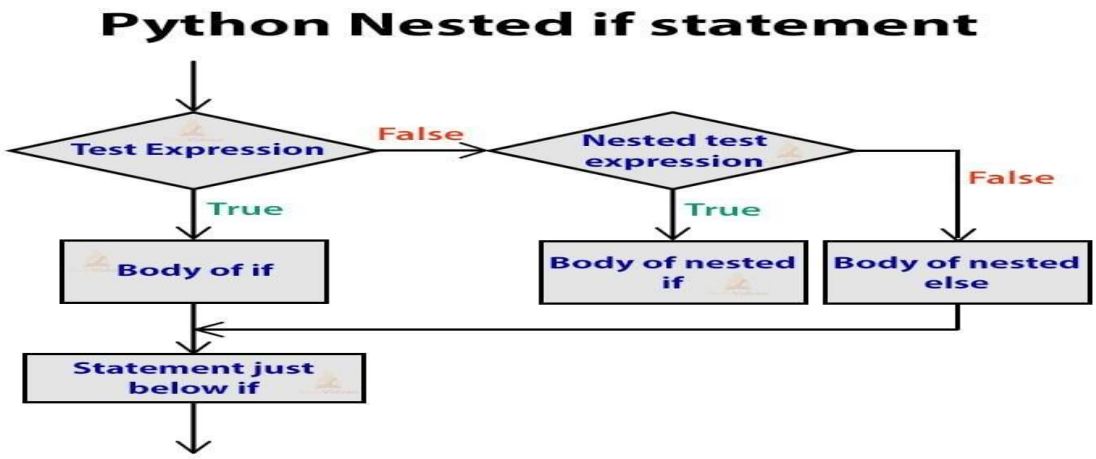
Python Nested-If Statements

In simple terms, nested if statements are if statements that are inside another if statement. Yes, Python allows us to nest any number of if statements inside another if statement block.

They come in handy when we have to make a series of decisions. Indentation is the only way to determine the level of nesting. They can be confusing, so avoid them unless necessary.

Python Nested-if Syntax

```
if (condition1):
   # Executes when condition1 is true
   if (condition2):
      # Executes when condition2 is true
   else:
#Executes when condition2 is false
else:
          #Execute when condition1 is false
```



Example:

i = 10

```
if (i == 10):
    #  First if statement
    if (i < 15):
        print("i is smaller than 15")
    # Nested - if statement
    # Will only be executed if statement above
    # it is true
    if (i < 12):
        print("i is smaller than 12 too")
    else:
        print("i is greater than 15")
else:
  print("i is not equal to 10")
```

Output:

i is smaller than 15

i is smaller than 12 too

Python If-elif ladder

A user can select from a variety of options in the if-elif ladder.
The if statements are executed from the top down. When one of the elif conditions is satisfied, the statement associated with that elif is executed, and the rest of the ladder is skipped. If none of the conditions is satisfied, the last else statement is executed.

Syntax for if..elif..else

```
if (condition1):
    Body of if
elif (condition2):
    Body of elif
elif (condition3):
    Body of elif
else:
    Body of else
```

Example **using “elif” to create a simple scoring system, where points are then equated to a grade ranging from A to F:**

```
score = 85
if score >= 90:
    print("A")
elif score >= 80:
    print("B")
elif score >= 70:
    print("C")
elif score >= 60:
    print("D")
else:
    print("F")
```

CALCULATOR PROGRAM

```
p = int(input("Enter the first input: "))

q = int(input("Enter the second input: "))

oper = input("Enter the type of operation you want to perform (+, -, *, /, %): ")

result = 0

if oper == "+":

    result = p+q

elif oper == "-":

    result = p-q

elif oper == "*":

    result = p*q

elif oper == "/":

    result = p//q # Integer Division

elif oper == "%":

    result = p%q

else:

    print("Invalid Input")

print("Your answer is: ",result)
```

Output:

```
Enter the first number: 3

Enter the second number: 7

Enter the type of operation you want to perform (+, -, *, /, %): +

Your answer is: 10
```

ShortHand if statement

Shorthand if can be used when only a single statement has to be processed inside the if block. The if statement and the statement can be on the same line.

Python program to illustrate shorthand if statement.

```
num=int(input()) #take any number as input
if num < 100: print("num is less than 100")
```

When you run the program with input 50,

the output will be:

```
num is less than 100
```

ShortHand if-else statement

When there is only one statement to be performed in both the if and else blocks, shorthand if-else can be used to write the statements in a single line.

Python program to illustrate shorthand if-else statement.

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

When you run the program with input 99, the

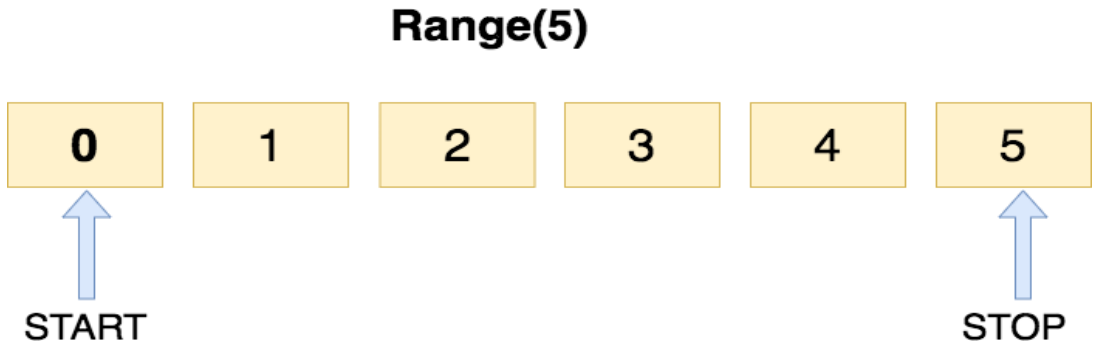
output will be:

```
=
```

Python range() Function Syntax

The range() function can be represented in three different ways, or you can think of them as three range() parameters:

- range(stop_value): By default, the starting point here is zero.
- range(start_value, stop_value): This generates the sequence based on the start and stop value.
- range(start_value, stop_value, step_size): It generates the sequence by incrementing the start value using the step size until it reaches the stop value.



Python range() function. Image by Author.

How to use range in Python (with examples)

Since two of the three parameters for Python `range()` are optional, there are several different ways to use this function. First, let's examine how to use it with the required parameters only.

With stop argument

Stop is the only required parameter because it tells Python which integer number will end the sequence**. For example**, suppose you want the range to start at 0 and stop at 6 (not including 6). In that case, your code might look like this:

If you want to include 6 in your iteration, your code would look like this instead:

```
for num in range(6):
    print(num)
```

```
0
1
2
3
4
5
```

Reminder
If you only pass a single argument to range, it will be passed as the stop argument.

With start and stop arguments

Start is not a required parameter. However, you will need to use it when starting your sequence with any integer number other than 0. Here's an example with 1 as the start argument and 10 as the stop argument (10included):

```
for num in range(1, 11):
    print(num)
```

```
1
2
3
4
5
6
7
8
9
10
```

With start, stop, and step arguments

Step is another optional parameter. It is useful for creating sequences that take steps larger or smaller than the default 1. You can also use it to print odd numbers or even numbers only. You have two options when choosing a custom step value:

1. Positive step
Any positive integer number you enter in the step value will result in an **incrementation**.

Example:

```
for num in range(1, 14, 3):
    print(num)
```

```
1
4
7
10
13
```

In the example above, the sequence starts at 1, stops at 14, and increments in steps of 3.

2. Negative step

If you want to **decrement**, your step must be a negative value.

Example:

```
for num in range(10, 0, -2):
    print(num)
```

```
10
8
6
4
2
```

In the example above, the sequence starts at 10, ends at 0, and increments in steps of -2.

Python For Loops

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

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

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

The basic syntax of a for loop is shown below:

```
for variable in iterable:
    <body>
```

Here's a more detailed breakdown of this syntax:

- **for** is the keyword that initiates the loop header.
- **variable** is a variable that holds the current item in the input iterable.
- **in** is a keyword that connects the loop variable with the iterable.
- **iterable** is a data collection that can be iterated over.
- **<body>** consists of one or more statements to execute in each iteration.

Print individual letters of a string using the for loop

```
word="anaconda"
for letter in word:
    print (letter)
```

Output:

```
a

n

a

c

o

n

d

a
```

Using the for loop to iterate over a Python list or tuple

Lists and Tuples are iterable objects. Let’s look at how we can loop over the elements within these objects now.

```
words= ["Apple", "Banana", "Car", "Dolphin" ]

for word in words:

    print (word)
```

Output:

```
Apple

Banana

Car

Dolphin
```

Python for loop with range() function

Consider the following example where I want to print the numbers 1, 2, and 3:

```
for x in range(3):
    print("Printing:", x)

# Output

# Printing: 0
# Printing: 1
# Printing: 2
```

The range function also takes another parameter apart from the start and the stop. This is the **step parameter**. It tells the range function how many numbers to skip between each count.

In the below example, I've used number 3 as the step and you can see the output numbers are the previous number + 3.

```
for n in range(1, 10, 3):
    print("Printing with step:", n)

# Output
# Printing with step: 1
# Printing with step: 4
# Printing with step: 7
```

We can also use a negative value for our step argument to iterate backwards, but we'll have to adjust our start and stop arguments accordingly:

```
for i in range(100,0,-10):
   print(i)
```

Here, 100 is the start value, 0 is the stop value, and -10 is the range, so the loop begins at 100 and ends at 0, decreasing by 10 with each iteration. This occurs in the output:

```
Output
100
90
80
70
60
50
40
30
20
10
```

Example 1: Print the first 10 natural numbers using for loop.

```
n = 11

for i in range(1,n):
        print(i)
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

Example 2: Python program to print all the even numbers within the given range.

```
# if the given range is 10
given_range = 10

for i in range(given_range):

        # if number is divisble by 2
        # then it's even
        if i%2==0:

                print(i)
```

Output

0
2
4
6
8

How to use while loop in Python

While loops continuously execute code for as long as the given condition or, boolean expression, is true. They are most useful when you don't know how many times you'll need the code to execute.

For example,

you may want to run a piece of code as long as a given number is between 1 and 5. In that case, the loop will run *while* the number remains between 1 and 5. When the user chooses a 6 or any other number that isn't between 1-5, the control flow will exit the while loop body.

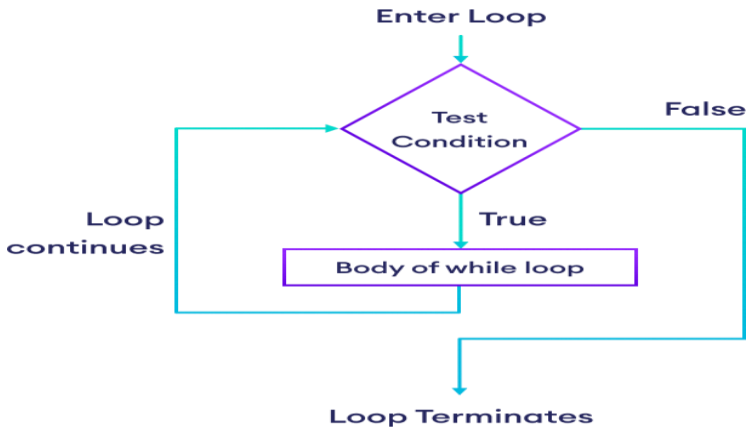
while Loop Syntax

```
while condition:
    # body of while loop
```

Here,

1. The while loop evaluates **condition**, which is a boolean expression.
2. If the condition is True, **body of while loop** is executed. The condition is evaluated again.
3. This process continues until the condition is False.
4. Once the condition evaluates to False, the loop terminates.

Flowchart of Python while Loop



Example 1

The following example illustrates the working of while loop. Here, the iteration run till value of count will become 5.

```
count=0
while count<5:
   count+=1
   print ("Iteration no. {}".format(count))

print ("End of while loop")
```

On executing, this code will produce the following output −

Iteration no. 1
Iteration no. 2
Iteration no. 3
Iteration no. 4
Iteration no. 5
End of while loop

infinite while Loop

If the condition of a `while` loop always evaluates to `True`, the loop runs continuously, forming an **infinite while loop**. For example,

```
age = 32
# The test condition is always True
while age > 18:
    print('You can vote')
```

Output

```
You can vote
You can vote
You can vote
.
.
```

The break Statement

The `break` statement terminates the `for` loop immediately before it loops through all the items.

For example,

```
languages = ['Swift', 'Python', 'Go', 'C++']

for lang in languages:
    if lang == 'Go':
        break
    print(lang)
```
Run Code

Output

```
Swift
Python
```

Here, when lang is equal to 'Go', the break statement inside the if condition executes which terminates the loop immediately. This is why Go and C++ are not printed.

The continue Statement :

skips the current iteration of the loop and continues with the next iteration.

For example,

```
languages = ['Swift', 'Python', 'Go', 'C++']
for lang in languages:
    if lang == 'Go':
        continue
    print(lang)
```

Output

```
Swift
Python
C++
```

While Loop Control Statements in Python

Continue Statement:

Continue statement written inside the body of while loop returns the flow of control back to the expression in the header, skipping the execution of the rest of the statements inside the body.

Example:

```
count = 0
while count < 5:
 count += 1;
 if(count == 2): continue
 print("The value of the count is " + str(count))
```

Output:

```
The value of the count is 1

The value of the count is 3
The value of the count is 4
```

-
-
-
-
-
-
-

- **Difference between Break and Continue in Python**

Basis for comparison	Break	Continue
Use	It is used for the termination of all the remaining iterations of the loop.	It is used for the termination of the only current iteration of the loop.
Control after using break/continue statement	The line which is just after the loop will gain control of the program.	The control will pass to the next iteration of that current loop by skipping the current iteration.
Causes	It performs the termination of the loop.	It performs early execution of the next loop by skipping the current one.
Continuation	It stops the continuation of the loop.	It stops the execution of the current iteration.
Other	It can be used with labels and switches.	It can't be used with labels and switches.

What is For Else and While Else in Python

For-else and while-else are helpful features provided by Python. You can use the else block just after the for and while loop. Otherwise, the block will be executed only if a break statement doesn't terminate the loop.

Simply put, if a loop is executed successfully without termination, then the else block will be executed.

Let’s try to understand this -

A loop must complete all its iterations so that the else block can be executed, but the else block will be executed only once.

Syntax

```
for i in range(n) :
  #code
else:
  #code
```

In the above example, for loop will execute **n times** and then else block will execute.

While-else

```
while condition:
  #code
else:
  #code
```

In the above example, while loop will execute until the condition is satisfied and then else block will be executed.

Scenarios to use For Else and While Else

Now, we will see three such scenarios where for else and while else are used.

1. Search Programs

Program to find a fruit ‘mango’ in the list of fruits using for-else and while-else.

For-Else Program:

```
my_list = ["papaya","banana","pineapple","mango","grapes"]

# iterating through the fruit list

for i in my_list:

   if i=="mango":
      print("mango found!")
      break

else:
   print("mango not found!")
```

Output:

```
mango found!
```

While-Else Program

```
# list of fruits
my_list = ["papaya","banana","pineapple","mango","grapes"]

size = len(my_list) #length/size of the list
i=0

# iterating through the fruit list
while i<size:
   if  my_list[i] == 'mango':
      print("mango found!")
      break
   i+=1
else:
   print("mango not found!")
```

Output:

```
mango found!
all the elements in the list are in the limit
```

3. Help to Break Out of Nested Loops

When using nested loops in python, it's often necessary to break out of all loops when a particular condition is met, not just the innermost one.**Example:**

```
numbers = [4, 6, 8, 9, 11, 16]

for num in numbers:
    if num > 1:
        for i in range(2, num):
            if (num % i) == 0:
                break
        else:
            print(f"The first prime number in the list is: {num}")
            break
else:
    print("No prime numbers found in the list.")
```

Output:

```
The first prime number in the list is: 11
```

When to use For Loop and While Loop in Python?

To decide when should you use for loop and when to use while loop, you can think of these situations and you will the answer:

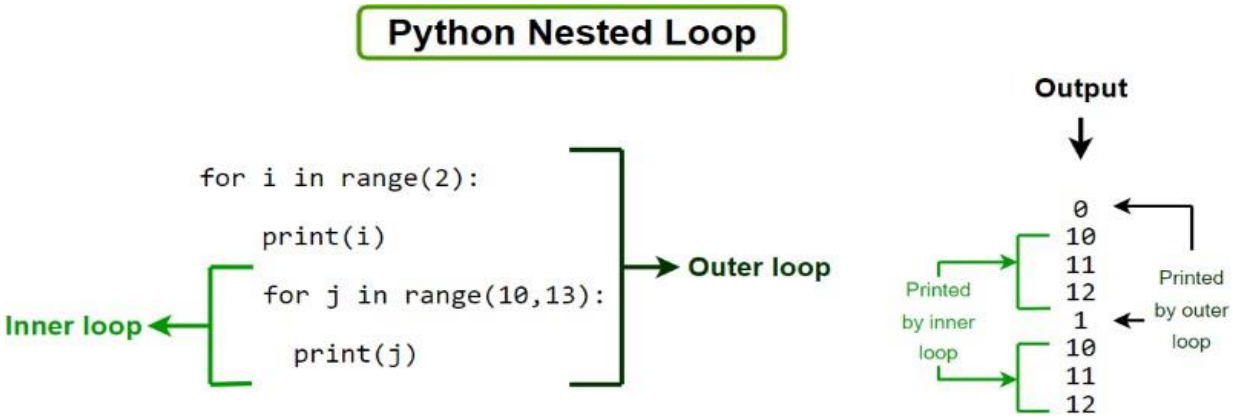
For Loop

- When there is a need of iteration over elements that are in sequence like lists, tuples, strings or dictionaries.
- Use for loops when a known number or a finite number of iteration is required.
- Another situation when you want that each element of a sequence should pass through iteration one by one.
- for loops can also be used when a task requires a sequential access of elements of a sequence.

While Loop

- When there is a need to repeat execution of a block of code until a specified condition becomes False.
- Use while loops when the number of iteration is not determined.
- while loops are can be used when there is a need to repeat execution based on a condition.
- When dynamic adaptation is required based upon condition that may change.
-
- **Here is a comparison table differentiating for loop and while loop:**

Aspect	**For Loop**	**While Loop**
Syntax	for item in iterable:	while condition:
Iteration behavior	It iterates through the specified sequence by itself.	It depends on the condition, whether it is 'True' or 'False'.
Number of iterations	The number of iterations is known or finite.	The number of iterations is not known.
Execution	Execution depends on the specific number of iterations.	It repeats the execution based on the specified condition.
Termination	It will terminate once all elements have gone through iteration.	It may result in an infinite loop if the condition never evaluates to 'False'.



Printing multiplication table using Python nested for loops

 # Running outer loop from 2 to 3

```
for i in range(2, 4):
    # Printing inside the outer loop
    # Running inner loop from 1 to 10
    for j in range(1, 11):
        # Printing inside the inner loop
        print(i, "*", j, "=", i*j)
    # Printing inside the outer loop
    print()
```

output:

```
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
```

```
2 * 10 = 20



3 * 1 = 3
3 * 2 = 6
3 * 3 = 9
3 * 4 = 12
3 * 5 = 15
3 * 6 = 18
3 * 7 = 21
3 * 8 = 24
3 * 9 = 27
3 * 10 = 30
```

```
# outer loop
for i in range(1, 11):
    # nested loop
    # to iterate from 1 to 10
    for j in range(1, 11):
        # print multiplication
        print(i * j, end=' ')
    print()
```

Output:

```
1 2 3 4 5 6 7 8 9 10

2 4 6 8 10 12 14 16 18 20

3 6 9 12 15 18 21 24 27 30

4 8 12 16 20 24 28 32 36 40

5 10 15 20 25 30 35 40 45 50

6 12 18 24 30 36 42 48 54 60
```

```
7 14 21 28 35 42 49 56 63 70

8 16 24 32 40 48 56 64 72 80

9 18 27 36 45 54 63 72 81 90

10 20 30 40 50 60 70 80 90 100
```

Print: List of Elements

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

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

output:

red apple

red banana

red cherry

big apple

big banana

big cherry

tasty apple

tasty banana

tasty cherry

Python Nested while Loop Syntax

The syntax for a **nested while loop** statement in Python programming language is as follows −

```
while expression:
   while expression:
      statement(s)
   statement(s)
```

Python Program

```
i = 1
while i <= 4 :
    j = 0
    while  j <= 3 :
        print(i*j, end=" ")
        j += 1
    print()
    i += 1
```

Output

```
0 1 2 3
0 2 4 6
0 3 6 9
0 4 8 12
```

Example 2 – 3 Level Nested While Loop

In this example, we shall write a nested while loop inside another while loop. Like three while loops one inside another.

Python Program

```
i = 1
while i <= 4 :
    j = 0
    while  j <= 3 :
        k = 0
        while  k <= 5 :
            print(i*j*k, end=" ")
            k += 1
        print()
        j += 1
    print()
    i += 1
```

Output

```
0 0 0 0 0 0
0 1 2 3 4 5
0 2 4 6 8 10
0 3 6 9 12 15

0 0 0 0 0 0
0 2 4 6 8 10
0 4 8 12 16 20
0 6 12 18 24 30

0 0 0 0 0 0
0 3 6 9 12 15
0 6 12 18 24 30
```

0 9 18 27 36 45

0 0 0 0 0 0
0 4 8 12 16 20
0 8 16 24 32 40
0 12 24 36 48 60

Common Python Syntax errors:

- Incorrect indentation
- Missing colon, comma, or brackets
- Putting keywords in the wrong place.

Example

```
print("Welcome to PYnative")
    print("Learn Python with us..")
```

Output:

```
print("Learn Python with us..")
    ^
IndentationError: unexpected indent
```

Logical errors (Exception)

Even if a statement or expression is syntactically correct, the error that occurs at the runtime is known as a **Logical error or Exception**. In other words, **Errors detected during execution are called exceptions**.

Common Python Logical errors:

- Indenting a block to the wrong level
- using the wrong variable name
- making a mistake in a boolean expression

Example

```
a = 10
b = 20
print("Addition:", a + c)
```

Output

```
print("Addition:", a + c)

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

What are Exceptions in Python?

Unlike errors, exceptions in Python occur at runtime, when the code is executed. Exceptions are errors that are discovered during the execution of a program.

Exceptions occur when the code tries to perform an invalid operation, such as dividing by zero or accessing an invalid variable.

Table difference between errors and exceptions in python

Errors	Exceptions
Errors refer to syntax and logical errors in code that prevent it from running.	Exceptions refer to runtime errors that occur during program execution.

Errors can be identified during the compilation phase or before the program execution.	Exceptions are identified only during the program execution phase when a specific statement is executed.
Examples of errors include syntax errors, type errors or missing dependencies.	Examples of exceptions include IndexError, ValueError, KeyError, and NameError etc.
Errors are typically fatal and prevent the program from running whereas exceptions can be caught and handled within the program.	Exceptions allow the program to continue to run and provide an opportunity for the programmer to handle and recover from the error.

Common examples of exceptions in Python include:

- ZeroDivisionError: If the code tries to divide a number by zero, then the ZeroDivisionError occurs.
- TypeError: If the code tries to perform an operation on an incompatible data type, then the TypeError occurs.
- IndexError: If the code tries to access an invalid index of a list, then the IndexError occurs.
- KeyError: If the code tries to access an invalid key in a dictionary, then the KeyError occurs.

IndexError

The `IndexError` is thrown when trying to access an item at an invalid index.

Example: IndexError

Copy

```
>>> L1=[1,2,3]
>>> L1[3]

Traceback (most recent call last):
File "<pyshell#18>", line 1, in <module>

L1[3]
IndexError: list index out of range
```

ModuleNotFoundError

The `ModuleNotFoundError` is thrown when a module could not be found.

Example: ModuleNotFoundError

```
>>> import notamodule

Traceback (most recent call last):
File "<pyshell#10>", line 1, in <module>
import notamodule
ModuleNotFoundError: No module named 'notamodule'
```

KeyError

The `KeyError` is thrown when a key is not found.

Example: KeyError

```
>>> D1={'1':"aa", '2':"bb", '3':"cc"}
>>> D1['4']

Traceback (most recent call last):
File "<pyshell#15>", line 1, in <module>
D1['4']
KeyError: '4'
```

ImportError

The `ImportError` is thrown when a specified function can not be found.

Example: ImportError

```
>>> from math import cube

Traceback (most recent call last):
File "<pyshell#16>", line 1, in <module>

from math import cube
ImportError: cannot import name 'cube'
```

TypeError

The `TypeError` is thrown when an operation or function is applied to an object of an inappropriate type.

Example: TypeError

```
>>> '2'+2
```

```
Traceback (most recent call last):
File "<pyshell#23>", line 1, in <module>
'2'+2
TypeError: must be str, not int
```

ValueError

The `ValueError` is thrown when a function's argument is of an inappropriate type.

Example: ValueError

```
>>> int('xyz')


Traceback (most recent call last):
File "<pyshell#14>", line 1, in <module>

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

NameError

The `NameError` is thrown when an object could not be found.

Example: NameError

```
>>> age

Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>

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

ZeroDivisionError

The `ZeroDivisionError` is thrown when the second operator in the division is zero.

Example: ZeroDivisionError

```
>>> x=100/0

Traceback (most recent call last):
File "<pyshell#8>", line 1, in <module>
x=100/0
ZeroDivisionError: division by zero
```

KeyboardInterrupt

The `KeyboardInterrupt` is thrown when the user hits the interrupt key (normally Control-C) during the execution of the program.

Example: KeyboardInterrupt

```
>>> name=input('enter your name')


enter your name^c
Traceback (most recent call last):
File "<pyshell#9>", line 1, in <module>
name=input('enter your name')
Output: KeyboardInterrupt
```

Python Exception Handling

In the last tutorial, we learned about Python exceptions. We know that exceptions abnormally terminate the execution of a program.
This is why it is important to handle exceptions. In Python, we use the `try...except` block

Python try...except Block

The `try...except` block is used to handle exceptions in Python. Here's the syntax of `try...except` block:

```
try:
    # code that may cause exception
except:
    # code to run when exception occurs
```

How try() works?

- First, the **try** clause is executed i.e. the code between **try.**
- If there is no exception, then only the **try** clause will run, **except** clause is finished.
- If any exception occurs, the **try** clause will be skipped and **except** clause will run.
- If any exception occurs, but the **except** clause within the code doesn't handle it, it is passed on to the outer **try** statements. If the exception is left unhandled, then the execution stops.
- A **try** statement can have more than one **except** clause

Example: Exception Handling Using try...except

```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

# Output: Error: Denominator cannot be 0.
Run Code
```

In the example, we are trying to divide a number by **0**. Here, this code generates an exception.
To handle the exception, we have put the code, `result = numerator/denominator` inside the `try` block.

Now when an exception occurs, the rest of the code inside the `try` block is skipped.
The `except` block catches the exception and statements inside the `except` block are executed.

Python try...finally

In Python, the `finally` block is always executed no matter whether there is an exception or not.
The `finally` block is optional. And, for each `try` block, there can be only one `finally` block.

Let's see an example,

```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

finally:
    print("This is finally block.")
```

Output

```
Error: Denominator cannot be 0.
This is finally block.
```

In the above example, we are dividing a number by **0** inside the `try` block. Here, this code generates an exception.
The exception is caught by the `except` block. And, then the `finally` block is executed.

Else Clause

In Python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

Syntax:

```
try:
    # Some Code
except:
    # Executed if error in the
    # try block
else:
    # execute if no exception
```

Example:

```
# Function which returns a/b
def AbyB(a , b):
        try:
                c = ((a+b) // (a-b))
        except ZeroDivisionError:
                print ("a/b result in 0")
        else:
                print (c)
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

Output:

```
-5.0
a/b result in 0
```

Finally Keyword in Python

Python provides a keyword finally, which is always executed after the try and except blocks. The final block always executes after the normal termination of the try block or after the try block terminates due to some exceptions.

Syntax:

try:

    # Some Code

except:

    # Executed if error in the

    # try block

else:

    # execute if no exception

finally:

    # Some code .....(always executed)

Example:

```
try:
    k = 5//0 # raises divide by zero exception.
```

```
    print(k)

    # handles zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")
finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')
```

Output:

Can't divide by zero

This is always executed

Multiple exceptions

You can use multiple `except` clauses to handle different exceptions with different operations.

```
def divide_each(a, b):
    try:
        print(a / b)
    except ZeroDivisionError as e:
        print('catch ZeroDivisionError:', e)
    except TypeError as e:
        print('catch TypeError:', e)
```

```
divide_each(1, 0)
# catch ZeroDivisionError: division by zero

divide_each('a', 'b')
# catch TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

You can also use the `else` and `finally` clause together. If no exception occurs, the `else` clause is executed and then the `finally` clause is executed.

```
def divide_else_finally(a, b):
    try:
        print(a / b)
    except ZeroDivisionError as e:
        print('catch ZeroDivisionError:', e)
    else:
        print('finish (no error)')
    finally:
        print('all finish')

divide_else_finally(1, 2)
# 0.5
# finish (no error)
# all finish

divide_else_finally(1, 0)
# catch ZeroDivisionError: division by zero
# all finish
```

Python - Functions

A function is a block of code that performs a specific task.

A Python function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

A Python function may be invoked from any other function by passing required data (called **parameters** or **arguments**). The called function returns its result back to the calling environment.

Types of Python Functions

There are two types of function in Python programming:

- **Standard library functions** - These are built-in functions in Python that are available to use.
- **User-defined functions** - We can create our own functions based on our requirements.

Python Function Declaration

The syntax to declare a function is:

```
def function_name(arguments):
    # function body

    return
```

Here,

- `def` - keyword used to declare a function
- `function_name` - any name given to the function
- `arguments` - any value passed to function
- `return` (optional) - returns value from a function

Return Value from a Function

In Python, to return value from the function, a `return` statement is used. It returns the value of the expression following the returns keyword.

Syntax of `return` statement

```
def fun():
    statement-1
    statement-2
    statement-3
    .
    .
    return [expression]
```

The return value is nothing but a outcome of function.

- For a function, it is not mandatory to return a value.
- If a `return` statement is used without any expression, then the `None` is returned.
- The `return` statement should be inside of the function block.

Here, we have created a function named `greet()`. It simply prints the text `Hello World!`.This function doesn't have any arguments and doesn't return any values.

Let's see an example,

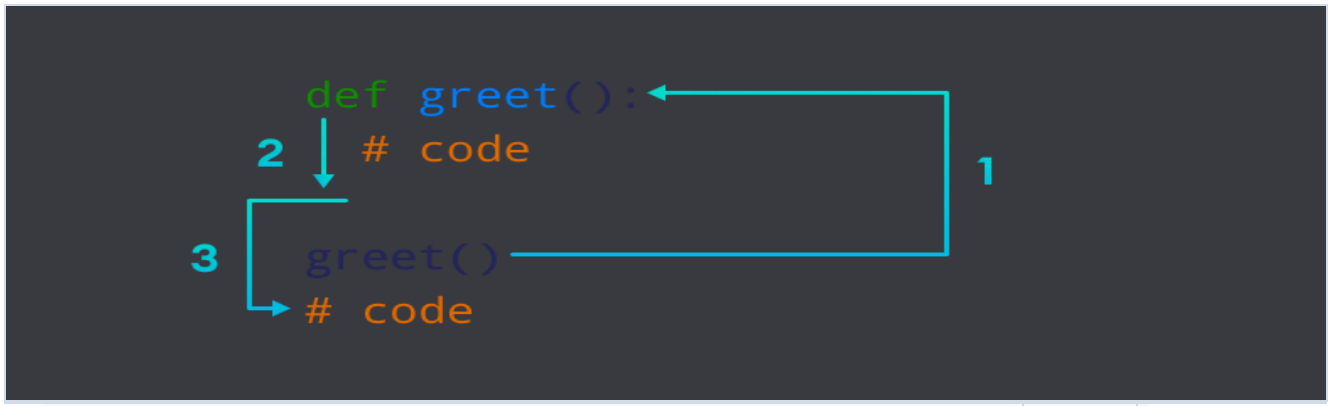
```
def greet():
    print('Hello World!')
```

Calling a Function in Python

```
def greet():
    print('Hello World!')

# call the function
greet()
```

Output

```
Hello World!
```



In the above example, we have created a function named `greet()`.

Here's how the program works:

Here,

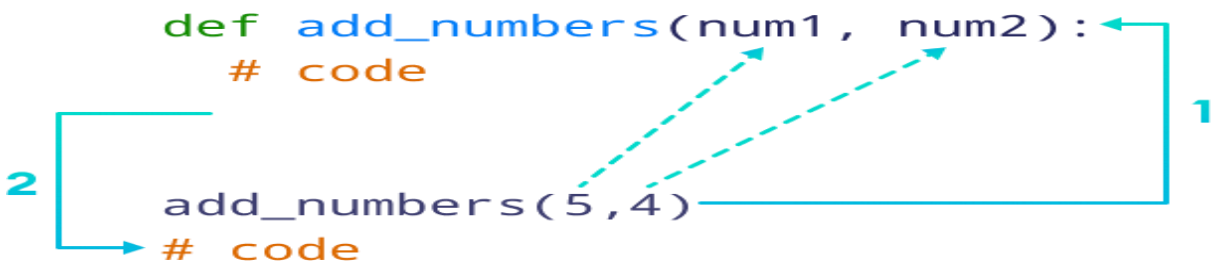
- When the function is called, the control of the program goes to the function definition.
- All codes inside the function are executed.
- The control of the program jumps to the next statement after the function call.

Python Function Arguments

As mentioned earlier, a function can also have **arguments**. An argument is a value that is accepted by a function.

```
# function with two arguments
def add_numbers(num1, num2):
    sum = num1 + num2
    print("Sum: ",sum)

# function call with two values
add_numbers(5, 4)

# Output: Sum: 9
```

In the above example, we have created a function named `add_numbers()` with arguments: *num1* and *num2*.



Python Function with Arguments

We can also call the function by mentioning the argument name as:

```
add_numbers(num1 = 5, num2 = 4)
```

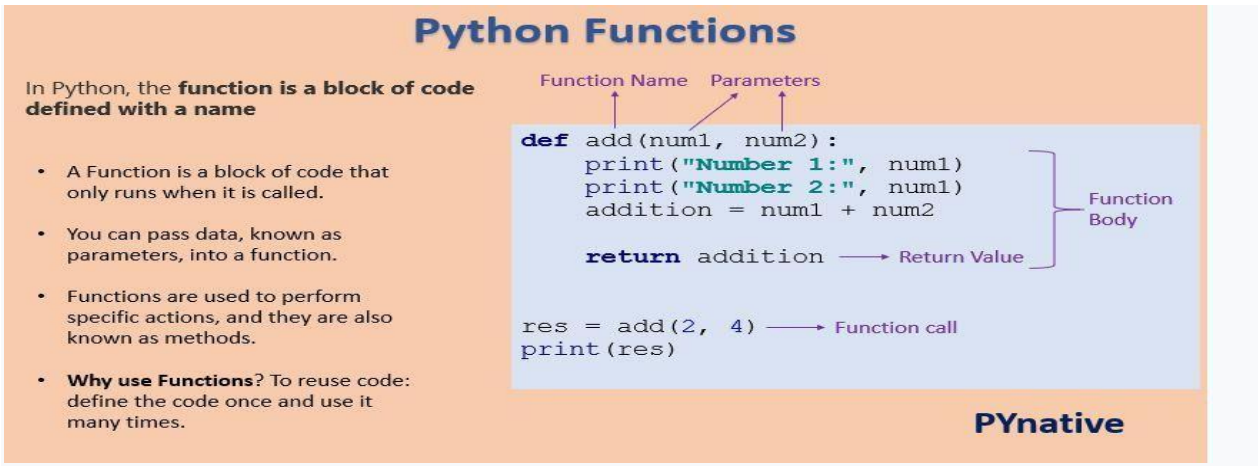
In Python, we call it Keyword Argument (or named argument). The code above is equivalent to

```
add_numbers(5, 4)
```

The return Statement in Python A Python function may or may not return a value. If we want our function to return some value to a function call, we use the `return` statement. **For example,**

```
def add_numbers():
    ...
    return sum
```

Here, we are returning the variable `sum` to the function call.



Example 3: Add Two Numbers with return statment

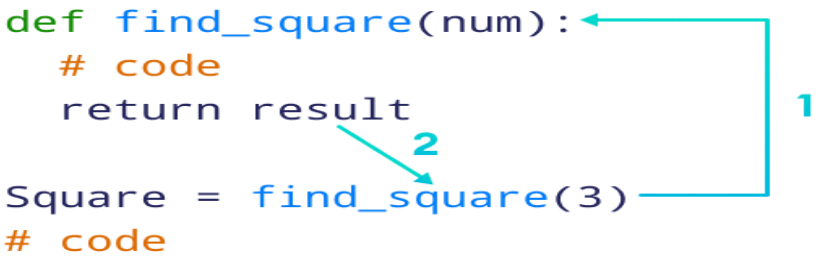
```
# function that adds two numbers
def add_numbers(num1, num2):
    sum = num1 + num2
    return sum

# calling function with two values
result = add_numbers(5, 4)

print('Sum: ', result)

# Output: Sum: 9
Run Code
```

Example 2: Function return Type

```
# function definition
def find_square(num):
    result = num * num
    return result

# function call
```

```
square = find_square(3)

print('Square:',square)

# Output: Square: 9
Run Code
```

In the above example, we have created a function named `find_square()`. The function accepts a number and returns the square of the number.



Working of functions in Python

Define a function that accepts 2 values and return its sum, subtraction and multiplication.

```
def result(a, b):
    sum = a+b
    sub = a-b
    mul = a*b
    print(f"Sum is {sum}, Sub is {sub}, & Multiply is {mul}")
    return
a = int(input("Enter value of a: "))  # 7
b = int(input("Enter value of b: ")) # 5
result(a,b)
```

Define a function that accepts radius and returns the area of a circle.

```
def area(radius):
    area = 3.14*radius*radius
    return area
    radius = int(input("Enter Radius: "))  # 4
print(area(radius))
```

Define a function in python that accepts 3 values and returns the maximum of three numbers.

```
def max(a, b, c):
    if a > b and a > c:
        print(f"{a} is maximum among all")
    elif b > a and b > c:
        print(f"{b} is maximum among all")
    else:
        print(f"{c} is maximum among all")

max(30,22,18)
```

Define a function that accepts a number and returns whether the number is even or odd.

```
def func(x):
    if x % 2 == 0:
        print(f"{x} is Even number")
    else:
        print(f"{x} is Odd number")
x = int(input("Enter a number "))
func(x)
```

Advantages of functions in Python

- ***Helps in increasing modularity of code*** – Functions in python help the user to divide the program into smaller parts and solve them individually, thus making it easier to implement.
- ***Minimizes Redundancy*** – Python functions help us to save the effort of rewriting the whole code. All we got to do is call the function once it is defined.
- ***Maximizes Code Reusability*** – Once a function is defined in python, it can be called as many times as needed, thus enhancing code reusability.
- ***Improves Clarity of Code*** – Since a large program is divided into sections with the help of functions, it helps increase the readability of code while ensuring easy debugging.

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python –

- **Global variables**
- **Local variables**

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope.**Example**

```
total = 0; # This is global variable.

# Function definition is here
def sum( arg1, arg2 ):

   # Add both the parameters and return them."
   total = arg1 + arg2; # Here total is local variable.

print ("Inside the function local total : ", total)
   return total;

# Now you can call sum function
sum( 10, 20 );
print ("Outside the function global total : ", total)
```

When the above code is executed, it produces the following result –

Inside the function **local** total : 30
Outside the function **global** total : 0

Difference Between Local and Global Variables: Local Variables vs Global Variables

Parameter	Local Variables	Global Variables
Definition	Defined inside a function or block.	Defined outside of all functions or blocks.
Scope	Accessible only within the function/block where it's defined.	Accessible throughout the entire program.
Lifetime	Exists only during the function's execution.	Remains in memory for the duration of the program.
Accessibility	Cannot be accessed outside its function.	Can be accessed and modified by any function in the program.
Keyword for Modification	No special keyword is required.	Use the **global** keyword to modify it inside a function.
Memory Storage	Stored in the stack.	Stored in the data segment of memory.
Risk of Unintended Modification	Low, as it's confined to its function.	Higher, as any function can modify it.

Python Library Functions

In Python, standard library functions are the built-in functions that can be used directly in our program.

For example,

- `print()` - prints the string inside the quotation marks
- `sqrt()` - returns the square root of a number
- `pow()` - returns the power of a number

These library functions are defined inside the module. And, to use them we must include the module inside our program.

For example, `sqrt()` is defined inside the `math` module.

Example 4: Python Library Function

```
import math

# sqrt computes the square root
square_root = math.sqrt(4)
print("Square Root of 4 is",square_root)
# pow() comptes the power
power = pow(2, 3)

print("2 to the power 3 is",power)
```

Run Code

Output

```
Square Root of 4 is 2.0
2 to the power 3 is 8
```

In the above example, we have used

- `math.sqrt(4)` - to compute the square root of **4**
- `pow(2, 3)` - computes the power of a number i.e. 2^3

Here, notice the statement,

```
import math
```

Since `sqrt()` is defined inside the `math` module, we need to include it in our program.

Types of function arguments

There are various ways to use arguments in a function. In Python, we have the following **4 types of function arguments**.

1. Default argument
2. Keyword arguments (named arguments)
3. Position Arguent
4. Arbitrary arguments (variable-length arguments `*args` and `**kwargs`)

Default Arguments

In a function, arguments can have default values. We assign default values to the argument using the '=' (assignment) operator at the time of function definition. You can define a function with any number of default arguments.

Example:

Let's define a function student() with four arguments name, age, grade, and school. In this function**, grade and school are default arguments with default values.**

- If you call a **function without** school and grade arguments, then the default values of grade and school are used.
- The age and name parameters do not have default values and are required **(mandatory**) during a function call.

```
# function with 2 keyword arguments grade and school
def student(name, age, grade="Five", school="ABC School"):
    print('Student Details:', name, age, grade, school)

# without passing grade and school
# Passing only the mandatory arguments
student('Jon', 12)

# Output: Student Details: Jon 12 Five ABC School
```

Passing one of the default arguments:

If you pass values of the grade and school arguments while calling a function, then those values are used instead of default values.

Example:

function with 2 keyword arguments grade and school

def student(name, age, grade="Five", school="ABC School"):

    print('Student Details:', name, age, grade, school)

student('Kelly', 12, 'Six')

passign all arguments

student('Jessa', 12, 'Seven', 'XYZ School')

Output:

```
Student Details: Kelly 12 Six ABC School

Student Details: Jessa 12 Seven XYZ School
```

Keyword Arguments

Keyword arguments are those arguments where **values get assigned to the arguments by their keyword (name)** when the function is called. It is preceded by the variable name and an (=) assignment operator. The Keyword Argument is also called a named argument.

Example:

```
# function with 2 keyword arguments
def student(name, age):
    print('Student Details:', name, age)

# default function call
student('Jessa', 14)

# both keyword arguments
student(name='Jon', age=12)

# 1 positional and 1 keyword
student('Donald', age=13)
```

Output:

```
Student Details: Jessa 14

Student Details: Jon 12

Student Details: Donald 13
```

Change the sequence of keyword arguments

Also, you can change the sequence of keyword arguments by using their name in function calls.

Python allows functions to be called using keyword arguments. But all the keyword arguments should match the parameters in the function definition. When we call functions in this way, the order (position) of the arguments can be changed.

Example:

```
# both keyword arguments by changing their order
student(age=13, name='Kelly')

# Output: Student Details: Kelly 13
```

Positional Arguments

Positional arguments are those arguments where values get assigned to the arguments by their position when the function is called.

For example, the 1st positional argument must be 1st when the function is called. The 2nd positional argument needs to be 2nd when the function is called, etc.

By default, Python functions are called using the positional arguments.

Example: Program to subtract 2 numbers using positional arguments.

```
def add(a, b):
    print(a - b)

add(50, 10)
# Output 40

add(10, 50)
# Output -40
```

Run

Note: If you try to pass more arguments, you will get an error.

```
def add(a, b):
    print(a - b)

add(105, 561, 4)
```

Output

```
TypeError: add() takes 2 positional arguments but 3 were given
```

Note: In the positional argument number and position of arguments must be matched. If we change the order, then the result may change. Also, If we change the number of arguments, we will get an error.

Variable-length arguments

In Python, sometimes, there is a situation where we need to **pass multiple arguments to the function**. Such types of arguments are called arbitrary arguments or variable-length arguments.We **use variable-length arguments if we don't know the number of arguments needed for the function in advance**.

Types of Arbitrary Arguments:

- arbitrary positional arguments (*args)
- arbitrary keyword arguments (**kwargs)

The *args and **kwargs allow you to pass multiple positional arguments or keyword arguments to a function.

Arbitrary positional arguments (*args)

We can declare a variable-length argument with the *** (asterisk)** symbol. Place an asterisk (*) before a parameter in the function definition to define an arbitrary positional argument.we can pass multiple arguments to the function. Internally all these values are represented in the form of a tuple. Let's understand the use of variable-length arguments with an example.

This is a simple function that takes three arguments and returns their average:

```
def percentage(sub1, sub2, sub3):
    avg = (sub1 + sub2 + sub3) / 3
    print('Average', avg)

percentage(56, 61, 73)
```

This function works, but it's limited to only three arguments. What if you need to calculate the **average marks of more than three subjects or the number of subjects is determined only at runtime**?

In such cases, it is advisable to use the **variable-length of positional arguments** to write a function that could calculate the average of all subjects no matter how many there are.

Ex:

```
# Program to add and display sum of n number of integer
def add(*num):
    sum = 0;
    for n in num:
        sum = sum + n;
    print("Sum:", sum)

add(3,4,5,6,7)
add(1,2,3,5,6,7,8)



SUM:25
SUM:32
```

Arbitrary keyword arguments (kwargs) or** keyword variable-length argument (**variable) in Python:

Just as variable length arguments, there are keyword variable length arguments that are n key-value pairs.

The syntax is given below.

```
def m1(**x):
Body of function
```

Example: keyword variable-length argument (variable)**

```
def m1(**x):
for k, v in x.items():
    print(k,"=",v)
m1(a=10, b=20, c=30)
m1(id=100, name="Subbalaxmi")
```

Output:

a=10,b=20,c=30

id=100,name=subbalaxmi

Function vs Module vs Library in Python:

1. A group of lines with some name is called a function
2. A group of functions saved to a file is called Module
3. A group of Modules is nothing but Library

Python Lambda Functions

- **Python Lambda Functions** are anonymous functions means that the function is without a name. As we already know the *def* keyword is used to define a normal function in Python. Similarly, the *lambda* keyword is used to define an anonymous function in Python.

  Let's explore Lambda Function in detail:

  Python Lambda Function Syntax

  Syntax: *lambda arguments : expression*

  - **lambda:** The keyword to define the function.
  - **arguments:** A comma-separated list of input parameters (like in a regular function).
  - **expression:** A single expression that is evaluated and returned.

  Let's see some of the practical uses of the Python lambda function.

  example of a lambda function that adds 2 numbers and return one result. The code is shown below -

```
1. a = lambda x, y, z : (x + y + z)
2. print(a(4, 5, 5))
```

  Output:

  Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

  14

example of a lambda function that multiply 2 numbers and return one result. The code is shown below -

```
1. a = lambda x, y : (x * y)
2. print(a(4, 5))
```

Output:

Now we compile the above code in python, and after successful compilation, we run it. Then the output is given below -

```
20
```

lambda with Condition Checking

A lambda function can include conditions using if statements.

Example:

```
# Example: Check if a number is positive, negative, or zero
n = lambda x: "Positive" if x > 0 else "Negative" if x < 0 else "Zero"

print(n(5))
print(n(-3))
print(n(0))
```

Output

```
Positive
Negative
Zero
```

Explanation:

- The lambda function takes **x** as input.
- It uses nested if-else statements to return “Positive,” “Negative,” or “Zero.”

Difference Between lambda and def Keyword
lambda is concise but less powerful than def when handling complex logic. Let's take a look at short comparison between the two:

Feature	lambda Function	Regular Function (def)
Definition	Single expression with lambda.	Multiple lines of code.
Name	Anonymous (or named if assigned).	Must have a name.
Statements	Single expression only.	Can include multiple statements.
Documentation	Cannot have a docstring.	Can include docstrings.
Reusability	Best for short, temporary functions.	Better for reusable and complex logic.

Example:

```
# Using lambda
sq = lambda x: x ** 2
print(sq(3))

# Using def
def sqdef(x):
    return x ** 2
print(sqdef(3))
```

Output

```
9

9
```

As we can see in the above example, both the **sq()** function and **sqdef()** function behave the same and as intended.

Lambda with List Comprehension

Combining lambda with list comprehensions enables us to apply transformations to data in a concise way.

Example:

```
li = [lambda arg=x: arg * 10 for x in range(1, 5)]
for i in li:
    print(i())
```

Output

```
10

20

30

40
```

Explanation:

- The lambda function squares each element.
- The list comprehension iterates through li and applies the lambda to each element.
- This is ideal for applying transformations to datasets in data preprocessing or manipulation tasks.

Lambda with if-else

lambda functions can incorporate conditional logic directly, allowing us to handle simple decision making within the function.

Example:

```
# Example: Check if a number is even or odd
check = lambda x: "Even" if x % 2 == 0 else "Odd"

print(check(4))
print(check(7))
```

Output

```
Even

Odd
```

Explanation:

- The lambda checks if a number is divisible by 2 (x % 2 == 0).
- Returns “Even” for true and “Odd” otherwise.
- This approach is useful for labeling or categorizing values based on simple conditions.

Lambda with Multiple Statements

Lambda functions do not allow multiple statements, however, we can create two lambda functions and then call the other lambda function as a parameter to the first function.

Example:

```
# Example: Perform addition and multiplication in a single line
calc = lambda x, y: (x + y, x * y)

res = calc(3, 4)
print(res)
```

Output

```
(7, 12)
```

Explanation:

- The lambda function performs both addition and multiplication and returns a tuple with both results.
- This is useful for scenarios where multiple calculations need to be performed and returned together.

Lambda functions can be used along with built-in functions like filter(), map() and reduce().

Using lambda with filter()

The filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence “sequence”, for which the function returns True.

Example:

```
# Example: Filter even numbers from a list
n = [1, 2, 3, 4, 5, 6]
even = filter(lambda x: x % 2 == 0, n)
print(list(even))
```

Output

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

Explanation:

- The lambda function checks if a number is even (x % 2 == 0).
- filter() applies this condition to each element in nums.

Using lambda with map()

The map() function in Python takes in a function and a list as an argument. The function is called with a lambda function and a new list is returned which contains all the lambda-modified items returned by that function for each item.

Example:

```
# Example: Double each number in a list
a = [1, 2, 3, 4]
b = map(lambda x: x * 2, a)
print(list(b))
```

Output

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

Explanation:

- The lambda function doubles each number.
- map() iterates through a and applies the transformation.

Using lambda with reduce()

The reduce() function in Python takes in a function and a list as an argument. The function is called with a lambda function and an iterable and a new reduced result is returned. This performs a repetitive operation over the pairs of the iterable. The reduce() function belongs to the ***functools*** module.

Example:

```
from functools import reduce

# Example: Find the product of all numbers in a list
a = [1, 2, 3, 4]
b = reduce(lambda x, y: x * y, a)
print(b)
```

Output

```
24
```

Explanation:

- The lambda multiplies two numbers at a time.
- reduce() applies this operation across the list.

lambda function Problems

Problem 1: Square of a Number

Task: Write a **lambda function** that takes a number and returns its square.
Example: square(5) should return 25.

Problem 2: Add Two Numbers

Task: Write a lambda function that takes two numbers and returns their sum.

Problem 3: Check Even or Odd

Task: Write a lambda function that checks if a number is even.
Hint: It should return True for even numbers, False otherwise.

Problem 4: Find the Maximum of Two Numbers

Task: Write a lambda function that takes two numbers and returns the larger one.

Problem 5: Convert List of Celsius to Fahrenheit

Task: Given a list of temperatures in Celsius, use map() with a lambda to convert them to Fahrenheit.

Problem 6: Filter Even Numbers from a List

Task: Use filter() with a lambda function to extract even numbers from a list.

Problem 7: Multiply Each Element by 3

Task: Given a list of numbers, use map() and a lambda to multiply each element by 3.

Problem 8: Add two lists element-wise by lambda functon

Problem 9: Get students who passed (marks > 50) by lambda functon

Problem 10: Find maximum element in a list by lambda functon

Problem 11: Square even numbers and sum them by lambda funct

```
# Problem 1: Square of a Number
print("Problem 1: Square of a Number")
square = lambda x: x ** 2
print("Square of 5 is:", square(5))

# Problem 2: Add Two Numbers
print("Problem 2: Add Two Numbers")
add = lambda x, y: x + y
print("Sum of 3 and 4 is:", add(3, 4))

# Problem 3: Check Even or Odd
print("Problem 3: Check Even or Odd")
is_even = lambda x: x % 2 == 0
print("Is 6 even?", is_even(6))
print("Is 7 even?", is_even(7))
```

 # Problem 4: Find the Maximum of Two Numbers

```
print("Problem 4: Maximum of Two Numbers")
maximum = lambda x, y: x if x > y else y
print("Max of 10 and 20 is:", maximum(10, 20))
```

 # Problem 5: Convert List of Celsius to Fahrenheit

```
print("Problem 6: Celsius to Fahrenheit")
celsius = [0, 20, 37, 100]
fahrenheit = list(map(lambda c: (c * 9/5) + 32, celsius))
print("Fahrenheit temperatures:", fahrenheit)
```

 # Problem 6: Filter Even Numbers from a List

```
print("Problem 7: Filter Even Numbers")
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print("Even numbers:", even_numbers)
```

 # Problem 7: Multiply Each Element by 3

```
print("Problem 8: Multiply Each Element by 3")
original_list = [1, 2, 3, 4, 5]
multiplied = list(map(lambda x: x * 3, original_list))
print("List after multiplying by 3:", multiplied)
```

Problem 8: Add two lists element-wise by lambda functon

```
a = [1, 2, 3]
b = [4, 5, 6]
sums = list(map(lambda x, y: x + y, a, b))
print(sums)
```

 # Output: [5, 7, 9]

Problem 9: Get students who passed (marks > 50) by lambda functon

```
marks = [20, 55, 78, 30, 90]
passed = list(filter(lambda x: x > 50, marks))
print(passed)
```

 # Output: [55, 78, 90]

Problem 10: Find maximum element in a list by lambda functon

```
from functools import reduce
numbers = [4, 8, 15, 16, 23, 42]
max_num = reduce(lambda a, b: a if a > b else b, numbers)
print(max_num)
```

 # Output: 42

Problem 11: Square even numbers and sum them by lambda functon

```
from functools import reduce
numbers = [1, 2, 3, 4, 5, 6]
# Step 1: Filter even numbers
evens = list(filter(lambda x: x % 2 == 0, numbers))

# Step 2: Square them
squares = list(map(lambda x: x**2, evens))

# Step 3: Sum them
total = reduce(lambda x, y: x + y, squares)
print("Sum of squares of even numbers:", total)

# Output: 56
```

Example : Age Group Classification

```
age_group = lambda age: "Child" if age < 18 else "Adult" if age <= 60 else "Senior"
print(age_group(15))  #  Output:  Child
print(age_group(35))  #  Output:  Adult
print(age_group(75))  # Output: Senior
```

Exception Handling Problems:

Problem 1: Division Handling

Task: Take two inputs from the user and divide them. Handle the case where the second number is zero.

□ **Problem 2: Invalid Integer Input**

Task: Ask the user to enter an integer. If they enter something else (like a letter), catch the error and print an error message.

□ **Problem 3: Try-Except-Else**

Task: Ask the user for a number. Use try, except, and else to handle errors and confirm when the input is valid.

□ **Problem 4: Using Multiple Except Blocks**

Task: Write a program that may raise either ValueError or ZeroDivisionError, and handle them with separate except blocks.

□ **Problem 5: TypeError Handling**

Task: Try to add an integer and a string together. Handle the TypeError.

Problem 6: out of range

Task: declare a list and find element which index is not present in list and handle exception.

Problem 7: out of range

Task: declare a dictionary and find element which is not present in dictionary and handle exception.

Problem 8: Handling KeyError with remove() in Set

 # Problem 1: Division Handling

```
try:
    num1 = float(input("Enter the first number: "))
    num2 = float(input("Enter the second number: "))
    result = num1 / num2
    print("Result:", result)
except ZeroDivisionError:
    print("please enter second number must be nonzero.")
```

 # Problem 2: Invalid Integer Input

```
try:
    user_input = int(input("Enter an integer: "))
    print("You entered:", user_input)
except ValueError:
    print("please inter valid integer  number")
```

 # Problem 3: Try-Except-Else

```
try:
    number = int(input("Enter a number: "))
except ValueError:
    print("Error: Invalid input. Not a number.")
else:
    print("Thank you! You entered:", number)
```

 # Problem 4: Using Multiple Except Blocks

```
try:
```

```
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    print("Result of division:", a / b)
except ValueError:
    print("Error: Invalid input. Please enter integers only.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

 # Problem 5: TypeError Handling

```
try:
    result = 5 + "hello"
except TypeError:
    print(" Cannot add an integer and a string.")
```

 # Problem 6 :Out of Range error:

```
try:
    even_numbers = [2,4,6,8]
    print(even_numbers[5])
except IndexError:
    print("Index Out of Range. Of the list")
```

***keyerror in dictionary:**

```
# Define a dictionary with some key-value pairs
```

```
person_info = {
    "name": "Alice",
    "age": 30,
    "city": "New York",
    "job": "Engineer"
}
```

 # Ask the user to enter a key

```
key = input("Enter the key you want to access (e.g., name, age, city, job): ")
# Try to access the key and handle the case when the key doesn't exist
try:
    a = person_info[key]
    print(f"The value for '{key}' is: {a}")
```

except KeyError:

```
    print(f"Error: The key '{key}' does not exist in the dictionary.")
```

* KeyError with remove() in Set

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

try:

```
    s.remove(4)  # Trying to remove a non-existent item
```

except KeyError :

```
    print("KeyError caught:")
```

Python, the finally block is part of the try-except-finally construct used for exception handling. It allows you to define a block of code that will always execute, regardless of whether an exception occurs or not. This is

particularly useful for cleanup actions, such as closing a file, releasing resources, or resetting variables.

Finally Block:

```
try:
    file = open("example.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("The file does not exist.")
finally:
    # Ensuring the file is closed
    file.close()
    print("File closed.")
```

File Handling in Python

What is a File?

A **file** is a way to store data permanently on your computer or device. It's a collection of data saved on a storage medium (like a hard drive) and can be read or written using a program—like Python.

Think of a file as a digital version of a notebook where you can write and later read information.

Advantages of File Handling in Python

- **Versatility** : File handling in Python allows us to perform a wide range of operations, such as creating, reading, writing, appending, renaming and deleting files.
- **Flexibility** : File handling in Python is highly flexible, as it allows us to work with different file types (e.g. text files, binary files, CSV files , etc.) and to perform different operations on files (e.g. read, write, append, etc.).
- **User – friendly** : Python provides a user-friendly interface for file handling, making it easy to create, read and manipulate files.
- **Cross-platform** : Python file-handling functions work across different platforms (e.g. Windows, Mac, Linux), allowing for seamless integration and compatibility.

Disadvantages of File Handling in Python

- **Security risks** : File handling in Python can also pose security risks, especially if the program accepts user input that can be used to access or modify sensitive files on the system.
- **Complexity :** File handling in Python can be complex, especially when working with more advanced file formats or operations. Careful attention must be paid to the code to ensure that files are handled properly and securely.

- **Performance :** File handling operations in Python can be slower than other programming languages, especially when dealing with large files or performing complex operations.

🗁 Types of Files

Files come in many different types based on how the data is stored or used. Here are the most common types:

1. Text Files

- Contains **human-readable** characters.
- Common extensions: .txt, .csv, .log, .json, .xml
- Example:txt

Hello, this is a simple text file.

2. Binary Files

- Contains data in **machine-readable** format (not readable as plain text).
- Common extensions: .exe, .jpg, .png, .mp3, .pdf, .docx
- Example: Image files, videos, audio, compiled programs.

---

key function for working with files in Python is

open() function.

The open() function takes two parameters; *filename*, and *mode*.

There are three different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

Here is a list of some commonly used file-handling functions in Python.

Function	Description
open()	Used to open a file.
readline()	Used to read a single line from the file.
readlines()	Used to read all the content in the form of lines.
read()	Used to read whole content at a time.
write()	Used to write a string to the file.
writelines()	Used to write multiple strings at a time.
close()	Used to close the file from the program.

Basics of File Handling in Python

Opening and Closing Files

The first step in file handling is opening a file.
Python's open() function is used for this purpose. It requires two arguments: **the file path and the mode in which the file should be opened.**

file = open('example.txt', 'r') # 'r' stands for read mode

Modes include:

- 'r': Read (default)
- 'w': Write (creates a new file if it doesn't exist or truncates the file if it exists)
- 'a': Append (creates a new file if it doesn't exist)
- 'b': Binary mode (e.g., 'rb' or 'wb')

What is file.close()?

The close() method is used to close an open file.

```
file = open("example.txt", "r")
# do something with the file
file.close()  # <- this is important
```

Reading from Files:

Reading the Entire File

The read() method reads the entire file content as a string.

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

Reading Line by Line

The readline() method reads one line at a time, which is useful for large files.

```
file = open('example.txt', 'r')
line = file.readline()
while line:
  print(line)
  line = file.readline()
file.close()
```

Reading All Lines

The readlines() method reads all lines into a list.

```
file = open('example.txt', 'r')
lines = file.readlines()
for line in lines:
  print(line)
file.close()
```

Read specific line in file:

```
with open('d:\jk.txt', 'r') as file:
lines = file.readlines()  # Read all lines into a list
s = lines[5]  # Access the 3rd line (index 2)
print(s)
```

Read specific word in file:

```
with open('d:\jk.txt', 'r') as file:
content = file.read()  # Read the entire file content
words = content.split()  # Split content into words (default splits by whitespace)
specific_word = words[4]  # Access the 5th word (index 4)
print(specific_word)
```

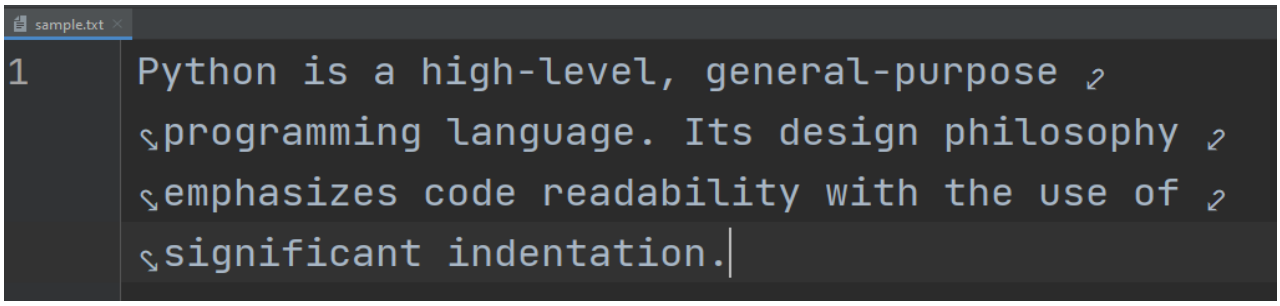
seek()

The seek() function in Python is used to **move the file cursor to the specified location**. When we read a file, the cursor starts at the beginning, but we can move it to a specific position by passing an arbitrary integer (based on the length of the content in the file) to the seek() function.

Visual representation of the seek function

Sample File

We'll be using the sample.txt file, which contains the information shown in the image below.

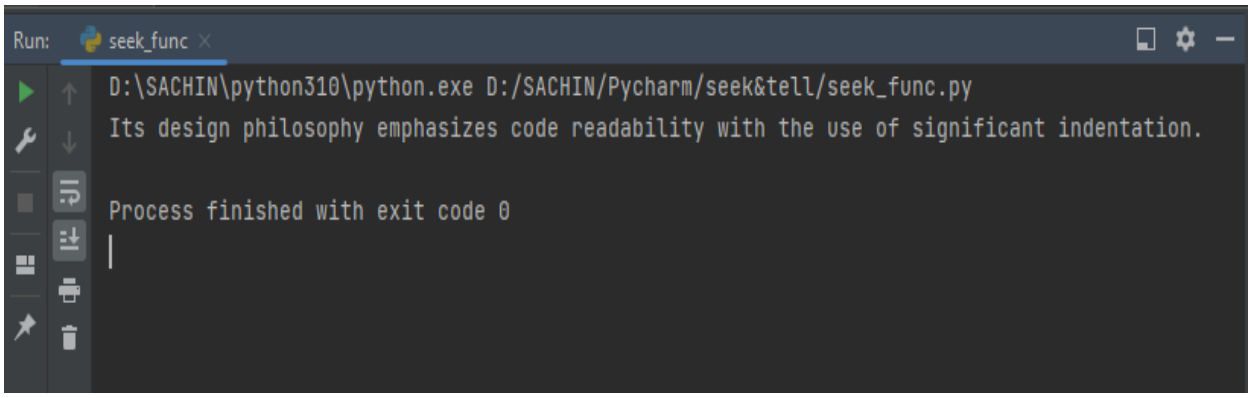


seek

```
# Opening a file for reading
with open('sample.txt', 'r') as file:
    # Setting the cursor at 62nd position
    file.seek(62)
    # Reading the content after the 62nd character
    data = file.read()
    print(data)
```

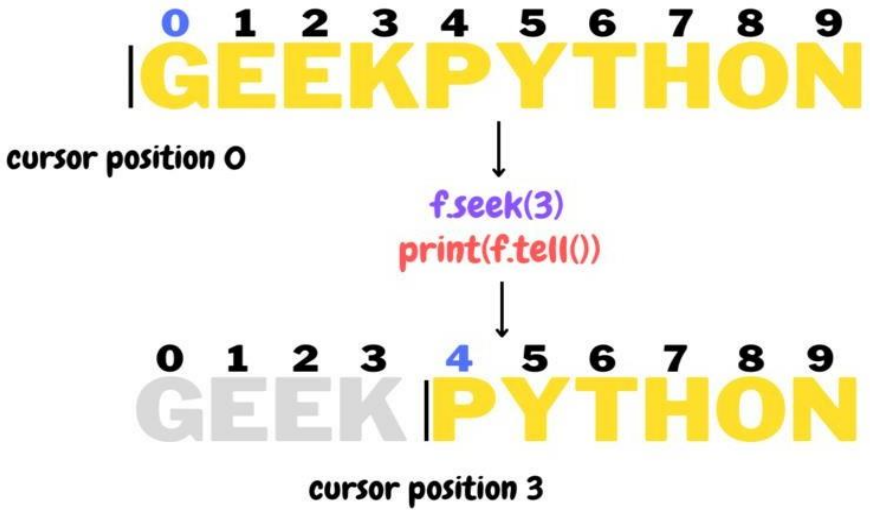
We've moved the cursor to the 62nd position, which means that if we read the file, we'll begin reading after the 62nd character.

Output:



tell

The seek() function is used to set the position of the file cursor, whereas the tell() function **returns the position where the cursor is set to begin reading**.



Visual presentation of the tell function

```
# Opening a file
with open('sample.txt', 'r') as file:
    # Using tell() function
    pos = file.tell()
    # Printing the position of the cursor
    print(f'File cursor position: {pos}')
```

```
    # Printing the content
    data = file.read()
    print(data)
```

Output

```
File cursor position: 0
Python is a high-level, general-purpose programming language. Its
design philosophy emphasizes code readability with the use of
significant indentation.
```

Example 1 – Setting cursor position from the end and printing the cursor position

```
with open('sample.txt', 'r') as file:
    # Setting cursor at the 25th pos from the end
    file.seek(-25, 2)
    # Using tell() function
    pos = file.tell()
    # Printing the position of the cursor
    print(f'File cursor position: {pos}')
    # Printing the content
    data = file.read()
    print(data)

```

Output

*1**File cursor position: 127***

*2**b' significant indentation**.*

The character at the 25th position from the end begins after the 127th character from the beginning, that's why we got the cursor position as 127.

How Do I Create a New text File in Python?

Here's a simple example using the 'w' method instead:

Example:

```
file = open('myfile.txt',)
file.close()
```

 # Output:

Creates a new file named 'myfile.txt' in the current directory

For appending data, use the 'a' mode.

```
file = open('myfile.txt', 'a')
file.write('\nAppending a new line.')
file.close()
```

writelines()

writelines() is used to write a list (or any iterable) of strings to a file.

📌 Important: It does not add newline characters automatically — you must include \n yourself if you want line breaks.

Example: Using writelines()

```
lines = [
    "Line 1\n",
```

```
    "Line 2\n",
    "Line 3\n"
]
with open("example.txt", "w") as file:
    file.writelines(lines)
```

This will write all three lines to the file as separate line

Using the with Statement

Python's with statement simplifies file handling by ensuring that files are automatically closed after the block of code is executed, even in case of exceptions.

```
with open('example.txt', 'r') as file:
  content = file.read()
  print(content)
```

This approach is preferred as it ensures that the file is properly closed, even if an exception occurs.

in Python, the finally block is part of the try-except-finally construct used for exception handling. It allows you to define a block of code that will always execute, regardless of whether an exception occurs or not.

This is particularly useful for cleanup actions, such as closing a file, releasing resources, or resetting variables**.**

Finally Block:

```
try:
    file = open("example.txt", "r")
    content = file.read()
    print(content)
```

except FileNotFoundError:

```
    print("The file does not exist.")
```

finally:

```
    # Ensuring the file is closed
    file.close()
    print("File closed.")
```

****Checking if File Exists**

```
import os
if os.path.exists("example.txt"):
    print("File exists!")
else:
    print("File not found.")
```

Deleting a File

```
import os
os.remove("example.txt")
```

Renaming or Moving a File

```
import os
os.rename("old.txt", "new.txt")
```

Copying a file in Python

is easy — you have a few ways to do it depending on what you're copying (text, binary, large files, etc.).

✅ 1. Using shutil module (Best & Easiest way)

```
import shutil
shutil.copy("source.txt", "destination.txt")
```

📌 This copies the **content and permissions**, but not metadata like creation time.

Manual Copy (for text files)

```
with open("source.txt", "r") as src:
    with open("destination.txt", "w") as dst:
        dst.write(src.read())
```

✅ 3. Manual Copy (for binary files like images)

```
python
CopyEdit
with open("image.jpg", "rb") as src:
    with open("copy.jpg", "wb") as dst:
        dst.write(src.read())
```

Capturing Program Output

If your program generates output (e.g., from a loop), you can store it like this:

```
with open('output.txt', 'w') as file:
    for i in range(5):
```

```
    result = f"Iteration {i}"
    file.write(result + '\n')
    # Or use: print(result, file=file)
```

Write python program to the number of letters and digits in given input sring into a file project.

```
file1 = open('myfile.txt', 'w')
L = ["a,", "b,", "c\n"]
L1 = ["1,", "2,", "3"]

# Writing a string to file
file1.writelines(L)

# Writing multiple strings
# at a time

file1.writelines(L1)
# Closing file

file1.close()
# Checking if the data is
# written to file or not

file1 = open('myfile.txt', 'r')
```

```
print(file1.read())
file1.close()
```

There is a file named Input.Txt. Enter some positive numbers into the file named Input.Txt. Read the contents of the file and if it is an odd number write it to ODD.TXT and if the number is even, write it to EVEN.TXT.

```
file1 = open("d:\jk.txt", "r")
# Open EVEN.txt for writing
file2 = open("d:\EVEN.txt", "w")
# Open ODD.txt for writing
file3 = open("d:\ODD.txt", "w")

for line in file1: # Read each number of file
    num = int(line.strip())  # Remove leading/trailing whitespace
    if num % 2 == 0:
        file2.write(str(num) + "\n")
    else:
        file3.write(str(num) + "\n")

file2 = open("d:\EVEN.txt", "r")
print(“Even no. List”)
print(file2.read())
file2.close()

file3 = open("d:\ODD.txt", "r")
```

```
print(“odd no. List”)
print(file3.read())
file3.close()
```

Reading a Binary File

```
with open("example.jpg", "rb") as file:
    content = file.read()
    print(content[:20])  # print first 20 bytes
```

Working with CSV Files

A CSV (Comma-Separated Values) file is a simple text file that stores tabular data, where each line represents a row and values are separated by commas. It's widely used for data exchange between applications because it's lightweight and easy to process.

Uses of CSV Files

- **Data Storage:** Used to store structured data like spreadsheets and databases.
- **Data Transfer:** Common format for importing/exporting data between software applications.
- **Machine Learning & Data Analysis:** Frequently used in Python, R, and Excel for processing large datasets.
- **Web Applications:** Helps in handling bulk data uploads and downloads.

```
import csv
# Writing to a CSV file
with open('example.csv', 'w', newline='') as file:
  writer = csv.writer(file)
  writer.writerow(['Name', 'Age', 'City'])
```

```
    writer.writerow(['Alice', 30, 'New York'])
    writer.writerow(['Bob', 25, 'San Francisco'])
```

 # Reading from a CSV file

```
with open('example.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

JSON Files

JSON (JavaScript Object Notation) is a popular data format for exchanging data between a server and a web application.
Python's json module allows you to work with JSON data.

Uses of JSON Files

- **Data Storage**: Used in databases and configuration files.
- **Data Exchange**: Common format for APIs to send and receive data.
- **Web Development**: Helps in transferring data between a server and a client.
- **Machine Learning & Data Processing**: Used for handling structured datasets

import json

 # Writing to a JSON file

```
data = {'name': 'Alice', 'age': 30, 'city': 'New York'}
with open('example.json', 'w') as file:
    json.dump(data, file)
```

 # Reading from a JSON file

```
with open('example.json', 'r') as file:
    data = json.load(file)
    print(data)
```

March 2025

UNIT-III	Data Structures in Python	8hours
Operations and Methods, Slicing and Indexing, Creating and Accessing Dictionaries, Dictionary Methods, Sets, Creating and Manipulating Sets, Set Operations		

Indexing and Slicing in Python

Indexing and slicing are used to access elements of sequences like strings, lists, and tuples

1. Indexing

- Indexing is used to access a specific element from a sequence.
- Python uses **zero-based indexing**, meaning the first element is at index 0.

Example of Indexing

```
my_list = [10, 20, 30, 40, 50]

print(my_list[0])   # First element -> 10
print(my_list[2])   # Third element -> 30
print(my_list[-1])  # Last element -> 50
print(my_list[-3])  # Third-last element -> 30
```

✅**Positive Indexing**: Starts from 0 (left to right).
✅**Negative Indexing**: Starts from -1 (right to left).

---

2. Slicing

- Slicing is used to extract a subset of elements from a sequence.
- The syntax is:

sequence[start:stop:step]

- start → Starting index (inclusive).
- stop → Ending index (exclusive).

Examples of Slicing

```
my_list = [10, 20, 30, 40, 50, 60, 70]

print(my_list[1:5])     # [20, 30, 40, 50] (index 1 to 4)
print(my_list[:4])      # [10, 20, 30, 40] (from index 0 to 3)
print(my_list[3:])      # [40, 50, 60, 70] (from index 3 to end)
print(my_list[::2])     # [10, 30, 50, 70] (every 2nd element)
print(my_list[::-1])    # [70, 60, 50, 40, 30, 20, 10] (reverse list)
```

---

3. Slicing Strings

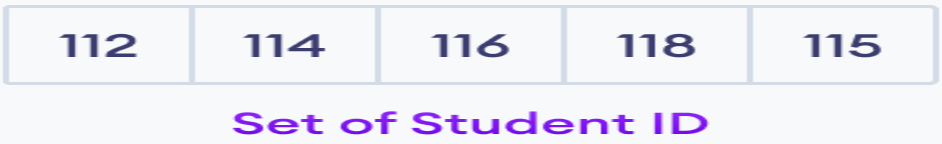
- Works the same way as with lists.

```
text = "Hello, World!"

print(text[0:5])    # 'Hello'
print(text[:5])     # 'Hello'
print(text[7:])     # 'World!'
print(text[::-1])   # '!dlroW ,olleH' (reverse string)
```

Python Sets

A set is a collection of unique data, meaning that elements within a set cannot be duplicated.

For instance, if we need to store information about student IDs, a set is suitable since student IDs cannot have duplicates.



Create a Set in Python In Python, we create sets by placing all the elements inside curly braces {}, separated by commas.

A set can have any number of items and they may be of different types (integer, float, tuple, string, etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.Let's see an example,

```
# create a set of integer type
student_id = {112, 114, 116, 118, 115}
print('Student ID:', student_id)

# create a set of string type
vowel_letters = {'a', 'e', 'i', 'o', 'u'}
print('Vowel Letters:', vowel_letters)

# create a set of mixed data types
mixed_set = {'Hello', 101, -2, 'Bye'}
print('Set of mixed data types:', mixed_set)
```

Output

```
Student ID: {112, 114, 115, 116, 118}
Vowel Letters: {'u', 'a', 'e', 'i', 'o'}
Set of mixed data types: {'Hello', 'Bye', 101, -2}
```

In the above example, we have created different types of sets by placing all the elements inside the curly braces {}.

> **Note:** When you run this code, you might get output in a different order. This is because the set has no particular order.

Create an Empty Set in Python

:Creating an empty set is a bit tricky. Empty curly braces {} will make an empty dictionary in Python.To make a set without any elements, we use the set() function without any argument. For example,

```
# create an empty set
empty_set = set()

# check data type of empty_set
print('Data type of empty_set:', type(empty_set))
```

Output

```
Data type of empty_set: <class 'set'>
```

Duplicate Items in a Set

Let's see what will happen if we try to include duplicate items in a set.

```
numbers = {2, 4, 6, 6, 2, 8}
print(numbers)   # {8, 2, 4, 6}
```

Here, we can see there are no duplicate items in the set as a set cannot contain duplicates.

Add and Update Set Items in Python

Sets are mutable. However, since they are unordered, indexing has no meaning.We cannot access or change an element of a set using indexing or slicing. The set data type does not support it.

Add Items to a Set in Python

In Python, we use the add() method to add an item to a set. For example,

```
numbers = {21, 34, 54, 12}
print('Initial Set:',numbers)
# using add() method
numbers.add(32)
print('Updated Set:', numbers)
```

Output

```
Initial Set: {34, 12, 21, 54}
Updated Set: {32, 34, 12, 21, 54}
```

In the above example, we have created a set named numbers. Notice the line,

```
numbers.add(32)
```

Here, add() adds **32** to our set.

Update Python Set

The update() method is used to update the set with items other collection types (lists, tuples, sets, etc).

For example,

```
companies = {'Lacoste', 'Ralph Lauren'}
tech_companies = ['apple', 'google', 'apple']
# using update() method
companies.update(tech_companies)
print(companies)

# Output: {'google', 'apple', 'Lacoste', 'Ralph Lauren'}
```

Here, all the unique elements of tech_companies are added to the companies set.

Remove an Element from a Set We use the discard() method to remove the specified element from a set. For example,

```
languages = {'Swift', 'Java', 'Python'}
print('Initial Set:',languages)
# remove 'Java' from a set
removedValue = languages.discard('Java')
print('Set after remove():', languages)
```

Output

```
Initial Set: {'Python', 'Swift', 'Java'}
Set after remove(): {'Python', 'Swift'}
```

Here, we have used the discard() method to remove 'Java' from the languages set.

Iterate Over a Set in Python

```
fruits = {"Apple", "Peach", "Mango"}

# for loop to access each fruits
for fruit in fruits:
    print(fruit)
```

Output

```
Mango
Peach
Apple
```

Here, we have used for loop to iterate over a set in Python.

Find Number of Set Elements

We can use the len() method to find the number of elements present in a Set. For example,

```
even_numbers = {2,4,6,8}
print('Set:',even_numbers)

# find number of elements
print('Total Elements:', len(even_numbers))
```

Output

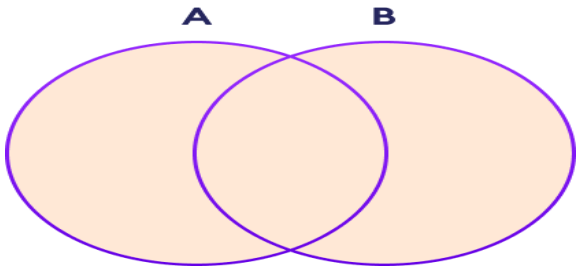
```
Set: {8, 2, 4, 6}
Total Elements: 4
```

Python Set Operations

Python Set provides different built-in methods to perform mathematical set operations like union, intersection, subtraction, and symmetric difference.

Union of Two Sets

The union of two sets A and B includes all the elements of sets A and B.



Set Union in Python

. For example,

```
# first set
A = {1, 3, 5}

# second set
B = {0, 2, 4}
# perform union operation using |
print('Union using |:', A | B)
# perform union operation using union()
print('Union using union():', A.union(B))
```

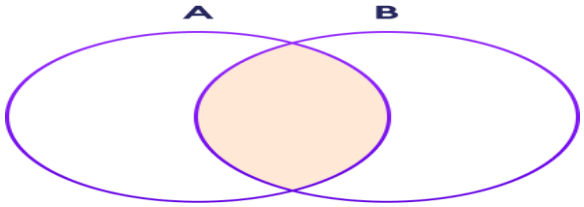
Output

```
Union using |: {0, 1, 2, 3, 4, 5}
Union using union(): {0, 1, 2, 3, 4, 5}
```

Note: A|B and union() is equivalent to A ⋃ B set operation.

Set Intersection

The intersection of two sets A and B include the common elements between set A and B.



Set Intersection in Python

In Python, we use the & operator or the intersection() method to perform the set intersection operation. For example,

```
# first set
A = {1, 3, 5}

# second set
B = {1, 2, 3}

# perform intersection operation using &
print('Intersection using &:', A & B)

# perform intersection operation using intersection()
print('Intersection using intersection():', A.intersection(B))
```

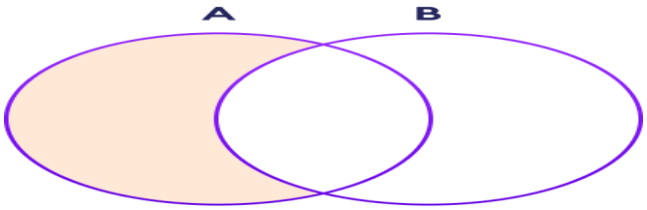
Output

```
Intersection using &: {1, 3}
Intersection using intersection(): {1, 3}
```

Note: A&B and intersection() is equivalent to A ⋂ B set operation.

Difference between Two Sets

The difference between two sets A and B include elements of set A that are not present on set B.



Set Difference in Python

We use the - operator or the difference() method to perform the difference between two sets. For example,

```
# first set
A = {2, 3, 5}

# second set
B = {1, 2, 6}

# perform difference operation using &
print('Difference using &:', A - B)

# perform difference operation using difference()
print('Difference using difference():', A.difference(B))
```

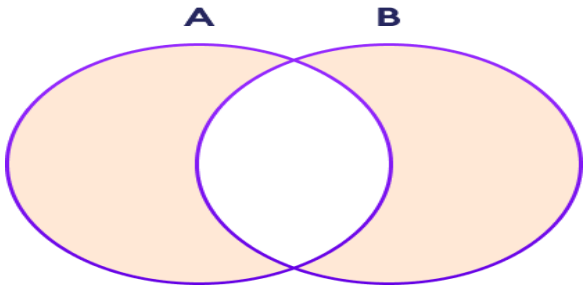
Output

```
Difference using &: {3, 5}
Difference using difference(): {3, 5}
```

Note: A - B and A.difference(B) is equivalent to A - B set operation.

Set Symmetric Difference

The symmetric difference between two sets A and B includes all elements of A and B without the common elements.



Set Symmetric Difference in Python

In Python, we use the ^ operator or the symmetric_difference() method to perform symmetric differences between two sets. For example,

```
# first set
A = {2, 3, 5}

# second set
B = {1, 2, 6}

# perform difference operation using &
print('using ^:', A ^ B)

# using symmetric_difference()
print('using symmetric_difference():', A.symmetric_difference(B))
```

Output

```
using ^: {1, 3, 5, 6}
using symmetric_difference(): {1, 3, 5, 6}
```

Check if two sets are equal

We can use the == operator to check whether two sets are equal or not. For example,

```
# first set
A = {1, 3, 5}

# second set
B = {3, 5, 1}

# perform difference operation using &
if A == B:
    print('Set A and Set B are equal')
else:
    print('Set A and Set B are not equal')
```

Output

```
Set A and Set B are equal
```

In the above example, A and B have the same elements, so the condition

```
if A == B
```

evaluates to True. Hence, the statement print('Set A and Set B are equal') inside the if is executed.

UNIT-3 Problems

Indexing and Slicing Programs

*Use of Split() with input()

You can use the split() method with input() to process user inputs by splitting the entered string into a list of parts. This is especially useful for handling multiple values entered at once, like when users input numbers or words separated by spaces or other delimiters.

Here's an example:

```
# Taking a space-separated input
user_input = input("Enter some words separated by spaces: ")
words = user_input.split()  # Splits the input by spaces
print(words)  # Output: A list of words
```

Output:

Enter some words separated by spaces: hello hi who are you

['hello', 'hi', 'who', 'are', 'you']

Here's an example input() without Split()

```
# Taking a space-separated input
user_input = input("Enter some words separated by spaces: ")
print(user_input)  # Output: A list of words
```

Output:

Enter some words separated by spaces: hello hi who are you

hello hi who are you

Example 1: Splitting input by default separator (whitespace)

1. **Accessing Elements**: Write a program to access the second and second-to-last elements of a list using indexing.

lst = input("Enter elements separated by space: ").**split()**

print("Second element:", lst[1])

print("Second-to-last element:", lst[-2])

Output:

Enter elements separated by space: 2 3 4 5 6 7 8

Second element: 3

Second-to-last element: 7

2. **Extract a Substring**: Write a program to extract a substring from a string (e.g., characters from index 2 to 6).

string = input("Enter a string: ")

substring = string[2:7]

print("Substring:", substring)

Output:

Enter a string: kitkanpur

Substring: tkanp

3. **Skipping Elements**: Create a program that prints every alternate character in a string.

```
string = input("Enter a string: ")
alternate_chars = string[::3]
print("Alternate characters:", alternate_chars)
```

Output:

Enter a string: KITKANPUR

Alternate characters: KTAPR

4. **Extract List Slice**: Write a program to extract a slice from a list.

```
lst = input("Enter elements separated by space: ").split()
start = int(input("Enter start index: "))
end = int(input("Enter end index: "))
slice_of_list = lst[start:end]
print("Sliced list:", slice_of_list)
```

Output:

Enter elements separated by space: 2 5 6 8 9 44 55 66 43

Enter start index: 3

Enter end index: 6

Sliced list: ['8', '9', '44']

5. **Negative Indexing**: Write a program to access the last 3 elements of a list using negative indexing.

```
lst = input("Enter elements separated by space: ").split()
print("Last three elements:", lst[-3:])
```

Output:

Enter elements separated by space: 2 3 4 5 8 9

Last three elements: ['5', '8', '9']

6. **Palindrome Checker**: Write a program to check if a string is a palindrome using slicing.

```
string = input("Enter a string: ")
if string == string[::-1]:
    print("It's a palindrome!")
else:
    print("Not a palindrome.")
```

Output:

Enter a string: MADAM

It's a palindrome!

Enter a string: HELLO

Not a palindrome.

7. **Using Slicing**: Write a program to create a shallow of a list using slicing.

```
lst = input("Enter elements separated by space: ").split()
copied_lst = lst[:]
print("Copied list:", copied_lst)
```

Output:

Enter elements separated by space: HELLO HI

Copied list: ['HELLO', 'HI']

Here are programs to help you explore and practice the use of sets:

Add a single element:

Use the add() method to add a single element to the set.

```
my_set = {1, 2, 3}
my_set.add(4) # Adding a single element
print(my_set)
```

Output: {1, 2, 3, 4}

Add multiple elements:

Use the update() method to add multiple elements (from a list, tuple, or another set).

```
my_set = {1, 2, 3}
my_set.update([4, 5, 6]) # Adding multiple elements
print(my_set)

# Output: {1, 2, 3, 4, 5, 6}
```

The update() method can also accept other iterables, such as strings or another set:

```
my_set.update({7, 8}, "90")
print(my_set)

# Output: {1, 2, 3, 4, 5, 6, '9', '0', 7, 8}
```

Remove and then add new elements:

If you want to update by replacing old elements, you can remove elements using remove() or discard(), and then add the new ones.

```
my_set = {1, 2, 3}
my_set.discard(2) # Removes 2
my_set.add(4)     # Adds 4
print(my_set)
# Output: {1, 3, 4}
```

Taking input one element at a time in a loop in SET

```
n = int(input("How many elements do you want to add to the set? "))
user_set = set()
for  i in range(n):
    element = input("Enter an element: ")
    user_set.add(element)
print("Set:", user_set)
```

Output:

How many elements do you want to add to the set? 3

Enter an element: 44

Enter an element: 55

Enter an element: 66

Set: {'66', '55', '44'}

Check Element in a Set

```
set_a = {1, 2, 3, 4, 5}
element = int(input("Enter a number: "))
print(f"Is {element} in the set?", element in set_a)
```

Output:

Enter a number: 4

Is 4 in the set? True

Find Max and Min

```
set_a = {10, 20, 5, 15}
print("Max element:", max(set_a))
print("Min element:", min(set_a))
```

Remove Duplicates from a List Using Set

```
numbers = [1, 2, 3, 4, 1, 2, 5]
unique_numbers = set(numbers)
print("List without duplicates:", unique_numbers)
```

To sort elements in a set in , follow these steps:

1. Convert the set into a list: Sets are unordered collections, so to sort their elements, you must convert them into a list.
2. Sort the list: Use the sorted() function, which creates a sorted list while leaving the original set unchanged.

here's an example Sorting in Ascending Order

```
my_set = {5, 2, 9, 1}
# Convert the set to a sorted list
sorted_list = sorted(my_set)
print("Sorted elements:", sorted_list)  # Output: [1, 2, 5, 9]
```

Sorting in Descending Order

To sort in descending order, include reverse=True in the sorted() function:

```
sorted_desc = sorted(my_set, reverse=True)
print("Descending order:", sorted_desc)  # Output: [9, 5, 2, 1]
```

Set Comprehension

```
squared_set = {x**2 for x in range(1, 6)}
print("Squared values set:", squared_set)
```

Output:

Squared values set: {1, 4, 9, 16, 25}

Find Common Elements Between Two Lists

```
list_a = [1, 2, 3, 4]
list_b = [3, 4, 5, 6]
common_elements = set(list_a) & set(list_b)
print("Common elements:", common_elements)
```

{3,4}

Check for Equality

You can use the equality operator == to check if two sets have the same elements:

```
set1 = {1, 2, 3}
set2 = {3, 2, 1}
print(set1 == set2)
```

 # Output: True

2. Subset and Superset Checks

To determine if one set is a subset or superset of another, use .issubset() and .issuperset():

```
set1 = {1, 2, 3}

set2 = {4, 5, 6}

print(set1.issubset(set2))  # True if set1 is contained in set2

print(set2.issuperset(set1))  # True if set2 contains set1

# Output:

False

False
```

3. Find Differences

To find elements present in one set but not in another:

```
set1 = {1, 2, 3}

set2 = {2, 5, 6}

print(set1 - set2)  # Elements in set1 but not in set2

print(set2 - set1)  # Elements in set2 but not in set1

# Output:

{1, 3}

{5, 6}
```

1. **Basic Operations on Sets**

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}
print("Union:", set_a.union(set_b))
print("Intersection:", set_a.intersection(set_b))
print("Difference (A - B):", set_a.difference(set_b))
print("Symmetric Difference:", set_a.symmetric_difference(set_b))
```

Output:

Union: {1, 2, 3, 4, 5}

Intersection: {3}

Difference (A - B): {1, 2}

Symmetric Difference: {1, 2, 4, 5}

Set Operations with User Input

```
n = int(input("How many elements do you want to add to the set1? "))
set_a = set()
for i in range(n):
    element = input("Enter an element: ")
    set_a.add(element)
n = int(input("How many elements do you want to add to the set2? "))
set_b = set()
for j in range(n):
    element = input("Enter an element: ")
    set_b.add(element)
```

```
print("Union:", set_a.union(set_b))
print("Intersection:", set_a.intersection(set_b))
print("Difference (A - B):", set_a.difference(set_b))
print("Symmetric Difference:", set_a.symmetric_difference(set_b))
```

Output:

How many elements do you want to add to the set1? 3

Enter an element: 9

Enter an element: 8

Enter an element: 7

How many elements do you want to add to the set2? 3

Enter an element: 6

Enter an element: 5

Enter an element: 7

Union: {'8', '6', '9', '7', '5'}

Intersection: {'7'}

Difference (A - B): {'8', '9'}

Symmetric Difference: {'6', '9', '8', '5'}

OR

```
n = int(input("How many elements do you want to add to the set1? "))
set_a = set()
for i in range(n):
    element = input("Enter an element: ")
    set_a.add(element)
n = int(input("How many elements do you want to add to the set2? "))
set_b = set()
for j in range(n):
    element = input("Enter an element: ")
    set_b.add(element)
print("Union:", set_a | set_b)
print("Intersection:", set_a & set_b)
print("Difference (A - B):", set_a - set_b)
print("Symmetric Difference:", set_a ^ set_b)
```

Output:

```
How many elements do you want to add to the set1? 3
Enter an element: 9
Enter an element: 8
Enter an element: 7
How many elements do you want to add to the set2? 3
Enter an element: 6
Enter an element: 5
Enter an element: 7
```

Union: {'8', '6', '9', '7', '5'}

Intersection: {'7'}

Difference (A - B): {'8', '9'}

Symmetric Difference: {'6', '9', '8', '5'}

10. Frozenset Example

A frozenset in is an immutable version of a set. While regular sets (set) can be modified by adding or removing elements, a frozenset cannot be changed once created.

Here are key features of frozenset:

- Immutable: You cannot add or remove elements from a frozenset.
- Supports set operations: Just like a normal set, you can perform operations like union, intersection, difference, and symmetric difference.

Example:

frozen = **frozenset**([1, 2, 3])

print("Frozen Set:", frozen)

Output:

Frozen Set: frozenset({1, 2, 3})

Example:

Frozen = frozenset([1, 2, 3])

frozen.add(4)

print("Frozen Set:", frozen)

Output:

Traceback (most recent call last):

File "C:/Users/Slim-5/AppData/Local/Programs//313/x.py", line 3, in <module>

frozen.add(4)

AttributeError: 'frozenset' object has no attribute 'add'

Example:

frozen = **frozenset**([1, 2, 3])

print("Frozen Set:", frozen)

Try adding an element (will throw an error because frozenset is immutable)

try:

frozen.add(4)

except AttributeError:

print("Cannot add element to frozenset")

Output:

Frozen Set: frozenset({1, 2, 3})

Cannot add element to frozenset

HOW TO USER INPUT FOR DICTIONARY

To take user input for a dictionary in , you can use a loop and ask the user to provide key-value pairs. Here's an example:

```
# Initialize an empty dictionary
my_dict = {}
# Ask for the number of key-value pairs
n = int(input("Enter the number of key-value pairs: "))
# Loop to take input
for  i  in range(n):
    key = input("Enter the key: ")
    value = input("Enter the value: ")
    my_dict[key] = value  # Add the key-value pair to the dictionary

# Display the dictionary
print("Your dictionary:", my_dict)
```

Output:

Enter the number of key-value pairs: 3

Enter the key: a

Enter the value: 1

Enter the key: b

Enter the value: 2

Enter the key: c

Enter the value: 3

Your dictionary: {'a': '1', 'b': '2', 'c': '3'}

Creating and Accessing a Dictionary

```
my_dict = {"name": "Kamal", "age": 25, "city": "Kanpur"}
print("Name:", my_dict["name"])
print("City:", my_dict.get("city"))  # Using get() method
```

Output:

```
Name: Kamal
City: Kanpur
```

2. Adding and Updating Key-Value Pairs

```
my_dict = {"name": "Kamal"}
my_dict["age"] = 25  # Adding a key-value pair
my_dict["name"] = "Raj"  # Updating an existing key
print(my_dict)
```

Output:

```
{'name': 'Raj', 'age': 25}
```

3. Removing Key-Value Pairs

```
my_dict = {"name": "Kamal", "age": 25, "city": "Kanpur"}
removed = my_dict.pop("age")  # Removing a specific key
print("Removed:", removed)
print("Dictionary after removal:", my_dict)
```

Output:

```
Removed: 25
Dictionary after removal: {'name': 'Kamal', 'city': 'Kanpur'}
```

4. Using keys(), values(), and items()

```
my_dict = {"name": "Kamal", "age": 25, "city": "Kanpur"}
print("Keys:", list(my_dict.keys()))
print("Values:", list(my_dict.values()))
print("Items:", list(my_dict.items()))
```

Output:

Keys: ['name', 'age', 'city']

Values: ['Kamal', 25, 'Kanpur']

Items: [('name', 'Kamal'), ('age', 25), ('city', 'Kanpur')]

5. Iterating Over a Dictionary

```
my_dict = {"name": "Kamal", "age": 25, "city": "Kanpur"}
for key, value in my_dict.items():
    print(f"{key}: {value}")
```

Output:

name: Kamal

age: 25

city: Kanpur

6. Checking for Key Existence

```
my_dict = {"name": "Kamal", "age": 25}
if "city" in my_dict:
    print("Key 'city' exists.")
else:
    print("Key 'city' does not exist.")
```

Output:

Key 'city' does not exist.

7. Merging Dictionaries

```
dict1 = {"name": "Kamal"}
dict2 = {"age": 25, "city": "Kanpur"}
dict1.update(dict2)  # Merging dict2 into dict1
print(dict1)
```

Output:

{'name': 'Kamal', 'age': 25, 'city': 'Kanpur'}

8. Copying a Dictionary

```
my_dict = {"name": "Kamal", "age": 25}
copy_dict = my_dict.copy()  # Creating a copy
print("Original:", my_dict)
print("Copy:", copy_dict)
```

Output:

Original: {'name': 'Kamal', 'age': 25}

Copy: {'name': 'Kamal', 'age': 25}

9. Clearing a Dictionary

```
my_dict = {"name": "Kamal", "age": 25}
my_dict.clear()  # Removing all elements
print("Cleared Dictionary:", my_dict)
```

Output:

Cleared Dictionary: {}

10. Dictionary Comprehension

```
squares = {x: x**2 for x in range(1, 6)}
print(squares)
```

Output:

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

Dictionary Equality Comparision

Check for Equality To check if two dictionaries are exactly the same (keys and values), you can use the equality operator ==:

```
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 2, "a": 1}
print(dict1 == dict2)
```

 # Output: True

find differences in keys or values between two dictionaries:

```
dict1 = {"a": 1, "b": 2, "c": 3}
dict2 = {"a": 1, "b": 5, "d": 4}
# Keys unique to dict1
print(dict1.keys() - dict2.keys())
# Output: {'c'}
# Keys unique to dict2
print(dict2.keys() - dict1.keys())
# Output: {'d'}
```

11. Concatenate following dictionaries to create a new one

Sample Solution:

```
dic1 = {1: 10, 2: 20}
dic2 = {3: 30, 4: 40}
dic3 = {5: 50, 6: 60}
# Create an empty dictionary 'dic4' that will store the combined key-value pairs from 'dic1', 'dic2', and 'dic3'.
dic4 = {}
# Iterate through each dictionary ('dic1', 'dic2', and 'dic3') using a loop.
for d in (dic1, dic2, dic3):
    dic4.update(d)
print(dic4)
```

Sample Output:

```
{1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}
```

13. Filter Dictionary Based on Values

marks = {'Cierra Vega': 175, 'Alden Cantrell': 180, 'Kierra Gentry': 165, 'Pierre Cox': 190}

print("Marks greater than 170:")

result = {key: value for (key, value) in marks.items() if value >= 170}

print(result)

Sample Output:

Marks greater than 170:

{'Cierra Vega': 175, 'Alden Cantrell': 180, 'Pierre Cox': 190}

14.Find Maximum and Minimum Values in a Set

Create a set 'setn' with elements 5, 10, 3, 15, 2, and 20.

setn = {5, 10, 3, 15, 2, 20}

Use the 'max()' function to find and print the maximum value in 'setn'.

print("\nMi value aximum of the said set:")

print(max(setn))

Print a message to indicate finding the minimum value in the set.

print("\nMinimum value of the said set:")

Use the 'min()' function to find and print the minimum value in 'setn'.

print(min(setn))

Sample Output:

Maximum value of the said set:

20

Minimum value of the said set:

2

Here is how to convert two lists into a dictionary in Python.

```
keys = ['a', 'b', 'c']
values = [1, 2, 3]
my_dict = dict(zip(keys, values))
print(my_dict)
```

Output {'a': 1, 'b': 2, 'c': 3}

Accessing Elements in a Nested Dictionary

```
nested_dict = {
    'person1': {'name': 'Kamal', 'age': 25},
    'person2': {'name': 'Ravi', 'age': 30}
}
# Access specific element
print("Name of person1:", nested_dict['person1']['name'])
```

Output:

Name of person1: Kamal

2. Adding Elements to a Nested Dictionary

```
nested_dict = {
    'person1': {'name': 'Kamal', 'age': 25},
    'person2': {'name': 'Ravi', 'age': 30}
}
# Adding a new person
nested_dict['person3'] = {'name': 'Anita', 'age': 28}
print("Updated nested dictionary:", nested_dict)
```

Output:

Updated nested dictionary: {'person1': {'name': 'Kamal', 'age': 25}, 'person2': {'name': 'Ravi', 'age': 30}, 'person3': {'name': 'Anita', 'age': 28}}

2. Iterating Through a Nested Dictionary

```
nested_dict = {
    'person1': {'name': 'Kamal', 'age': 25},
    'person2': {'name': 'Ravi', 'age': 30}
}

for person, details in nested_dict.items():
    print(f"{person}:")
    for key, value in details.items():
        print(f"  {key}: {value}")
```

Output:

person1:

name: Kamal

age: 25

person2:

name: Ravi

age: 30

4. Updating Elements in a Nested Dictionary

```
nested_dict = {
    'person1': {'name': 'Kamal', 'age': 25},
    'person2': {'name': 'Ravi', 'age': 30}
}
# Update age of person1
nested_dict['person1']['age'] = 26
print("After update:", nested_dict)
```

Output:

After update: {'person1': {'name': 'Kamal', 'age': 26}, 'person2': {'name': 'Ravi', 'age': 30}}

5. Deleting an Element from a Nested Dictionary

```
nested_dict = {
    'person1': {'name': 'Kamal', 'age': 25},
    'person2': {'name': 'Ravi', 'age': 30}
}
# Delete person1
del nested_dict['person1']
print("After deletion:", nested_dict)
```

Output:

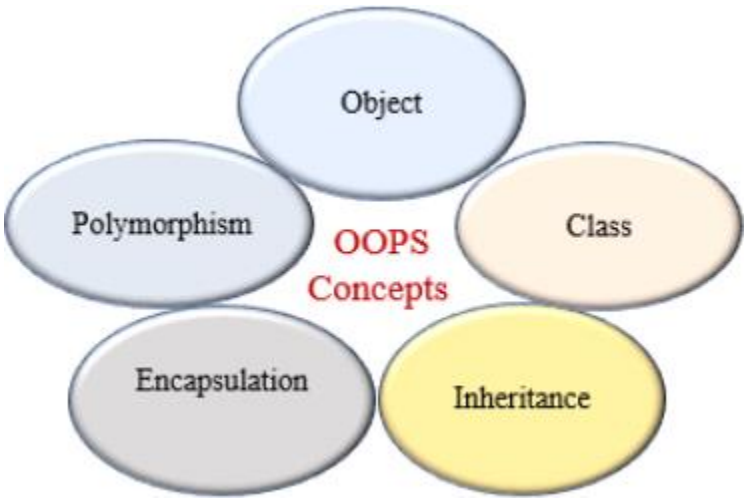
After deletion: {'person2': {'name': 'Ravi', 'age': 30}}

UNIT-IV	Object-Oriented Programming(OOP) in Python	10hours
Introduction to OOP: Classes and Objects, Inheritance and Polymorphism, Advanced OOP Concepts: Encapsulation and Abstraction, Class Methods and Static Methods.		

What is Object Oriented Programming in Python

Object-oriented programming (OOP) is a programming paradigm based on the concept of "**objects**". The object contains both data and code: Data in the form of properties (often known as attributes), and code, in the form of methods (actions object can perform).

An object-oriented paradigm is to design the program using classes and objects. Python programming language supports different programming approaches like functional programming, modular programming. One of the popular approaches is object-oriented programming (OOP) to solve a programming problem is by creating objects



Python OOP concepts

An object has the following two characteristics:

- Attribute
- Behavior

For example, A Car is an object, as it has the following properties:

- **name, price, color as attributes**
- **breaking, acceleration as behavior**

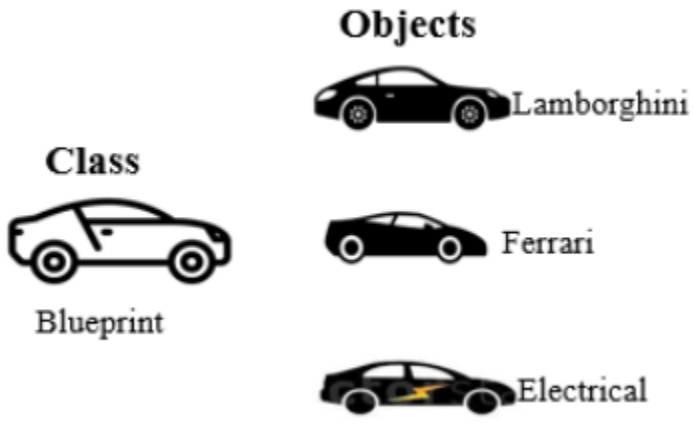
One important aspect of OOP in Python is to create **reusable code** using the concept of inheritance. This concept is also known as DRY (Don't Repeat Yourself).

Class and Objects

In Python, everything is an object. A **class is a blueprint for the object**. To create an object we require a model or plan or blueprint which is nothing but class.

For example, you are creating a vehicle according to the Vehicle blueprint (template). The plan contains all dimensions and structure. Based on these descriptions, we can construct a car, truck, bus, or any vehicle. Here, a car, truck, bus are objects of Vehicle class

A class contains the properties (attribute) and action (behavior) of the object. Properties represent variables, and the methods represent actions. Hence class includes both variables and methods.



Python Class and Objects

Object is an instance of a class. The physical existence of a class is nothing but an object. In other words, the object is an entity that has a state and behavior. It may be any real-world object like the mouse, keyboard, laptop, etc.

Class Attributes and Methods

When we design a class, we use instance variables and class variables.

In Class, attributes can be defined into two parts:

- Instance variables: The instance variables are attributes attached to an instance of a class. We define instance variables in the constructor (the __init__() method of a class).
- Class Variables: A class variable is a variable that is declared inside of class, but outside of any instance method or __init()___method.

Inside a Class, we can define the following three types of methods.

- **Instance method**: Used to access or modify the object attributes. If we use instance variables inside a method, such methods are called instance methods.
- **Class method**: Used to access or modify the class state. In method implementation, if we use only class variables, then such type of methods we should declare as a class method.
- **Static method**: It is a general utility method that performs a task in isolation. Inside this method, we don't use instance or class variable because this static method doesn't have access to the class attributes.

Creating Class and Objects

In Python, Use the keyword class to define a Class. In the class definition, the first string is docstring which, is a brief description of the class.

Syntax

```
class classname:
    '''documentation string'''
   class_suite
```

- **Documentation string**: represent a description of the class. It is optional.
- **class_suite: class suite contains class attributes and methods**

We can create any number of objects of a class. use the following syntax to create an object of a class.

The Python __init__ Method

The __init__ **method** gets invoked as soon as the object is created. It is like the **constructor** of a class. The method **initializes the data members of the class for an object.**

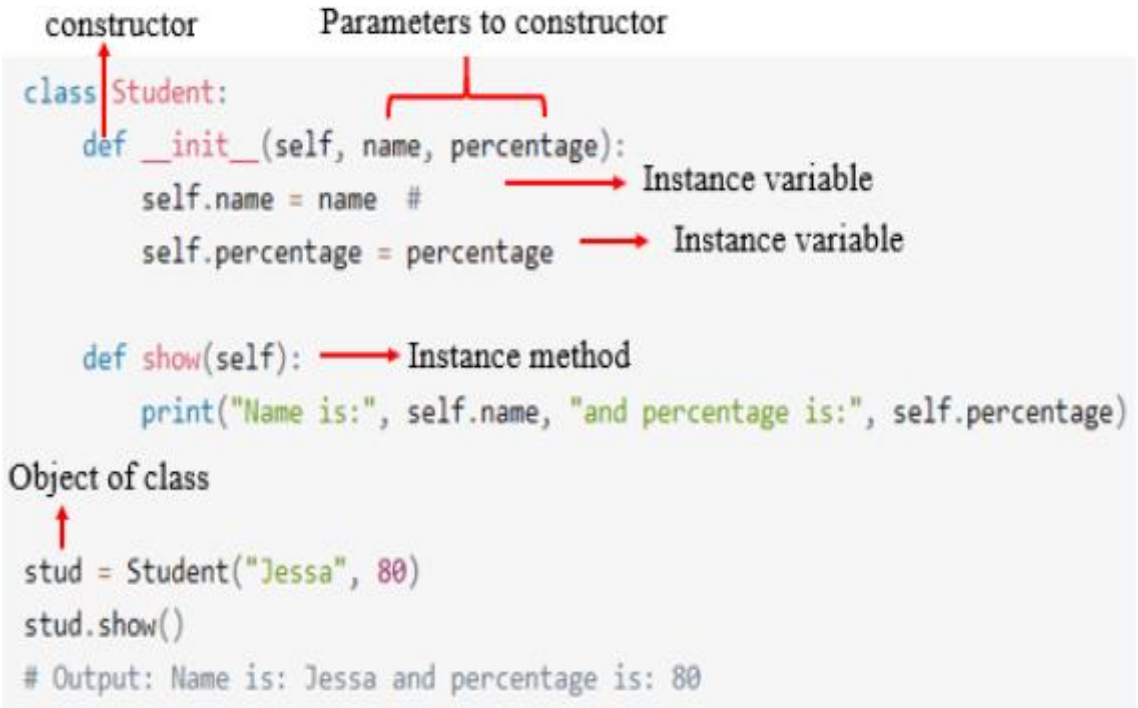
2. The Python self

self is the first parameter in the **method definition of a class**. We do **not give a value for this parameter when we call the method, Python provides it It is a default argument for any method of the class..**

```
reference_variable = classname()
```

OOP Example: Creating Class and Object in Python

```
class Student:
    def __init__(self, name, percentage):
        self.name = name
        self.percentage = percentage
   def show(self):
        print(f"{self.name} is studying for percentge is {self.percentage}.")
# Object
student1 = Student("Alice", 70)
student2 = Student("rakesh", 80)
student1.show()
student2.show()
```



line by line to understand what it's doing:

---

class Student:

- This line defines a **new class** named Student. A class is a blueprint for creating objects.

---

def __init__(self, name, percentage):

- This defines the **constructor** method __init__, which is automatically called when a new Student object is created.
- It takes three parameters: self, name, and percentage.
- self refers to the instance of the class (i.e., the object being created).

---

self.name = name

self.percentage = percentage

- These lines assign the name and percentage parameters to the instance variables self.name and self.percentage.
- This allows each object to store its own name and percentage.

---

def show(self):

- This defines a **method** named show inside the Student class.
- It takes one parameter: self, referring to the object calling the method.

---

print(f"{self.name} is studying for percentge is {self.percentage}.")

- This line prints a message using **f-string formatting** to insert the student's name and percentage.
- Note: There's a small typo in the message — "percentge" should be "percentage".

 # Object

student1 = Student("Alice", 70)

- This line creates an object student1 of the class Student, with "Alice" as the name and 70 as the percentage.

---

student2 = Student("rakesh", 80)

- This line creates another object student2 with name "rakesh" and percentage 80.

---

student1.show()

- This calls the show() method on student1. It will print:
  "Alice is studying for percentge is 70."

---

student2.show()

- This calls the show() method on student2. It will print:
  "rakesh is studying for percentge is 80."

Example:

```
class Employee:
    # class variables
    company_name = 'ABC Company'

    # constructor to initialize the object
    def __init__(self, name, salary):
        # instance variables
        self.name = name
        self.salary = salary

    # instance method
    def show(self):
        print('Employee:', self.name, self.salary, self.company_name)

# create first object
emp1 = Employee("Harry", 12000)
emp1.show()

# create second object
emp2 = Employee("Emma", 10000)
emp2.show()
```

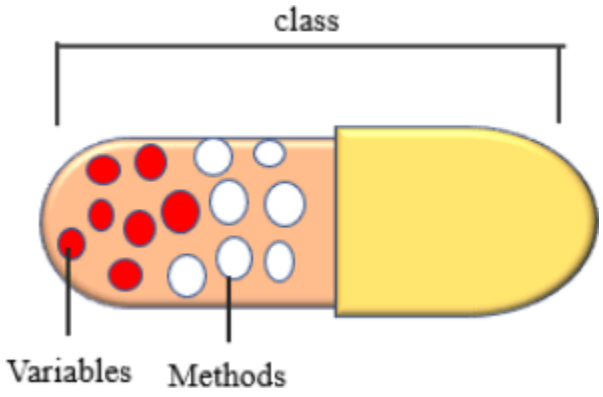
Output:

```
Employee: Harry 12000 ABC Company
Employee: Emma 10000 ABC Company
```

- **In the above example**, we created a Class with the name Employee.
- Next, we defined two attributes name and salary.
- Next, in the __init__() method, we initialized the value of attributes. This method is called as soon as the object is created. The init method initializes the object.
- Finally, from the Employee class, we created two objects, Emma and Harry.
- Using the object, we can access and modify its attributes.

Encapsulation in Python

In Python, encapsulation is a method of wrapping data and functions into a single entity. For example, A class encapsulates all the data (methods and variables**). Encapsulation means** the internal representation of an object is generally **hidden from outside of the object's definition.**



Python Encapsulation

Need of Encapsulation

Encapsulation acts as a protective layer. We can restrict access to methods and variables from outside, and It can prevent the data from being modified by accidental or unauthorized modification. Encapsulation provides security by hiding the data from the outside world.

Example: Encapsulation in Python :**we do not have access modifiers, such as public, private, and protected**. But we can achieve encapsulation by using prefix **double underscore** to control access of variable and method within the Python program.

```
class Employee:
    def __init__(self, name, salary):
        # public member
        self.name = name
        # private member
        # not accessible outside of a class
        self.__salary = salary

    def show(self):
        print("Name is ", self.name, "and salary is", self.__salary)
emp = Employee("Jessa", 40000)
emp.show()

# access salary from outside of a class
print(emp.__salary)
```

Output:

```
Name is  Jessa and salary is 40000

AttributeError: 'Employee' object has no attribute '__salary'
```

In the above example, we create a class called Employee. Within that class, we declare two variables name and __salary. We can observe that the name variable is accessible, but __salary is the **private variable**. We **cannot access it from outside of class**. If we try to access it, **we will get an error**

Sure! Let's break down this Python code line by line:

class Employee:

- This defines a class named Employee. A class is a **blueprint for creating objects (instances).**

def __init__(self, name, salary):

- This is the constructor method of the class. It gets called automatically when a new object is created.
- name and salary are parameters passed during object creation.

 # public member

self.name = name

- This line creates a public instance variable name and assigns it the value passed via the constructor.
- Being public, it can be accessed outside the class (e.g., emp.name).

 # private member

 # not accessible outside of a class

self.__salary = salary

- This line creates a private instance variable __salary (note the double underscore prefix).
- In Python, variables with __ prefix are name-mangled to make them harder to access from outside the class.

def show(self):

print("Name is ", self.name, "and salary is", self.__salary)

- This method show prints the employee's name and salary.
- It can access both name (public) and __salary (private) because it's within the class.

emp = Employee("Jessa", 40000)

- This creates an object emp of class Employee with name="Jessa" and salary=40000.

emp.show()

- This calls the show() method, which prints the name and salary of the employee.

 # access salary from outside of a class

print(emp.__salary)

- This **line raises an error (AttributeError**) because __salary is a private attribute.
- Python's name mangling renames __salary internally to _Employee__salary, so it can't be accessed directly like this.

If you want to access the private variable (not recommended without a method), you'd have to do:

print(emp._Employee__salary)

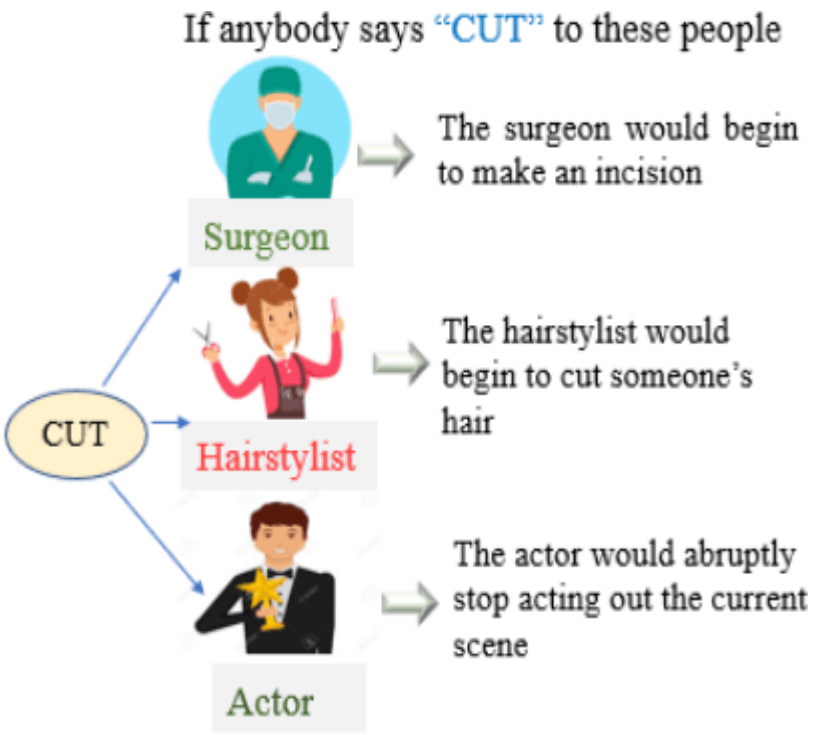
Would you like a version of this with error handling or with getter/setter methods?

Polymorphism in Python

Polymorphism in OOP is the **ability of an object to take many forms**. In simple words, polymorphism allows us to **perform the same action in many different ways.**

Polymorphism is taken from the Greek words Poly (many) and morphism (forms). Polymorphism defines the ability to take different forms.

For example, The **student can act as a student in college**, **act as a player on the ground, and as a daughter/brother in the home**. Another example in the programming language, the **+ operator, acts as a concatenation and arithmetic addition.**



=

Polymorphism

Example:

```
def add(x, y):
    return x + y
print(add(5, 10))         # Integers
print(add("Hello, ", "World!"))  # Strings
print(add([1, 2], [3, 4]))       # Lists
```

Example:

```
def multiply(x, y):
    return x * y
print(multiply(3, 4))           # Integers
print(multiply("Hi", 3))        # String repeat
```

Example:

```
print(len("Hello"))       # String length
print(len([1, 2, 3]))     # List length
print(len({'a': 1}))      # Dict lengthrepeat
```

Example:

```
class Circle:
    def area(self):
        return 3.14 * 5 * 5
class Square:
    def area(self):
        return 4 * 4
for shape in [Circle(), Square()]:
    print(shape.area())
```

Example: Using Polymorphism in Python

```
class Circle:
    pi = 3.14
    def __init__(self, redius):
        self.radius = redius

    def calculate_area(self):
        print("Area of circle :", self.pi * self.radius * self.radius)

class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculate_area(self):
        print("Area of Rectangle :", self.length * self.width)

# function
def area(shape):
    # call action
    shape.calculate_area()
# create object
cir = Circle(5)
rect = Rectangle(10, 5)
# call common function
area(cir)
area(rect)
```

Output:

```
Area of circle : 78.5

Area of Rectangle : 50
```

For example, In the below example, **calculate_area() instance method created in both Circle and Rectangle class**. Thus, we can **create a function that takes any object and calls the object's calculate_area() method to implement polymorphism.** Using this object can perform

Polymorphism with class methods is useful when we want objects to perform the same action in different ways. **In the below example, both objects calculate the area (same action) but in a different way (different formulas)**

Here's a line-by-line explanation of the code:

class Circle:

- This defines a new class called Circle.

pi = 3.14

- A class variable pi is set to 3.14, which will be shared across all instances of the class.

def __init__(self, redius):

self.radius = redius

- This is the **constructor method (__init__) for the class.**
- It takes a parameter redius (likely a misspelling of radius) and assigns it to the instance variable self.radius.

def calculate_area(self):

print("Area of circle :", self.pi * self.radius * self.radius)

- This method calculates and prints the area of the circle using the formula: area = $\pi \times \text{radius}^2$.

class Rectangle:

- Defines a new class called Rectangle.

def __init__(self, length, width):

self.length = length

self.width = width

- **Constructor that initializes** the length and width of the rectangle.

def calculate_area(self):

print("Area of Rectangle :", self.length * self.width)

- Method that calculates and prints the area of the rectangle using the formula: area = length × width.

def area(shape):

- Defines a standalone function called **area that takes one argument shape.**

shape.calculate_area()

- **Calls the calculate_area() method on the given shape object.** This works as long as the **object passed has a calculate_area() method — demonstrating polymorphism.**

cir = Circle(5)

- Creates a Circle object with a radius of 5 and stores it in the variable cir.

rect = Rectangle(10, 5)

- Creates a Rectangle object with a length of 10 and width of 5 and stores it in the variable rect.

area(cir)

- Calls the area() function with the Circle object, which results in printing the area of the circle.

area(rect)

- Calls the area() function with the Rectangle object, which results in printing the area of the rectangle.

Inheritance In Python

In an Object-oriented programming language, inheritance is an important aspect. In Python, **inheritance is the process of inheriting the properties of the parent class into a child class.**

The primary purpose of inheritance is the reusability of code. Using inheritance, we can use the existing class to create a new class instead of recreating it from scratch.

Syntax

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

Example: Use of Inheritance in Python

```
# Base class
class Vehicle:
    def __init__(self, name, color, price):
        self.name = name
        self.color = color
        self.price = price

    def info(self):
        print(self.name, self.color, self.price)
```

```
# Child class
class Car(Vehicle):


    def change_gear(self, no):
        print(self.name, 'change gear to number', no)
# Create object of Car
car = Car('BMW X1', 'Black', 35000)
car.info()
car.change_gear(5)
```

Output:

```
BMW X1 Black 35000

BMW X1 change gear to number 5
```

in the below example, From a vehicle class, we are creating a Car class. We don't need to define common attributes and methods again in Car class. We only need to add those attributes and methods which are specific to the Car.

In inheritance, the child class acquires all the data members, properties, and functions of the parent class. Also, a child class can customize any of the parent class methods.

Sure! Let's go through this code line by line and explain what each part does:

 # Base class

class Vehicle:

- This defines a base class named Vehicle.
- It's the parent class that other classes can inherit from.

def __init__(self, name, color, price):

- This is the constructor method of the Vehicle class.
- It initializes the object with name, color, and price.

  self.name = name

  self.color = color

  self.price = price

- **These lines assign the parameters name, color, and price to the** instance variables (self.name, etc.) so that they can be used later in methods.

def info(self):

print(self.name, self.color, self.price)

- This defines a method info() that prints out the vehicle's name, color, and price.
-

 # Child class

class Car(Vehicle):

- This defines a **child class** Car that **inherits from the Vehicle class**.
- It means Car gets **all the attributes and methods of Vehicle by default.**

def change_gear(self, no):

print(self.name, 'change gear to number', no)

- This defines a **new method** change_gear() inside the Car class.
- It prints a message saying which gear the car is changing to.
- **It uses self.name, inherited from Vehicle.**

 # Create object of Car

car = Car('BMW X1', 'Black', 35000)

- This line creates an **instance car of the Car class.**
- It **passes the arguments** 'BMW X1', 'Black', and 35000 to the constructor.
- Since Car doesn't define its own __init__, it uses the Vehicle class’s constructor.

car.info()

- This calls the info() **method (inherited from Vehicle**) on the car object.
- It prints the car's name, color, and price.

car.change_gear(5)

- **This calls the change_gear() method on the car object**, passing 5 as the gear number.
- It prints a message like: BMW X1 change gear to number 5.

Abstraction

Abstraction is one of the important principles of object-oriented programming. It refers to a programming approach by which only the relevant data about an object is exposed, hiding all the other details. This approach helps in reducing the complexity and increasing the efficiency of application development.

Python Abstract Class

In object-oriented programming terminology, a class is said to be an abstract class if it cannot be instantiated, that is you can have an object of an abstract class. You can however use it as a base or parent class for constructing other classes.

Example: Create an Absctract Class

```
from abc import ABC, abstractmethod
# Abstract class
class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass

# Concrete class
class Car(Vehicle):
    def start_engine(self):
        return "Car engine started"

# Another concrete class
class Motorcycle(Vehicle):
    def start_engine(self):
        return "Motorcycle engine started"
v1 = Car()
v2 = Motorcycle()
print(v1.start_engine())    # Output: Car engine started
print(v2.start_engine())    # Output: Motorcycle engine started
```

Explanation Line-by-Line

Line 1

from abc import ABC, abstractmethod

- Importing ABC and abstractmethod from Python's abc module.
- ABC lets you define abstract base classes.
- abstractmethod marks a method that **must** be implemented by subclasses.

Line 3

class Vehicle(ABC):

- Vehicle is an abstract class.
- It inherits from ABC, so it **cannot be instantiated directly**.

Line 4–6

```
@abstractmethod
def start_engine(self):
    pass
```

- start_engine() is an abstract method.
- It has **no body**, just a pass, meaning "subclasses must define this".

Line 8–10

class Car(Vehicle):

def start_engine(self):

return "Car engine started"

- Car is a concrete class that **inherits** from Vehicle.
- It implements the start_engine() method.

Line 12–14

class Motorcycle(Vehicle):

def start_engine(self):

return "Motorcycle engine started"

- Another subclass that provides its own version of start_engine().

✅ **Usage Example**

v1 = Car()

v2 = Motorcycle()

print(v1.start_engine()) # Output: Car engine started

print(v2.start_engine()) # Output: Motorcycle engine started

 Key Benefits of Abstraction:

- Enforces a **standard structure** in subclasses.
- Reduces **code duplication**.
- Makes code **easier to maintain** and **extend**.
- Helps achieve **loose coupling** between components.

Class Method vs Static Method in Python

What is a static method?

Static methods in Python are extremely similar to python class level methods, the difference being that a static method is bound to a class rather than the objects for that class.

This means that a static method can be called without an object for that class. This also means that static methods cannot modify the state of an object as they are not bound to it. Let's see how we can create static methods in Python.

Why and when to use static methods

Static methods are useful when you want to:

- Create methods that don’t need access to instance attributes or methods
- Implement functionality that belongs conceptually to a class but doesn’t require an instance
- Write helper methods that operate on class attributes rather than instance attributes
- Perform operations that are related to the class but don’t need to access or modify class state
- Avoid creating unnecessary instances when you just need to call a function

class Calculator:

```
    # create addNumbers static method
    @staticmethod
    def addNumbers(x, y):
        return x + y
print('Product:', Calculator.addNumbers(15, 110))
```

Note that we called the addNumbers we created without an object.

Advantages of Python static method

Static methods have a very clear use-case. When we need some functionality not w.r.t an Object but w.r.t the complete class, we make a method static. This is pretty much advantageous when we need to create Utility methods as they aren't tied to an object lifecycle usually.

How a static method differ from class method

A static method does not have access to the class or instance state, whereas a class method has access to the class state. This means a static method cannot modify or use class or instance variables, whereas a class method can modify class variables.

Example:

```
class MyClass:
    class_var = "This is a class variable"
    @staticmethod
    def static_method():
        # This method cannot access or modify class_var
        pass
    @classmethod
    def class_method(cls):
        # This method can access and modify class_var
        cls.class_var = "This is a modified class variable"
        pass
```

UNIT-V	Advanced Topics and Applications	8hours
Modules and Packages: Creating and Using Modules, Organizing Code into **Packages Regular Expressions**: Pattern Matching with remodule, String Manipulation, **Introduction to GUI Programming**: Basics of Tkinter. Web Development with Flask(optional):**Basics of Web Development**		

write difference bw function,module,package in python

- A function is a single reusable code block.
- A module is a file containing multiple functions/classes.
- A package is a collection of related modules.

What is Module in python

A module in Python is simply a file containing Python code—functions, classes, or variables—that you can reuse in **other Python** programs.

Or

A module is like a toolbox. Instead of writing the same code over and over, you write it once in a .py file (your module), and then you can use it wherever you need by importing it.

Why Use Modules?

- Reusability – Use your code in multiple programs.
- Organization – Keep related functions together.
- Maintainability – Easier to update or fix.
- Collaboration – Share modules with others.

Types of Modules:

1. **Built-in Modules** – Come with Python (no need to install).
    - **Examples**: math, random, datetime, os
2. **User-defined Modules** – Created by you (custom .py files).
    - **Example:** A file named **mytools.py** with your own functions.
3. **Third-party Modules** – Installed via tools like pip.
    - **Examples:** numpy, pandas, flask

Example of Built-in Module:

```
import math
print(math.sqrt(25))  # Output: 5.0
```

Creating and Using User-defined Modules in python

Program 1: Math Utility Module

math_utils.py

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

def cube(num):
    return num ** 3
```

main2.py

```
from math_utils import square, cube

print(square(4))  # Output: 16
print(cube(3))    # Output: 27
```

◈ Program 2: Module with Constants

constants.py

```
PI = 3.14159
GRAVITY = 9.8
```

main3.py

```
import constants

print("Value of Pi:", constants.PI)
print("Gravity:", constants.GRAVITY)
```

◈ Example 2: Custom Calculator Module

Calculator1.py

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

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        return "Cannot divide by zero"
    return a / b
```

main3.py

```
import calculator as calc

print("Add:", calc.add(10, 5))
print("Subtract:", calc.subtract(10, 5))
print("Multiply:", calc.multiply(10, 5))
print("Divide:", calc.divide(10, 5))
```

Example 3:Temperature Converter Module

temp_converter.py

```
def celsius_to_fahrenheit(c):
    return (c * 9/5) + 32

def fahrenheit_to_celsius(f):
    return (f - 32) * 5/9
```

main.py

```
import temp_converter

print(temp_converter.celsius_to_fahrenheit(37))
print(temp_converter.fahrenheit_to_celsius(98.6))
```

Example 4: Area Calculation Module

area.py

```
def circle_area(radius):
    return 3.14 * radius * radius

def rectangle_area(length, width):
    return length * width
```

main.py

```
import area
print(area.circle_area(5))
print(area.rectangle_area(4, 6))
```

Example 5:Find Reverse a string Module or Palindrome

string_utils.py

```
def reverse_string(s):
    return s[::-1]

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

main.py

```
import string_utils

print(string_utils.reverse_string("hello"))
print(string_utils.is_palindrome("madam"))
```

What is package in python

A **package** in Python is a way of organizing related **modules** into a **directory hierarchy**. It's like a **folder** that contains **multiple .py files.**

in Simple Words:

A **module** is a .py file.
A **package** is a **folder** that contains **one or more modules** (and maybe other sub-packages).

Why Use Packages?

- Organize large codebases
- Group related functionality
- Avoid naming conflicts
- Make code modular and reusable

Structure Example of a Package:

```
my_package /                # Marks this directory as a package
├── module1.py
└── module2.py
```

main.py:Calling Program

***First make** my_package **folder** in Hard Disk then save **module1.py, module2.py** file in my_package folder and store calling program **main.py** store **outside** the **my_package** folder.Please ensure that all file name must be uniquely stored in Hard Disk.

Organizing Code into Packages

File: module1.py

```
def greet():
    print("Hello from module1!")
```

File: module2.py

```
def farewell():
    print("Goodbye from module2!")
```

File: main.py (Store outside the package folder) -This is Calling program.

```
from my_package import module1, module2

module1.greet()
module2.farewell()
```

 # Output: Hello from module1!
 # Output: Goodbye from module2!

Simple Calculator Package

Folder: calcpack/

calcpack/basic.py

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

def subtract(a, b):
    return a - b
```

calcpack/advanced.py

```
def power(a, b):
    return a ** b

def modulus(a, b):
    return a % b
```

main.py

```
from calcpack import basic, advanced

print(basic.add(10, 5))
print(advanced.power(2, 3))
```

Regular Expressions: (re module)

Pattern Matching with **remodule**, String Manipulation Let's combine Regular Expressions (regex) with the re module and string manipulation in Python. This is super useful for searching, cleaning, or modifying text based on patterns.

What is re in Python?

The **re module** in Python stands for **regular expression**.
It provides powerful tools for **pattern matching** and **string manipulation**.

Think of it as a way to search, validate, extract, or modify strings based on patterns.

Python RegEx & Common Methods in the Python 're' Module

Regex is the abbreviation for '**Regular Expressions**' It is a sequence of characters that forms a **search pattern.**

1) search( )

#**Search for an upper case**

"G" character in the beginning of a word, and returns the word:

```
import re
text = "Life is Good"
y = re.search(r"\bG\w+", text)
print(y.group())
```

The output is *'Good'.*

break this code down line by line and explain what each part does:

---

import re

✅ This line imports the re module, which provides support for regular expressions in Python. Regular expressions (regex) are patterns used to match character combinations in strings.

---

text = "Life is Good"

✅ This assigns the string "Life is Good" to the variable text.

---

y = re.search(r"\bG\w+", text)

✅ Here's where the magic happens:

- re.search(...) searches the string text for the first match of the regex pattern.
- **The pattern is**: r"\bG\w+"

Let's break that pattern down:

- **r"..."** is a raw string, so backslashes (\) are treated literally (not as escape characters)
- **\b** is a word boundary, meaning the pattern should start at the beginning of a word.
- **G** matches the uppercase letter G.
- **\w+** matches one or more word characters (letters, digits, or underscores). So together with G, this matches a whole word starting with G.

So the full regex \bG\w+ matches a word that **starts with G, like "Good".**

---

print(y.group())

✅ This prints the actual text that matched the pattern.

- **y is a match object** returned by re.search().
- **.group() returns** the matched string, in this case: "Good"

Search() method either returns None (if the pattern doesn't match), or a re.MatchObject that contains information about the matching part of the string.

This method stops after the first match, so this is best suited for testing a regular expression more than extracting data.

```
import re
text = "The rain in Spain stays mainly in the plain"
match = re.search(r"in", text)
print(match.group())
```

output: in

2) findall( )
All matches are returned in a list.

```
import re

text = "The rain in Spain stays mainly in the plain"
matches = re.findall(r"in", text)
print(matches)
```

 # Output: ['in', 'in', 'in', 'in','in', 'in']

3) split( )
Returns the **list of strings** where **splitting occurred**. If a split pattern is not found, return the original string in the list.

```
import re
text = "Life is good"
y = re.split("\s", text)
print(y)
```

The output is *['Life', 'is', 'good']*

Here, re.split() is used to split the text based on a **pattern**. The pattern **"\s"** represents **any whitespace character**, including spaces, tabs, or newlines.

4) sub( )
Replaces or substitutes the matched pattern with the given string.

```
import re
text = "The rain in Spain"
y = re.sub("i", "9", text)
print(y)
```

***The output** is 'The ra9n 9n Spa9n'*

5.Span()

#Search for an raw string "d" character in the given words, and print its **starting and ending position of first occurrence**:

```
import re
text = "Life is good"
y = re.search(r"d", text)
print(y.span())
```

The output is (11, 12).

introduction to GUI Programming: Basics of Tkinter

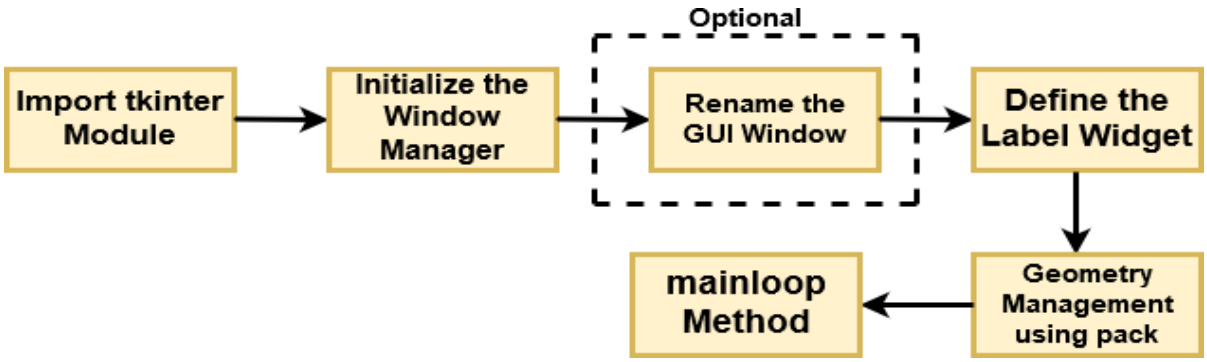
🎨 What is Tkinter?

Tkinter is Python's standard library for creating **Graphical User Interfaces (GUIs)** — windows, buttons, text fields, labels, etc.
It lets you **build desktop applications without any external installations.**

Tkinter is a built-in module in Python that lets you build desktop applications with windows, buttons, text fields, and more—without installing anything extra.
Basics of Tkinter

Here are some **basic Tkinter programs**, each showcasing a different widget. These will help you get started with GUI development in using Tkinter.

w, let's build a very simple GUI with the help of Tkinter and understand it with the help of a flow diagram.



Flow Diagram for Rendering a Basic GUI

How Tkinter Works (Simple Idea)

1. Create a main window (with Tk()).
2. Add widgets (like Label, Button, Entry).
3. Arrange them (pack(), grid(), or place()).
4. Write what happens when users click or type.
5. Run the main event loop (mainloop()) to show everythin

important Components in Tkinter

Widget	Purpose
Label	Display text or images
Button	Button to perform actions
Entry	Single-line text input
Text	Multi-line text area
Frame	Container to group widgets
Listbox	Show a list of selectable items
Checkbutton	Create a checkbox
Radiobutton	Create multiple choices (only one selected)

First Tkinter Program

```
import tkinter as tk
root = tk.Tk()
root.title("Label Example")
root.geometry("600x500")
label = tk.Label(root, text="Hello, Tkinter!")
label.pack()
root.mainloop()
```

Here's a line-by-line explanation of your Tkinter program:

---

import tkinter as tk

- This imports the **Tkinter library** and gives it the **alias tk**, so we can use tk.Label, tk.Tk(), etc., instead of typing tkinter.Label.

---

root = tk.Tk()

- Creates the **main application window**.
- **The Tk()** method in Tkinter is used to create the **main window** of a graphical user interface (GUI) application in Python. It initializes the **Tkinter interpreter** and provides a root window where widgets like buttons, labels, and text fields can be placed.
- **root is your app's base window** — everything else (buttons, labels, etc.) will be placed inside this.

---

root.title("Label Example")

- Sets the **title of the window** that appears in the title bar.

---

root.geometry("600x500")

- Sets the **size of the window** to **600 pixels wide** and **500 pixels tall**.

---

label = tk.Label(root, text="Hello, Tkinter!")

- Creates a **label widget** with the text "Hello, Tkinter!".
- root is passed as the parent, meaning this label will be displayed inside the root window.

---

label.pack()

- This **packs** the label into the window — meaning it tells Tkinter to display it.

- **.pack()** is one of Tkinter's layout managers (others are .grid() and .place()).

---

root.mainloop()

- Starts the **Tkinter event loop**, which waits for user interactions (like clicking a button or closing the window).
- This line **keeps the window open** and responsive.

Widget Methods

These methods control individual widgets:

Method	Purpose
pack(), grid(), place()	Position the widget in the window.
config(options) or configure()**	Change properties (text, color, size, etc.).
cget('option')	Get the current value of an option (like text, bg color).
destroy()	Delete the widget from the window.
update()	Force update/redraw the widget.
after(ms, func)	Run a function after a delay (milliseconds).
bind(event, handler)	Bind a function to an event (like clicking).

---

2. Tk (Root Window) Methods

These manage the whole window:

Method	Purpose
mainloop()	Starts the Tkinter event loop (required to keep the window open).
title("text")	Set the window's title.
geometry("WxH")	Set window width (W) and height (H). Example: "400x300".
resizable(width_bool, height_bool)	Make window resizable or not.
withdraw()	Hide the main window.

---

3. Event Handling Methods

Method	Purpose
bind("<Event>", function)	Handle specific events, like mouse clicks or keypresses.
unbind("<Event>")	Remove an event binding.
focus_set()	Give a widget keyboard focus.

Tkinter Widget Methods (Most Common)

Method	Purpose	Example
pack()	Place widget vertically/horizontally.	button.pack()
grid()	Place widget in a table/grid layout.	entry.grid(row=0, column=1)
place()	Place widget at an exact position (x, y).	label.place(x=50, y=100)
config(options) / configure()**	Change widget properties.	label.config(text="New Text", fg="blue")

Method	Purpose	Example
cget("option")	Get the current value of an option.	text = label.cget("text")
destroy()	Delete the widget.	button.destroy()
update()	Force update/redraw the widget immediately.	root.update()
after(ms, func)	Wait for a certain time, then call a function.	root.after(1000, say_hello)
bind("<event>", function)	Bind a widget to an event (like click).	button.bind("<Button-1>", on_click)
unbind("<event>")	Remove a previously set event binding.	button.unbind("<Button-1>")
focus_set()	Give focus to a widget (for keyboard input).	entry.focus_set()
selection_set(start, end)	(For Listbox) Select a range of items.	listbox.selection_set(0, 2)
selection_clear(start, end)	(For Listbox) Clear selection.	listbox.selection_clear(0, 'end')
insert(index, item)	Add new item to widgets like Listbox, Text, etc.	listbox.insert(tk.END, "Mango")
delete(start, end)	Delete item(s) in widgets like Listbox, Text, etc.	listbox.delete(0)

1. Tkinter Window with Label Widget

```
import tkinter as tk
root = tk.Tk()
root.title("Label Example")
root.geometry("600x500")
label = tk.Label(root, text="Hello, Tkinter!")
label.pack()
root.mainloop()
```

2. Button Widget

```
import tkinter as tk

def say_hello():
    print("Hello!")
root = tk.Tk()
root.title("Button Example")
root.geometry("300x200")
button = tk.Button(root, text="Click Me", command=say_hello)
button.pack()
root.mainloop()
```

---

3. Entry Widget (Text Input)

```
import tkinter as tk
def show_input():
    print(entry.get())
root = tk.Tk()
root.title("Entry Example")
root.geometry("300x200")
entry = tk.Entry(root)
entry.pack()
button = tk.Button(root, text="Submit", command=show_input)
button.pack()
root.mainloop()
```

Explanation:

def show_input():
print(entry.get())

- This defines a function named show_input.
- Inside the function:
  - **entry.get()** retrieves the text entered in the entry widget.
  - print(...) displays that text in the console (terminal).

---

entry = tk.Entry(root)

- Creates an Entry widget — this is a single-line text input box.
- root is its parent, meaning it's placed inside the main window.

entry.pack()
Adds (packs) the entry widget into the window.
It tells Tkinter: “Display this widget.”

button = tk.Button(root, text="Submit", command=show_input)
Creates a Button widget with:
the label text "Submit"
when clicked, it calls the show_input function using the command= argument.

button.pack()
Adds the button to the window layout using pack() so it becomes **visible.root** is its parent, meaning it's placed inside the main window.

4. Checkbox Widget

```
import tkinter as tk

def show_status():
    print("Checked" if var.get() else "Unchecked")
root = tk.Tk()
root.title("Checkbutton Example")
root.geometry("300x200")

var = tk.BooleanVar()

check = tk.Checkbutton(root, text="I agree", variable=var,
command=show_status)
check.pack()
root.mainloop()
```

Explanation:

```
def show_status():
    print("Checked" if var.get() else "Unchecked")
```

This defines a function called **show_status**.When the checkbox is **clicked,** this function is called.
var.get() returns either True or False depending on whether the checkbox is selected.

The line prints:
"Checked" if the checkbox is selected.
"Unchecked" if it's not.

var = tk.BooleanVar()
Creates a Tkinter variable that can store True or False.
It's used to track the state of the checkbox (checked or unchecked).

check = tk.Checkbutton(root, text="I agree", variable=var, **command=show_status)**

Creates a Checkbutton widget

Placed in root (the main window).Displays the text "I agree" beside the checkbox.Linked to the variable var (which tracks whether it's checked).
Calls **show_status** when the checkbox is toggled.

check.pack()
Adds the checkbox to the window layout using the pack() geometry manager so it's displayed

5. Radio Buttons

```
import tkinter as tk

def show_choice():
    print("Selected:", var.get())

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

var = tk.StringVar()

tk.Radiobutton(root, text="Option 1", variable=var, value="Option 1", command=show_choice).pack()
tk.Radiobutton(root, text="Option 2", variable=var, value="Option 2", command=show_choice).pack()

root.mainloop()
```

Explanation

def show_choice():
print("Selected:", var.get())

This defines a function called show_choice.When a radiobutton is selected, this function is called.

var.get() returns the value of the selected radiobutton.
That value is printed to the console with "Selected: ...".

var = tk.StringVar()
Creates a Tkinter string variable.

This variable will hold the value of the selected radiobutton.
Only one radiobutton can be selected at a time — the selected one's value will be stored in var.

tk.Radiobutton(root, text="Option 1", variable=var, value="Option 1", command=show_choice).pack()

this creates the first Radiobutton:

- Parent is root (the main window).
- The label shown next to the button is "Option 1".
- **variable=var:** this radio shares the same control variable, so Tkinter knows it's part of a group.
- **value="Option 1"**: if selected, this value will be stored in var.
- **command=show_choice**: this function will be called when the button is selected.
- **.pack():** adds it to the window layout so it's visible.

The Listbox Widget in Tkinter

The Listbox widget in Tkinter (Python's standard GUI library) provides a way to display a list of items from which users can select one or more items.

Key Features and Methods

Adding Items

- **insert(index, item):** Add items at specified position
    - **END** - adds to the end
    - **ACTIVE** - adds before the active item
    - **Or** use numerical index (0-based)

Deleting Items

- **delete(first, last=None)**: Delete items in range
- **delete(ACTIVE**): Delete the active item
- **delete(0, END**): Clear all items

Selection Modes

- **SINGLE (default**): Only one item can be selected
- **BROWSE**: Similar to SINGLE but can drag to select
- **MULTIPLE**: Multiple items can be selected
- **EXTENDED:** Multiple ranges can be selected with Shift/Ctrl

listbox = Listbox(root, selectmode=EXTENDED)

Getting Selections

- **curselection():** Returns tuple of selected item indices
- **get(first, last=None**): Get item(s) at index/range

Other Useful Methods

- **size():** Number of items in listbox

- **activate(index)**: Set active item
- **see(index)**: Scroll to make item visible

A simple Listbox displaying programming languages.

```
from tkinter import *
root = Tk()
root.title("Basic Listbox")
root.geometry("300x200")
listbox = Listbox(root)
listbox.pack()
items = ["Python", "Java", "C++", "JavaScript", "Ruby"]
for item in items:
    listbox.insert(END, item)
root.mainloop()
```

The line from tkinter import * is an import statement in Python. It means that you are importing all the classes, functions, and variables provided by the Tkinter module into your program.

Essentially, the * acts as a wildcard, enabling you to use Tkinter components without needing to prefix them with tkinter.

For example:

```
from tkinter import *
root = Tk()
label = Label(root, text="Hello, Tkinter!")
label.pack()
root.mainloop()
```

In this code, **Tk(), Label(), and pack()** are **used directly without the tkinter**. prefix, thanks to the from tkinter import * statment.

Multiple Selection Listbox

```
from tkinter import *

def show_selected():
    selected = listbox.curselection()
    for idx in selected:
        print(listbox.get(idx))

root = Tk()
root.title("Multiple Selection")
root.geometry("300x200")
listbox = Listbox(root, selectmode=MULTIPLE)
listbox.pack()
for item in ["Red", "Green", "Blue", "Yellow", "Black"]:
    listbox.insert(END, item)
Button(root, text="Show Selected", command=show_selected).pack()
root.mainloop()
```

Here's a line-by-line explanation

```
from tkinter import *
```

- Imports all classes, functions, and constants from the `tkinter` module, which is Python's standard GUI toolkit.

```
def show_selected():
```

- Defines a function named `show_selected` that will be called when the button is clicked.

```
selected = listbox.curselection()
```

- Gets the indices of currently selected items in the listbox using the `curselection()` method, which returns a tuple of indices.

```
 for idx in selected:
```

- Starts a loop that iterates over each selected index.

```
print(listbox.get(idx))
```

- For each selected index, retrieves the corresponding item text using `listbox.get(idx)` and prints it to the console.

```
root = Tk()
```

- Creates the main application window (the root window) by instantiating the `Tk` class.

```
root.title("Multiple Selection")
```

- Sets the title of the main window to "Multiple Selection".

```
root.geometry("300x200")
```

- Sets the initial size of the main window to 300 pixels wide and 200 pixels tall.

```
listbox = Listbox(root, selectmode=MULTIPLE)
```

- Creates a `Listbox` widget as a child of the root window with multiple selection enabled (`selectmode=MULTIPLE` allows selecting multiple items).

```
listbox.pack()
```

- Packs the listbox into the root window using the `pack()` geometry manager, which sizes it to fit its contents and makes it visible.

```
for item in ["Red", "Green", "Blue", "Yellow", "Black"]:
```

- Starts a loop that iterates over each color name in the list

```
listbox.insert(END, item)
```

- Inserts each color name at the end of the listbox (`END` is a constant that represents the end position).

```
Button(root, text="Show Selected", command=show_selected).pack()
```

- Creates a button widget with the label "Show Selected" that calls the `show_selected` function when clicked, and packs it into the root window.

```
root.mainloop()
```

- Starts the Tkinter event loop, which keeps the application running and responsive to user interactions until the window is closed.

- **4. Add and Delete Items Dynamically**

```
from tkinter import *
def add_item():
    listbox.insert(END, entry.get())
    entry.delete(0, END)
def delete_item():
    selected = listbox.curselection()
    for idx in selected[::-1]: # Delete from last to first
        listbox.delete(idx)
root = Tk()
root.title("Dynamic Listbox")
entry = Entry(root)
entry.pack()
Button(root, text="Add", command=add_item).pack()
Button(root, text="Delete Selected",
command=delete_item).pack()
listbox = Listbox(root)
listbox.pack()
root.mainloop()
```

Description: Dynamically add and remove items from a Listbox.

Move Items Between Two Listboxes

```
From tkinter import *
def move_to_right():
    selected = left_listbox.curselection()
    for idx in selected[::-1]:
        item = left_listbox.get(idx)
        right_listbox.insert(END, item)
        left_listbox.delete(idx)
def move_to_left():
    selected = right_listbox.curselection()
    for idx in selected[::-1]:
        item = right_listbox.get(idx)
        left_listbox.insert(END, item)
        right_listbox.delete(idx)
root = Tk()
root.title("Move Items Between Listboxes")
left_listbox = Listbox(root)
left_listbox.pack(side=LEFT)
for item in ["Apple", "Banana", "Orange"]:
    left_listbox.insert(END, item)
right_listbox = Listbox(root)
right_listbox.pack(side=RIGHT)
Button(root, text=">>", command=move_to_right).pack()
Button(root, text="<<", command=move_to_left).pack()
root.mainloop()
```

Here's a line-by-line explanation

Function: `move_to_right()`

```
def move_to_right():
```

- Defines a function called `move_to_right()` that moves selected items from the **left listbox** to the **right listbox**.

```
    selected = left_listbox.curselection()
```

- Gets the indices of the currently selected items in the **left listbox** (`left_listbox.curselection()` returns a tuple of indices).

```
    for idx in selected[::-1]:
```

- Loops over the selected indices **in reverse order** (`[::-1]` reverses the tuple to avoid index shifting issues when deleting items).

```
        item = left_listbox.get(idx)
```

- Retrieves the text of the item at index `idx` from the left listbox.

```
        right_listbox.insert(END, item)
```

- Inserts the item at the **end** (`END`) of the **right listbox**.

```
        left_listbox.delete(idx)
```

- Deletes the item from the **left listbox** at index `idx`.

---

3. Function: move_to_left()

```
def move_to_left():
```

- Defines a function called move_to_left() that moves selected items from the **right listbox** back to the **left listbox**.

```
    selected = right_listbox.curselection()
```

- Gets the indices of the currently selected items in the **right listbox**.

```
    for idx in selected[::-1]:
```

- Loops over the selected indices **in reverse order** (to prevent index shifting issues).

```
        item = right_listbox.get(idx)
```

- Retrieves the text of the item at index idx from the right listbox.

```
        left_listbox.insert(END, item)
```

- Inserts the item at the **end** (END) of the **left listbox**.

```
        right_listbox.delete(idx)
```

- Deletes the item from the **right listbox** at index idx.

5. Left Listbox Setup

```
left_listbox = Listbox(root)
```

- Creates a **left listbox** widget inside the root window.

```
left_listbox.pack(side=LEFT)
```

- Packs the left listbox to the **left side** of the window using pack(side=LEFT).

```
for item in ["Apple", "Banana", "Orange"]:
    left_listbox.insert(END, item)
```

- Inserts the items **"Apple"**, **"Banana"**, and **"Orange"** into the left listbox (`END` places each new item at the bottom).

---

6. Right Listbox Setup

```
right_listbox = Listbox(root)
```

- Creates a **right listbox** widget inside the root window.

```
right_listbox.pack(side=RIGHT)
```

- Packs the right listbox to the **right side** of the window using `pack(side=RIGHT)`.

---

7. Buttons to Move Items

```
Button(root, text=">>", command=move_to_right).pack()
```

- Creates a button (`>>`) that calls `move_to_right()` when clicked (moves items from left to right).

```
Button(root, text="<<", command=move_to_left).pack()
```

- Creates a button (`<<`) that calls `move_to_left()` when clicked (moves items from right to left).

Listbox with Filter (Search)

```
from tkinter import *

def filter_list():
    search_term = entry.get().lower()
    listbox.delete(0, END)
    for item in all_items:
        if search_term in item.lower():
            listbox.insert(END, item)

root = Tk()
root.title("Searchable Listbox")

entry = Entry(root)
entry.pack()
Button(root, text="Search", command=filter_list).pack()

all_items = ["Python", "Java", "JavaScript", "C++"]
listbox = Listbox(root)
listbox.pack()

filter_list()  # Initialize list
root.mainloop()
```

Case-Insensitive Search: Converts both input and items to lowercase before comparison.

Example:Addition of two number program using TKINTER

```
import tkinter as tk
def add_numbers():
    try:
        num1 = float(entry1.get())
        num2 = float(entry2.get())
        result = num1 + num2
        result_label.config(text=f"Result: {result}")
    except ValueError:
        result_label.config(text="Please enter valid numbers")

# Create main window
root = tk.Tk()
root.title("Addition App")
root.geometry("300x200")

# Entry for first number
tk.Label(root, text="Enter first number:").pack()
entry1 = tk.Entry(root)
entry1.pack()

# Entry for second number
tk.Label(root, text="Enter second number:").pack()
entry2 = tk.Entry(root)
entry2.pack()

# Button to add numbers
add_button = tk.Button(root, text="Add", command=add_numbers)
add_button.pack(pady=10)

# Label to display result
result_label = tk.Label(root, text="Result: ")
result_label.pack()

# Run the GUI
root.mainloop()
```

What is Flask?

- **Flask** is a **lightweight** and **simple web framework** for Python.
- It helps you quickly create websites, APIs, or web applications.
- It’s called a "**microframework**" because it doesn’t force you to use a lot of complicated tools — you can keep it very simple if you want!

🛠 o install and uninstall Flask, you can use pip, the Python package manager.

✅ To Install Flask

Open your terminal or command prompt and run:

```
pip install flask
```

If you're using Python 3 specifically:

```
pip3 install flask
```

You can verify the installation with:

```
python -m flask --version
```

❌ To Uninstall Flask

Run the following command:

```
pip uninstall flask
```

Create your first Flask app

Example:

```
from flask import Flask

app = Flask(__name__)   # Create a Flask app

@app.route('/')         # Home page route
def home():
    return "Hello, World!"

if __name__ == '__main__':
    app.run(debug=True)  # Run the app
```

✨ **Explanation:**

Code	What it does
from flask import Flask	Import the Flask class.
app = Flask(__name__)	Create an app object.
@app.route('/')	Define a **route** (a URL) — in this case, the home page /.
def home():	Function that runs when you visit the / route.
return "Hello, World!"	Shows text in the browser.
app.run(debug=True)	Starts a web server on your computer. debug=True helps you see error messages easily while coding.

Let's go through this program line by line.

Importing flask module in the project is mandatory. An object of Flask class is our **WSGI** application.

Let's break it down step by step.

1. Import Flask

from flask import Flask

- Flask is a lightweight web framework for Python.
- This line imports the Flask class from the Flask module so that we can create a web application.

2. Create a Flask App

app = Flask(__name__)

- Flask(__name__) initializes a Flask web application.
- The __name__ **argument** tells Flask that this is **the main module.**

4. Define a Route

@app.route('/')

- @app.route('/') is a decorator that tells Flask to execute the function below when someone accesses '/' (the root URL).
- This means when a user visits your website's main page, it will call the hello_world() function.

4. Define the Function

```
def hello_world():
  return 'Hello World'
```

- hello_world() is the function that gets executed **when the root URL is visited.**
- It returns 'Hello World', which will be displayed as a **response on the webpage**

5. Run the Flask App

```
if __name__== '__main__':
   app.run()
```

- The if __name__== '__main__': block ensures that the Flask app only runs when this script is executed directly.
- app.run() starts the **Flask web server**, making the application **accessible via a web browser.**

How It Works

- When you **run this script, Flask starts a local web server.**
- Opening **http://127.0.0.1:5000/ in your browser will display** "Hello World" as a response.

That URL **http://127.0.0.1:5000/** refers to a **local web server** running on your computer.

Breaking It Down:

- 127.0.0.1 → This is the **loopback address**, meaning it refers to your own computer. It's commonly called **localhost**.
- :5000 → This is the **port number** where your **Flask application is running.**
- http:// → This means it's using the **HTTP protocol** for communication.

Why Does This Matter?

- If you're using Flask and have started your app with app.run(), this means you can access your application in a browser by visiting **http://127.0.0.1:5000/**.
- Since it's a local address, only **your machine** can access it—it's not available on the internet.

*** Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)**

Output: Hello World message will be displayed on **web browser**