

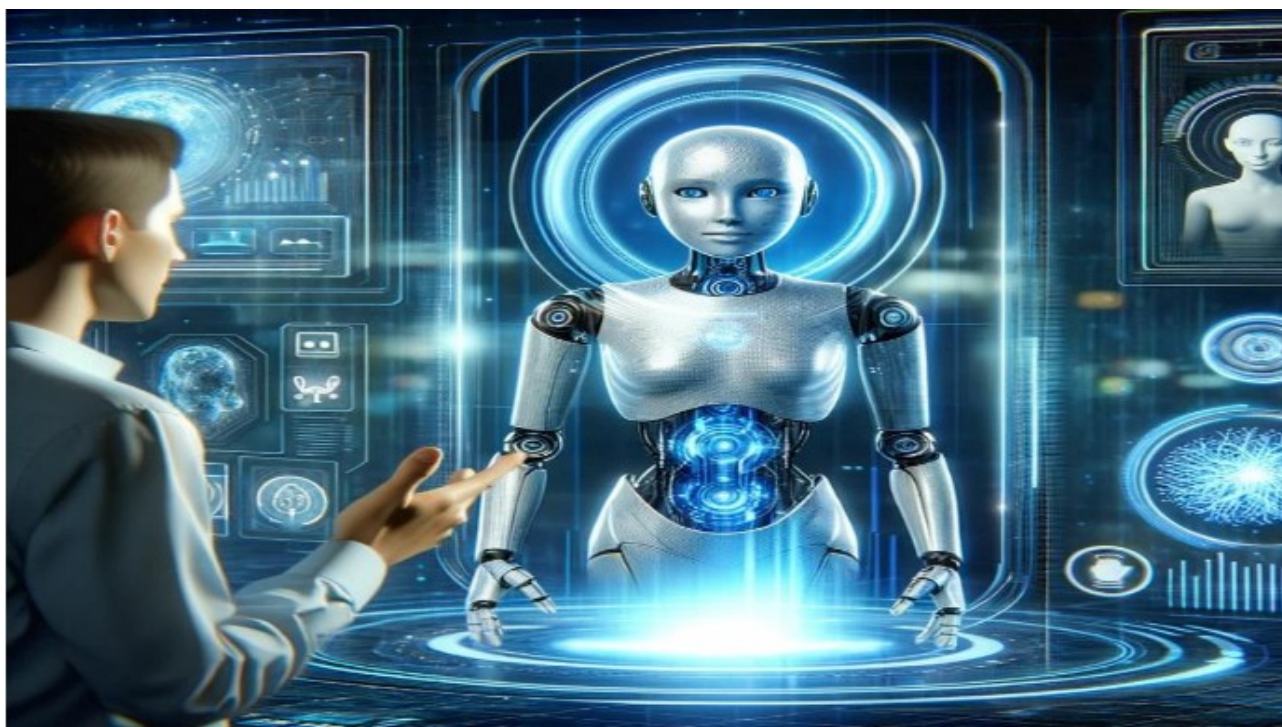
Draft Study Material

Artificial Intelligence Assistant

(QP Code: NIE/SSC/Q1003)

**Sector: Information Technology-Information Technology
Enable Services (IT-ITeS)**

Grade X



PSS CENTRAL INSTITUTE OF VOCATIONAL EDUCATION
(a constituent unit of NCERT, under Ministry of Education, Government of India)
Shyamla Hills, Bhopal- 462 002, M.P., India
<http://www.psscive.ac.in>

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise without the prior permission of the publisher.

Preface

Vocational Education is a dynamic and evolving field, and ensuring that every student has access to quality learning materials is of paramount importance. The journey of the PSS Central Institute of Vocational Education (PSSCIVE) toward producing comprehensive and inclusive study material is rigorous and time-consuming, requiring thorough research, expert consultation, and publication by the National Council of Educational Research and Training (NCERT). However, the absence of finalized study material should not impede the educational progress of our students. In response to this necessity, we present the draft study material, a provisional yet comprehensive guide, designed to bridge the gap between teaching and learning, until the official version of the study material is made available by the NCERT. The draft study material provides a structured and accessible set of materials for teachers and students to utilize in the interim period. The content is aligned with the prescribed curriculum to ensure that students remain on track with their learning objectives.

The contents of the modules are curated to provide continuity in education and maintain the momentum of teaching-learning in vocational education. It encompasses essential concepts and skills aligned with the curriculum and educational standards. We extend our gratitude to the academicians, vocational educators, subject matter experts, industry experts, academic consultants, and all other people who contributed their expertise and insights to the creation of the draft study material.

Teachers are encouraged to use the draft modules of the study material as a guide and supplement their teaching with additional resources and activities that cater to their students' unique learning styles and needs. Collaboration and feedback are vital; therefore, we welcome suggestions for improvement, especially by the teachers, in improving upon the content of the study material.

This material is copyrighted and should not be printed without the permission of the NCERT-PSSCIVE.

Deepak Paliwal
(Joint Director)
PSSCIVE, Bhopal

April, 2025

STUDY MATERIAL DEVELOPMENT COMMITTEE

Members

Deepak D. Shudhalwar, Professor (CSE), Department of Engineering and Technology, PSSCIVE, NCERT, Bhopal, Madhya Pradesh

Prakash Khanale, Sr Coordinator, School of Computer Science, YCMOU, Nashik, Ex-Professor and Head, Department of Computer Science, DSM College, Parbhani, Maharashtra

Member Coordinator

Deepak D. Shudhalwar, Professor (CSE), Head, Department of Engineering and Technology, PSSCIVE, NCERT, Bhopal, Madhya Pradesh

CONTENT

Module 1. Python Programming

Session 1. Control Structures in Python

Session 2. Functions in Python

Module 2. Data Science

Session 1. Introduction to NumPy

Session 2. Array Manipulation using NumPy

Session 3. Array Computation using NumPy

Module 3. Data Analysis

Session 1. Introduction to Pandas

Session 2. Coding with Pandas

Session 3. Data Visualisation using Matplotlib

Module 4. Neural Network

Session 1. Artificial Neural Network (ANN)

Session 2 Applications of Neural Network

Session 3. Machine Learning Tools

Module 5. AI Project

Session 1. Project Guidelines

Session 2. Project Formats

Session 3. Project Review

Session 4. Sample Project

Module 1. Python Programming

In our daily life there are some situations where we have to follow a fixed sequence of steps to complete a task and in other situations we have choices of steps to complete a task. For example, in air travel every passenger has to follow the following steps to board an airplane.

1. Show ticket and identity proof at the main entrance of the Airport.
2. Scanning of luggage.
3. Take your boarding pass.
4. Pass security check.
5. Finally, board the plane.

Passengers don't have the choice to change the sequence of above written steps to board on the airplane. On the other hand, Passengers can take their boarding pass from the airline's check-in counter or they can also get a boarding pass at an electronic kiosk nearby as shown in Figure 1.1. Similarly, in programming generally statements are executed from beginning to end as in the sequence they are written. But we may encounter some situations in programming also where statements are required to change the normal sequence of execution. In this unit we are going to discuss the control flow with decisions and loops.



Fig. 1.1: Steps to board on airplane

Session 1. Control Structures in Python

In computer programming a series of statements executed in top-down order. The order of execution of the statements in a program is known as **flow of control**. The flow of control can be implemented using **control structures**. When the program needs to make some decisions depending on different situations, then this is achieved using control flow statements. Python supports two types of control structures – **selection and repetition**. The **if statement** is of selection type, whereas **for and while** structure are of repetition type.

There are three types of *if statements* in Python - **if, for and while**. The statement *if* and *for* is of the selection type and *while* is of repetition type.

In this session, you will understand these control structures with their syntax and how to use it in Python programming.

1.1 if Statement

In real life when we need to make some decisions and based on these decisions, we decide what we should do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we execute the next block of code. Decision making statements in programming languages decide the direction of flow of program execution. Decision making statements available in Python are:

- *if statement*
- *if...else statements*
- *if-elif ladder*

1.1.1 if statement

The if statement is used to check a condition: if the condition is true, we run a block of statements (called the *if-block*).

Syntax

The syntax for if statement is as follows.

```
if test expression:  
    statement(s)
```

The program evaluates the *test expression* and will execute statement(s) only if the text expression is **True**. If the text expression is **False**, the statement(s) is not executed.

In Python, the body of the *if* statement is indicated by the **indentation**. Body starts with an indentation and the first unindented line marks the end. Python interprets *non-zero* values as *True*. None and 0 are interpreted as *False*.

Flowchart

The flowchart of if statement is shown in Figure 1.2.

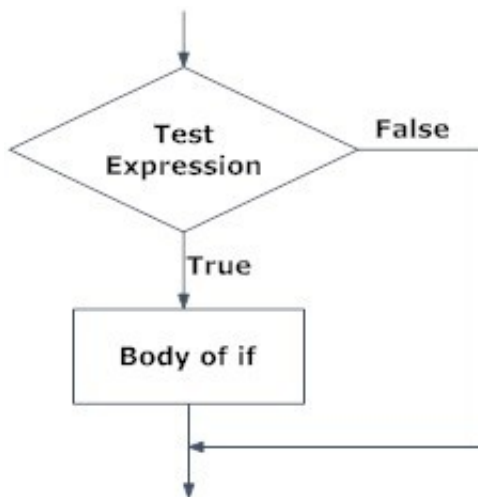


Fig: Operation of if statement

Fig. 1.2 : Flowchart of if statement

Example 1.1: Let us take an example to understand the control flow of if statement from the following example. The program check the age entered by the user to make the decision that whether a person is eligible to vote or not. In India if the age is greater than or equal to 18 the person is eligible to vote.

```

File Edit Format Run Options Window Help
# Program to check the eligibility of vote using if statement
age = int(input("Enter your age : "))
if age >= 18:
    print("You are ELIGIBLE to vote")
Ln: 4 Col: 38
  
```

When you run the program with three different inputs of age 20, 18 and 17, the output will be as below:

```

File Edit Shell Debug Options Window Help
=====
Enter your age : 20
You are ELIGIBLE to vote
>>>
=====
Enter your age : 18
You are ELIGIBLE to vote
>>>
=====
Enter your age : 16
>>>
  
```

In the above example, if the age entered by the user is greater than 18, then the message is printed as **"You are ELIGIBLE to vote"**. If the condition is true, then the indented statement(s) are executed otherwise not.

The indentation implies that its execution is dependent on the condition. There is no limit on the number of statements that can appear as a block under the if statement.

1.1.2 if...else Statement

A variant of if statement called *if...else* statement that allows writing two alternative paths and the control condition determines which path gets executed.

Syntax

The syntax for *if...else* statement is as follows.

```
if test expression:  
    Body of if  
else:  
    Body of else
```

The *if...else* statement evaluates test expression and will execute the body of if only when the test condition is True.

If the condition is False, the body of another is executed. Indentation is used to separate the blocks.

Flowchart

The flowchart of *if...else* statement is shown in Figure 1.3.

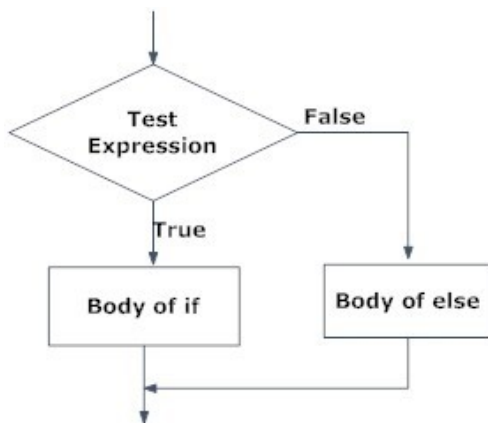


Fig. 1.3 : Flowchart of if...else statement

Example 1.2: Let us now modify the code in Example 1.1 to illustrate the use of *if else* structure. The modified code is as below.

```
File Edit Format Run Options Window Help
# Program to check the eligibility of vote using if statement
age = int(input("Enter your age : "))
if age >= 18:
    print("You are ELIGIBLE to vote")
else:
    print("You are NOT ELIGIBLE to vote")
Ln: 6 Col: 4
```

When you run the program with three different inputs of age 18, 20, and 17, the output will be as below. Observe the difference with the output in Example 1.1.

```
File Edit Shell Debug Options Window Help
===== RESTART:
Enter your age : 18
You are ELIGIBLE to vote
>>>
===== RESTART:
Enter your age : 20
You are ELIGIBLE to vote
>>>
===== RESTART:
Enter your age : 17
You are NOT ELIGIBLE to vote
>>>
```

In this example, if the age entered by the person is greater than or equal to 18, s/he can vote. Otherwise, the person is not eligible to vote.

1.1.3 *if...elif...else Statement*

There may be a situation, when you have multiple conditions to check and these all conditions are independent to each other. You can use *elif* statements to include multiple conditional expressions after the *if* condition or between the *if* and *else* control structure.

Syntax

The syntax for a selection structure using *if... elif... else* is as below.

if test expression:

 Body of *if*

elif test expression:

 Body of *elif*

else:

 Body of *else*

The *elif* is short for ***else if***. It allows to check for multiple expressions. If the condition for *if* is ***False***, it checks the condition of the next *elif* block and so on. If all the conditions are ***False***, the body of *else* is executed. Only one block

among the several *if...elif...else* blocks is executed according to the condition. The if block can have only one else block. But it can have multiple elif blocks.

Flowchart

The flowchart of *if...else* statement is shown in Figure 1.4

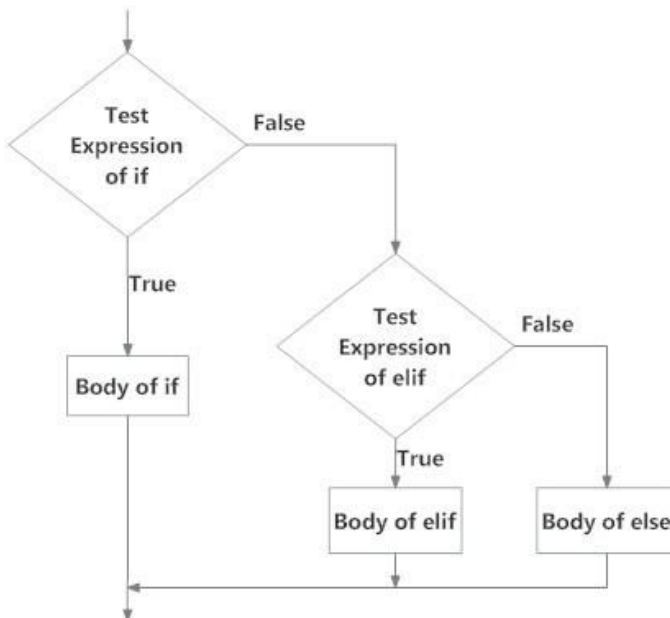


Fig. 1.4 : Flowchart of *if...elif...else* statement

Example 1.3 : Let us now modify the code in Example 1.2 to illustrate the use of *if...elif...else* structure.

```
File Edit Format Run Options Window Help
# Program to check the eligibility of vote
# using if elif else statement
age = int(input("Enter your age : "))
if age == 18:
    print("You become just ELIGIBLE to vote")
elif age > 18:
    print ("You are ELIGIBLE to vote")
else:
    print ("You are NOT ELIGIBLE to vote")
Ln: 5 Col: 28
```

When you run the program with for three different inputs of age 18,19,17 the output will be as below. Observe the difference with the output in Example 1.1. and 1.2.


```
File Edit Shell Debug Options Window Help
Enter your age : 18
You become just ELIGIBLE to vote
>>>
===== RESTART:
Enter your age : 19
You are ELIGIBLE to vote
>>>
===== RESTART:
Enter your age : 17
You are NOT ELIGIBLE to vote
>>> |
```

1.1.4 Nested if statements

Nested if statements have *if...elif...else* statement inside another *if...elif...else* statement. This is called nesting in computer programming.

The number of statements can be nested inside one another. Indentation is the only way to identify the level of nesting. This get confusing, so must be avoided if it can be.

Example 1.4: of nested ... if Statement

Let us now modify the above code to illustrate the use of nested *if* structure.

```
File Edit Format Run Options Window Help
# Program to check the eligibility of vote
# using nested if statement
age = int(input("Enter your age : "))
if age >= 18:
|   if age == 18:
|       print("You become just ELIGIBLE to vote")
|   else:
|       print ("You are ELIGIBLE to vote")
else:
    print ("You are NOT ELIGIBLE to vote")
Ln: 5 Col: 0
```

When you run the program for three different inputs of age 18,19,17 the output will be as below. Observe the difference with the output in Example 1.1, 1.2 and 1.3.


```
File Edit Shell Debug Options Window Help
Enter your age : 18
You become just ELIGIBLE to vote
>>>
===== RESTART:
Enter your age : 19
You are ELIGIBLE to vote
>>>
===== RESTART:
Enter your age : 17
You are NOT ELIGIBLE to vote
>>> |
```

Example 1.5: Write a program using *nested if* to check whether a number is positive, negative, or zero.

```
File Edit Format Run Options Window Help
# Program to check the number is positive,
# negative or zero using nested if statement
number = int(input("Enter the number : "))
if number > 0:
    print("The number is positive")
elif number < 0:
    print("The number is negative")
else:
    print("The number is zero")
Ln: 3 Col: 0
```

When you run the program with four different inputs of number the output will be as below.

```
File Edit Shell Debug Options Window Help
Enter the number : 11
The number is positive
>>>
===== RESTART:
Enter the number : -5
The number is negative
>>>
===== RESTART:
Enter the number : 0
The number is zero
>>> |
```

Example: 1.6 Write a program to display the appropriate message as per the color of signal at the road crossing.

```
File Edit Format Run Options Window Help
# Program to display the message as per
# colour of signal at the road crossing
signal = input("Enter the colour : ")
if signal == "RED" or signal == "red":
    print("STOP")
elif signal == "ORANGE" or signal == "orange":
    print("Go Slow")
elif signal == "GREEN" or signal == "green":
    print("Go Ahead!")
else:
    print("Wait for the Signal to glow")
Ln: 10 Col: 5
```

When you run the program with four different inputs of number the output will be as below.

```
File Edit Shell Debug Options Window Help
Enter the colour : red
STOP
>>>
===== |
Enter the colour : ORANGE
Go Slow
>>>
===== |
Enter the colour : GREEN
Go Ahead!
>>>
===== |
Enter the colour :
Wait for the Signal to glow
>>> |
```

Number of *elif* is dependent on the number of conditions to be checked. If the first condition is false, then the next condition is checked, and so on. If one of the conditions is true, then the corresponding indented block executes, and the if statement terminates.

Example 1.7: Let us write a program to create a simple calculator to perform basic arithmetic operations on two numbers. The program should do the following:

- Accept two numbers from the user.
- Ask user to input any of the operator (+, -, *, /).
- An error message is displayed if the user enters anything else.
- Display only positive difference in case of the operator "-".
- Display a message "Please enter a value other than 0" if the user enters the second number as 0 and operator '/' is entered.

```
File Edit Format Run Options Window Help
# Program of Calculator to perform
# Four basic arithmetic operations
result = 0
val1 = float(input("Enter value 1: "))
val2 = float(input("Enter value 2: "))
op = input("Enter the Operator (+, -, *, /): ")
if op == "+":
    result = val1 + val2
elif op == "-":
    if val1 > val2:
        result = val1 - val2
    else:
        result = val2 - val1
elif op == "*":
    result = val1 * val2
elif op == "/":
    if val2 == 0:
        print("Error ! Division by Zero")
    else:
        result = val1/val2
else:
    print("Wrong input, Program terminated")
print("The Result is : ", result)
Ln: 26 Col: 0
```

When you run the program with four different inputs of number the output will be as below.

```
File Edit Shell Debug Options Window Help
Enter value 1: 45
Enter value 2: 68
Enter the Operator (+, -, *, /): +
The Result is : 113.0
>>>
===== RESTART:
Enter value 1: 78
Enter value 2: 89
Enter the Operator (+, -, *, /): -
The Result is : 11.0
>>>
===== RESTART:
Enter value 1: 85
Enter value 2: 45
Enter the Operator (+, -, *, /): -
The Result is : 40.0
>>>
===== RESTART:
Enter value 1: 12
Enter value 2: 5
Enter the Operator (+, -, *, /): *
The Result is : 60.0
>>>
===== RESTART:
Enter value 1: 90
Enter value 2: 0
Enter the Operator (+, -, *, /): /
Error ! Division by Zero
The Result is : 0
>>>
===== RESTART:
Enter value 1: 50
Enter value 2: 10
Enter the Operator (+, -, *, /): /
The Result is : 5.0
```

In the above program, for the operators "-" and "/", there exists an *if...else* condition within the *elif* block. This is called nested if. There can be many levels of nesting inside *if...else* statements.

It is possible to use *if ... elif* structure as illustrated. In the latest version of Python, a much more powerful and flexible construct called Structural Pattern Matching is available. It can be used as a simple switch statement but is capable of much more.

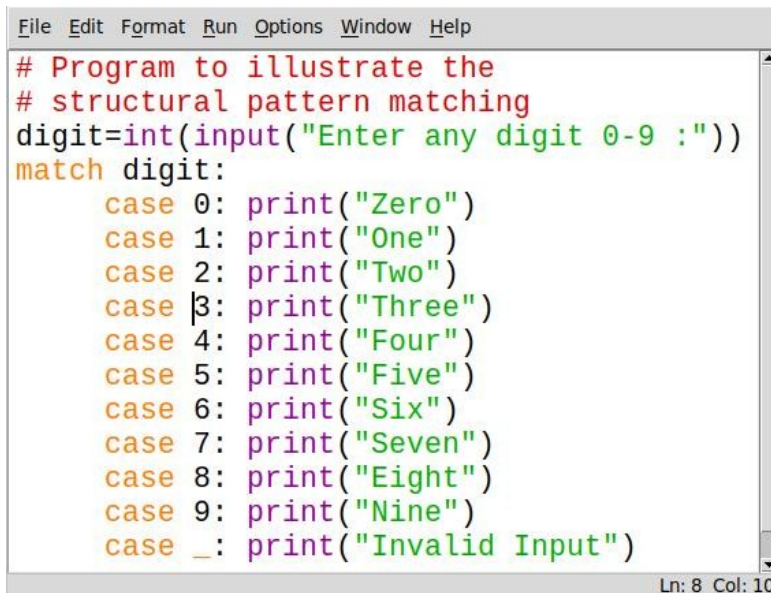
1.1.5 Structural pattern matching

Structural pattern matching introduces the *match...case* statement and the pattern syntax to Python. The *match...case* statement follows the same basic outline as *switch...case* in other programming languages. It takes an object, tests it against one or more match patterns, and takes an action if it finds a match.

Python performs matches by going through the list of cases from top to bottom. On the first match, Python executes the statements in the corresponding case block, then skips to the end of the match block and continues with the rest of

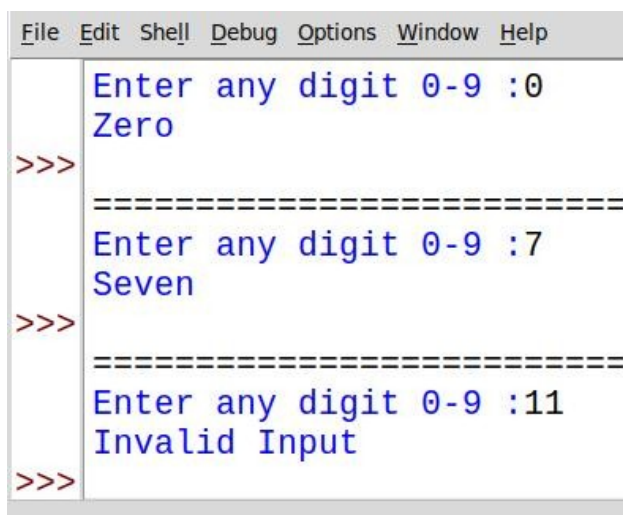
the program. There is no “*fall-through*” between cases, but it’s possible to design your logic to handle multiple possible cases in a single case block. Let us take an example to use structural pattern matching.

Example 1.8: Write a program to illustrate the use of structural pattern matching.



```
File Edit Format Run Options Window Help
# Program to illustrate the
# structural pattern matching
digit=int(input("Enter any digit 0-9 :"))
match digit:
    case 0: print("Zero")
    case 1: print("One")
    case 2: print("Two")
    case 3: print("Three")
    case 4: print("Four")
    case 5: print("Five")
    case 6: print("Six")
    case 7: print("Seven")
    case 8: print("Eight")
    case 9: print("Nine")
    case _: print("Invalid Input")
Ln: 8 Col: 10
```

In the above program, values entered by the user are stored in variable digits. The values written after the case will be matched with the value of the digit one by one. If digit = 0 as specified in case 0 then statement to print zero will be executed. If the digit is not equal to 0, control will go to the next case statement specified as case 1. If digit =1, a statement to print one will be executed. Similarly, it will go on for case 2 to case 9. For the last case, if the digit is anything except digits 0 to 9, It will print “*Invalid input*”.



```
File Edit Shell Debug Options Window Help
Enter any digit 0-9 :0
Zero
>>>
=====
Enter any digit 0-9 :7
Seven
>>>
=====
Enter any digit 0-9 :11
Invalid Input
>>>
```

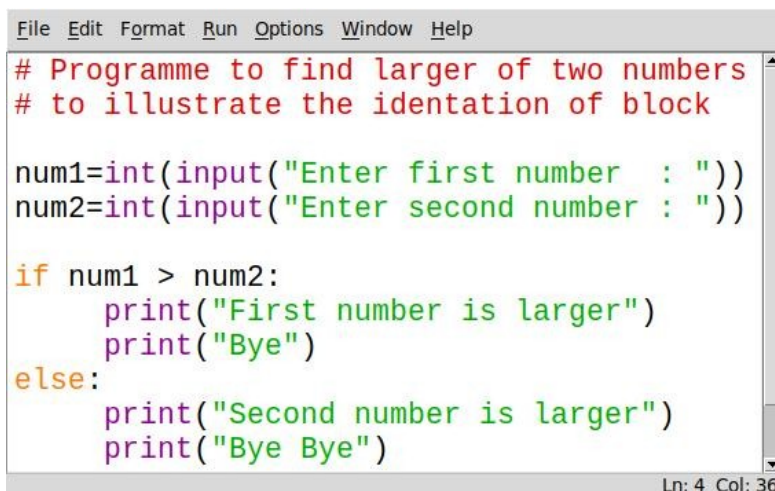

Assignment 1.1

1. Write a program to check whether a number is divisible by 7 or not.
2. Write a program to check whether an alphabet is a vowel or consonant.
3. Write a program to input the month number and print the month name.
4. Write a program to check if a triangle is equilateral, isosceles or scalene on the basis of the length of the sides provided by the user.

1.1.6 Indentation

In most programming languages, the statements within a block are put inside curly brackets. However, Python uses indentation for block as well as for nested block structures. Leading whitespace (spaces and tabs) at the beginning of a statement is called indentation. In Python, the same level of indentation associates statements into a single block of code. The interpreter checks indentation levels very strictly and throws up syntax errors if indentation is not correct. It is a common practice to use a single tab for each level of indentation.

In the following program the if-else statement has two blocks and the statements in each block are indented with the same amount of spaces or tabs.



```
File Edit Format Run Options Window Help
# Programme to find larger of two numbers
# to illustrate the indentation of block

num1=int(input("Enter first number : "))
num2=int(input("Enter second number : "))

if num1 > num2:
    print("First number is larger")
    print("Bye")
else:
    print("Second number is larger")
    print("Bye Bye")

Ln: 4 Col: 36
```

In the above program, the condition *num 1 > num 2* is false for the values of *num1* and *num2* taken in the program. Therefore, the block of statements associated with *else* will be executed as shown in the output.

```
File Edit Shell Debug Options Window Help
Enter first number : 25
Enter second number : 23
First number is larger
Bye
>>>
=====
Enter first number : 45
Enter second number : 56
Second number is larger
Bye Bye
```

1.2 Repetition

Sometimes we need to repeat tasks such as payment of electricity bills to be paid every month. Let us take an example which illustrates an iterative process in nature also. Figure 11.3 shows the phases of the day like morning, midday and evening. These phases are repeated every day in the same order.

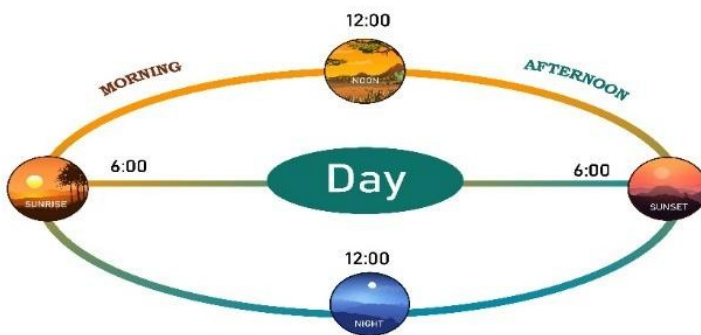


Fig. 1.5 : Phases of the day

Another example where we repeat steps is calculation of mean marks for each student of the class separately. First, we list the steps to calculate mean of the marks scored in 5 subjects by a student

1. Read the marks scored by the student in five subjects.
2. Calculate sum of the marks of all 5 subjects.
2. Divide the sum by 5.
3. Result will be stored as mean.

Above steps will be repeated for each student of the class to get mean marks. This kind of repetition is also called iteration. Repetition of a set of statements in a program is made possible using looping constructs. To understand further, let us look at the following program to print the first 5 natural numbers.

```
File Edit Format Run Options Window Help
# Program to print first
# five natural numbers
print("First 5 natural numbers")
print(1)
print(2)
print(3)
print(4)
print(5)
Ln: 9 Col: 0
```

```
File Edit Shell Debug Options Window Help
Python 3.12.3 (main, Jan
Type "help", "copyright"
>>>
=====
First 5 natural numbers
1
2
3
4
5
>>>
```

In the above program, `print ()` function is used 5 times to print 5 different natural numbers. But in the situation to print the first 100,000 natural numbers, it will not be efficient to write 100,000 `print` statements. In such cases it is better to use loop or repetition in the program.

Looping constructs provide the facility to execute a set of statements in a program repetitively, based on a condition. The statements in a loop are executed again and again as long as the particular logical condition remains true. This condition is checked based on the value of a variable called the loop control variable. When the condition becomes false, the loop terminates. It is the responsibility of the programmer to ensure that this condition eventually does become false so that there is an exit condition and it does not become an infinite loop. For example, if we did not set the condition `count <= 100000`, the program would have never stopped. There are two looping constructs in Python - ***for*** and ***while***. Let us learn these looping constructs in detail.

1.2.1 For Loop

The *for* statement is used to iterate over a range of values or a fixed number of sequences. The for loop is executed for each of the items in the range. These values can be *numeric*, *string*, *list*, or *tuple*.

Syntax

```
for <control-variable> in <sequence/items in range>:  
    <statements inside body of the loop>
```

With every iteration of the loop, the control variable checks whether each of the values in the range have been traversed or not. When all the items in the range are exhausted, the control is then transferred to the statement(s) immediately following the for loop. In for loop, it is known in advance how many number of times the loop will execute.

Flowchart

The flowchart depicting the execution of for loop is given in Figure 1.6.

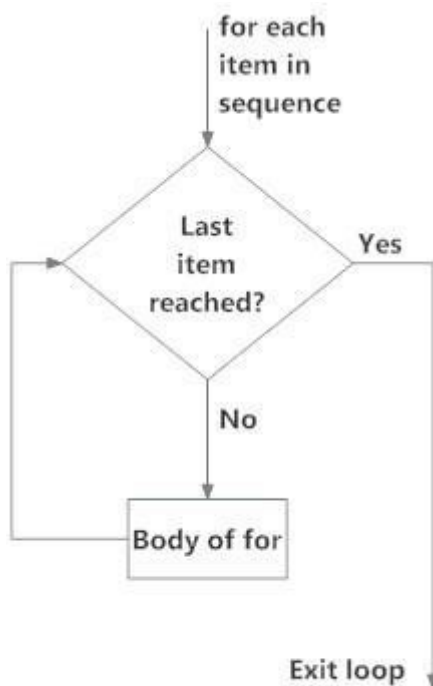


Fig. 1.6 : Flow chart of for loop

Example 1.9 : Write a program using a for loop to find the sum of all numbers stored in a list.

```
File Edit Format Run Options Window Help  
# Program to find the sum of all numbers stored in a list  
numbers = [6,5,3,7,9,1,11] # List of numbers  
sum = 0 # variable to store the sum  
for val in numbers: # control variable to iterate  
    sum = sum + val # Adding the no. in the list sum  
print("The sum is :",sum) # Sum of all no. stored in list  
Ln: 8 Col: 0
```

When you run the program, the output will be:

```
File Edit Shell Debug Options Window Help
=====
The sum is : 42
>>> |
```

In the above program, numbers are a sequence of integers. Variable *val* represents different integers in different iterations of the for loop. So, it prints the sum of all elements in the list.

Example 1.10: Write a program to check the number as even or odd stored in a list.

```
File Edit Format Run Options Window Help
# Program to check the number as EVEN or ODD from the list
numbers = [6,5,8,7,4,1,11]          # List of numbers
for val in numbers:                 # control variable
    if (val % 2) ==0:                # Check for EVEN number
        print(val, 'is EVEN number') # Number is EVEN
    else:
        print(val, 'is ODD number')  # Number is ODD
Ln: 5 Col: 38
```

When you run the program, the output will be:

```
File Edit Shell Debug Options Window Help
=====
6 is EVEN number
5 is ODD number
8 is EVEN number
7 is ODD number
4 is EVEN number
1 is ODD number
11 is ODD number
```

The above program illustrates the use of for loop in Python. Let us also take a look at how the range function can be used with a for loop.

The Range() Function

The range () is a built-in function in Python. Syntax of range () function is:

range (start, stop, step)

It is used to create a list containing a sequence of integers from the given start value upto stop value (excluding stop value), with a difference of the given step value. Following examples illustrate the use of the range() function for various conditions.

```
File Edit Shell Debug Options Window Help
>>> # Illustrate the range() function
>>> # Print the single digit numbers
>>> list(range(0,10,1))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> # Print the single digit odd numbers
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
>>> # Print the single digit even numbers
>>> list(range(2,10,2))
[2, 4, 6, 8]
>>> # Print the double digit numbers in step 10
>>> list(range(10,99,10))
[10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> # Print the single digit number in decreasing order
>>> list(range(9,0,-1))
[9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> # When start and step not specified
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

Ln: 15 Col: 19

In function `range()`, start, stop and step are parameters. More on functions is covered in next chapter. The start and step parameters are optional. If the start value is not specified, by default the list starts from 0. If step is also not specified, by default the value increases by 1 in each iteration. It will take `start=0` and `step=1` by default.

All parameters of `range()` function must be integers. The step parameter can be a positive or a negative integer excluding zero. Negative value of step parameter of `range()` function will generate a decreasing sequence as illustrated in the above example.

The function `range()` is often used in for loops for generating a sequence of numbers as illustrated below.

```
File Edit Shell Debug Options Window Help
>>>
>>> for num in range(10):
...     print(num)
...
...
0
1
2
3
4
5
6
7
8
9
>>>
```

Ln: 102 Col: 0

In above python code, function `range(10)` will generate a sequence of numbers `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`. Variable `num` will hold the elements of this sequence

one by one for different iterations of the for loop. In each iteration of the for loop *print(num)* statement will be executed. Therefore, all the elements of the sequence will be printed one by one as shown in output.

Assignment 1.2

1. Write a program to print odd numbers from the given list using a for loop.
2. Write a program to display a list that stores squares of the elements of a given list.
3. Write a program using a for loop that prompts the user for a hobby 3 times, then appends each one to a list named hobbies. Display the elements of list hobbies.

1.2.2 While Loop

The while statement allows to repeatedly execute a block of statements as long as a condition is true.

The control condition of the while loop is executed before any statement inside the loop is executed. After each iteration, the control condition is tested again and the loop continues as long as the condition remains true. When this condition becomes false, the statements in the body of the for loop are not executed and the control is transferred to the statement immediately following the body of the while loop. If the condition of the while loop is initially false, the body is not executed even once.

A while statement is an example of what is called a looping statement. A while statement can have an optional else clause.

Syntax

```
while test expression :  
    body of while
```

The statements within the body of the while loop must ensure that the condition eventually becomes false, otherwise the loop will become an infinite loop, leading to a logical error in the program.

Flowchart

The flowchart of the while loop is shown in Figure 1.7.

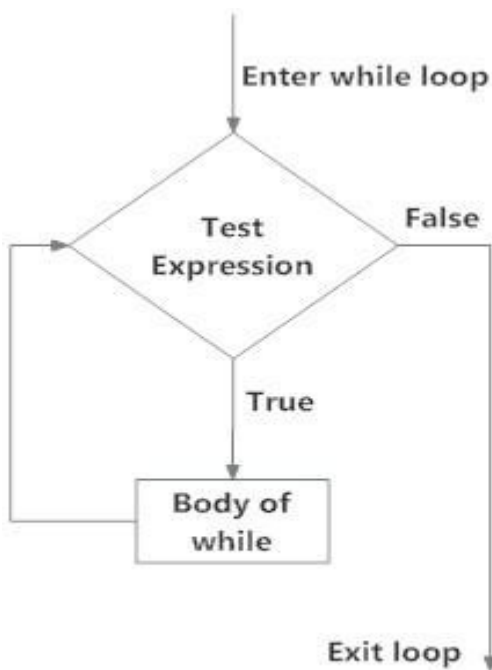


Fig. 1.7 : Flowchart of while Loop

Example 1.11 : Following program will illustrate the use of a while loop.

```

File Edit Format Run Options Window Help
# Program to add n natural numbers
# Input n number of natural no.from user
n = int(input("Enter n: "))
# initialize sum and counter
sum = 0
i = 1
while i <= n:
    sum = sum + i
    i = i+1
# update counter
# print the sum
print("Sum of",n, "natural numbers is:| ", sum)
Ln: 12 Col: 38
  
```

When you run the program, the output will be:

```

File Edit Shell Debug Options Window Help
===== RESTART:
Enter n: 5
Sum of 5 natural numbers is: 15
>>>
===== RESTART:
Enter n: 10
Sum of 10 natural numbers is: 55
>>>
  
```

Assignment 1.3

1. Write a program to print the first 10 even numbers using a while loop.
2. Write a program to find the sum of the digits of a number accepted from the user.
3. Write a program to reverse the number accepted from the user using a while loop.
4. Write a program to check if the input number is Palindrome or not.
5. Write a program to check if the input number is Armstrong or not.

1.3 Break and Continue Statement

Looping constructs allow programmers to repeat tasks efficiently. In certain situations, when some particular condition occurs, we may want to exit from a loop or skip some statements of the loop before continuing further in the loop. This can be achieved by using *break* and *continue* statements, respectively. Python provides these statements as a tool to give more flexibility for the programmer to control the flow of execution of a program.

1.3.1 Break Statement

The *break* statement alters as it terminates the current loop and resumes execution of the statement following that loop. The flowchart of the *break* statement is shown in Figure 1.8.

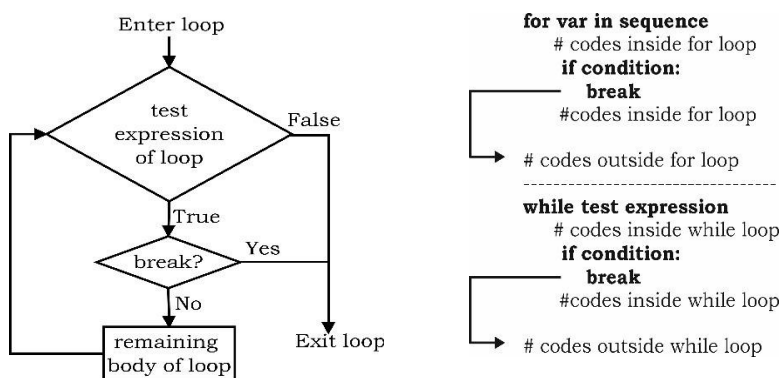


Fig. 1.8: Flowchart for using break statement in loop

Example 1.12: Let us take a program to demonstrate use of break in for loop.

```
# Program to demonstrate use use of break statement
num = 0
for num in range(10):
    num = num +1
    if num == 8:
        break
print('Number has value'+str(num))
print('Encountered break !! Out of loop')
```



```
File Edit Format Run Options Window Help
# Program to demonstrate use use of break statement
num = 0
for num in range(10):
    num = num +1
    if num == 8:
        break
print('Number has value'+str(num))
print('Encountered break !! Out of loop')
Ln: 11 Col: 0
```

When you run the program, the output will be:

```
File Edit Shell Debug Options Window Help
===== RESTART
Number has value8
Encountered break !! Out of loop
>>> |
```

In the above program, when the value of *num* becomes 8, the break statement is executed and the for loop terminates.

Example 1.13: Consider another example to find the sum of all positive numbers using a *while* loop. When the number entered is negative stop taking further input and display the sum.

```
# Find the sum of all positive numbers by taking the
# input using a while loop. Stop taking input when the
# number entered is negative using break statement
sum=0
print("Enter the number :")
while True:
    num = int(input()) # typecast string to integer
    if (num<0):
        break
    sum = sum + num
print("Sum = ",sum)
```

```
File Edit Format Run Options Window Help
# Find the sum of all positive numbers by taking the
# input using while loop. Stop taking input when the
# number entered is negative using break statement
sum=0
print("Enter the number :")
while True:
    num = int(input()) # typecast string to integer
    if (num<0):
        break
    sum = sum + num
print("Sum = ",sum)
Ln: 10 Col: 0
```

When you run the program, the output will be:

```
File Edit Shell Debug Options Window Help
Enter the number :
1
2
3
4
5
6
7
8
9
0
-1
Sum = 45
>>> |
```

Example 1.14: Consider the following example to check if the number entered is prime or not.

```
# Program to check the input number is prime or not
num=int(input("Enter the number to check :"))
flag=0 # Assign flag as 0
if num>1:
    for i in range(2,int(num/2)):
        if (num % i == 0):
            flag = 1
            # number is not prime
    if flag==1:
        print(num,"is not a prime number")
    else:
        print(num,"is a prime number")
else:
    print("Number entered is <=1, execute again!")
```



```
File Edit Format Run Options Window Help
# Program to check the input number is prime or not
num=int(input("Enter the number to check :"))
flag=0 # Assign flag as 0
if num>1:
    for i in range(2,int(num/2)):
        if (num % i == 0):
            flag = 1
            # number is not prime
    if flag==1:
        print(num,"is not a prime number")
    else:
        print(num,"is a prime number")
else:
    print("Number entered is <=1, execute again!")
Ln: 5 Col: 0
```

When you run the program, the output will be:

```
File Edit Shell Debug Options Window Help
Enter the number to check :0
Number entered is <=1, execute again!
>>> ===== RESTART: /I
Enter the number to check :1
Number entered is <=1, execute again!
>>> ===== RESTART: /I
Enter the number to check :2
2 is a prime number
>>> ===== RESTART: /I
Enter the number to check :3
3 is a prime number
>>> ===== RESTART: /I
Enter the number to check :4
4 is a prime number
>>> ===== RESTART: /I
Enter the number to check :5
5 is a prime number
>>> ===== RESTART: /I
Enter the number to check :6
6 is not a prime number
>>> ===== RESTART: /I
Enter the number to check :7
7 is a prime number
>>> ===== RESTART: /I
Enter the number to check :8
8 is not a prime number
>>> ===== RESTART: /I
Enter the number to check :9
9 is not a prime number
>>>
```

Assignment

1. Write a program to find the location of a particular item in a list.
2. Find output of the following Python code:
for val in "string":

```

if val == "i":
    break
print(val)
print("The end")

```

1.3.2 Continue Statement

When a continue statement is encountered, the control skips the execution of remaining statements inside the body of the loop for the current iteration and jumps to the beginning of the loop for the next iteration. If the loop's condition is still true, the loop is entered again, else the control is transferred to the statement immediately following the loop. Figure 1.9 shows the flowchart of the continue statement.

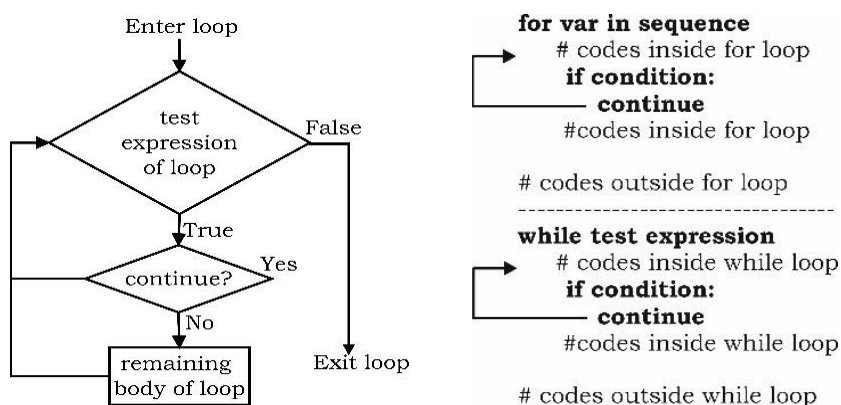


Fig. 1.9: Flowchart of continue statement

Example 1.15: Following program illustrates the use of the continue statement to find even and odd numbers from 0 to 9.

```

# Demonstrate the use of continue statement
# Program to print ODD/EVEN numbers from 0-9
num = 0
print("ODD numbers from 0-9")
for num in range(9):
    num = num + 1
    if num%2 == 0:
        continue
    print("The next ODD number is :",num)
print('End of for loop')
print("EVEN numbers from 0-9")
for num in range(9):
    num = num + 1
    if num%2 != 0:
        continue
    print("The next EVEN number is:",num)
print('End of for loop')

```

```
File Edit Format Run Options Window Help
# Demonstrate the use of continue statement
# Program to print ODD/EVEN numbers from 0-9
num = 0
print("ODD numbers from 0-9")
for num in range(9):
    num = num + 1
    if num%2 == 0:
        continue
    print("The next ODD number is :",num)
print('End of for loop')
print("EVEN numbers from 0-9")
for num in range(9):
    num = num + 1
    if num%2 != 0:
        continue
    print("The next EVEN number is:",num)
print('End of for loop')
```

Ln: 11 Col: 28

When you run the program, the output will be:

```
File Edit Shell Debug Options Window Help
ODD numbers from 0-9
The next ODD number is : 1
The next ODD number is : 3
The next ODD number is : 5
The next ODD number is : 7
The next ODD number is : 9
End of for loop
EVEN numbers from 0-9
The next EVEN number is: 2
The next EVEN number is: 4
The next EVEN number is: 6
The next EVEN number is: 8
End of for loop
>>>
```

Observe that the value 3 is not printed in the output, but the loop continues after the continue statement to print other values till the for loop terminates.

Assignment

1. Write a program to print values from 1 to 20 except multiples of 3.
2. Write a program to print all the prime numbers from 1 to 50.

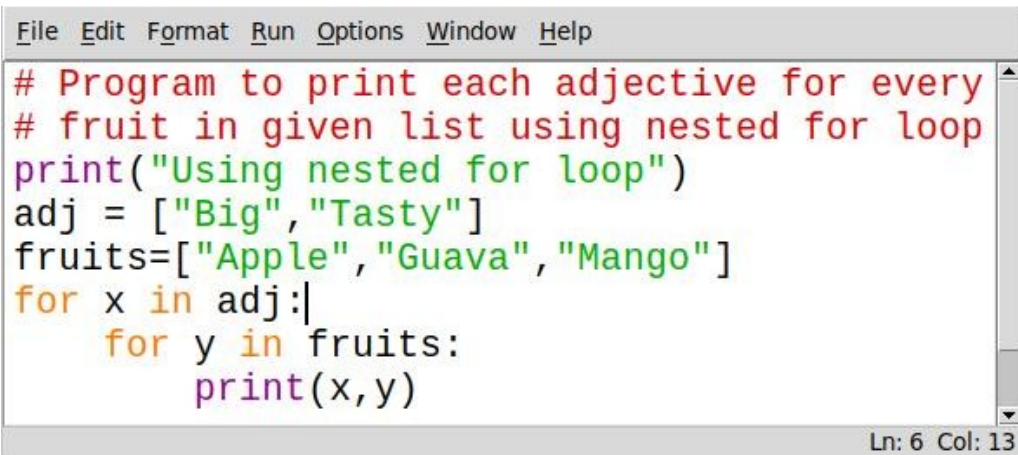
1.3.3 Nested For Loops

A loop may contain another loop inside it. A loop inside another loop is called a nested loop. The inner or outer loop can be any type, such as a while loop or for loop. For example, the outer for loop can contain a while loop and vice versa. The outer loop can contain more than one inner loop. There is no limitation on the chaining of loops.

Nested loops are typically used for working with number and star pattern programs. These are also useful to work with multidimensional data structures, such as printing two-dimensional arrays, iterating a list that contains a nested list.

Example 1.16: Following program illustrates the use of nested for loop.

```
# Program to print each adjective for every
# fruit in given list using nested for loop
print("Using nested for loop")
adj = ["Big", "Tasty"]
fruits = ["Apple", "Guava", "Mango"]
for x in adj:
    for y in fruits:
        print(x, y)
```

A screenshot of a code editor window with a menu bar (File, Edit, Format, Run, Options, Window, Help). The code is color-coded: comments are red, print statements are green, and variable names are blue. The code is the same as the one in the previous block. The status bar at the bottom right shows "Ln: 6 Col: 13".

```
# Program to print each adjective for every
# fruit in given list using nested for loop
print("Using nested for loop")
adj = ["Big", "Tasty"]
fruits = ["Apple", "Guava", "Mango"]
for x in adj:
    for y in fruits:
        print(x, y)
```

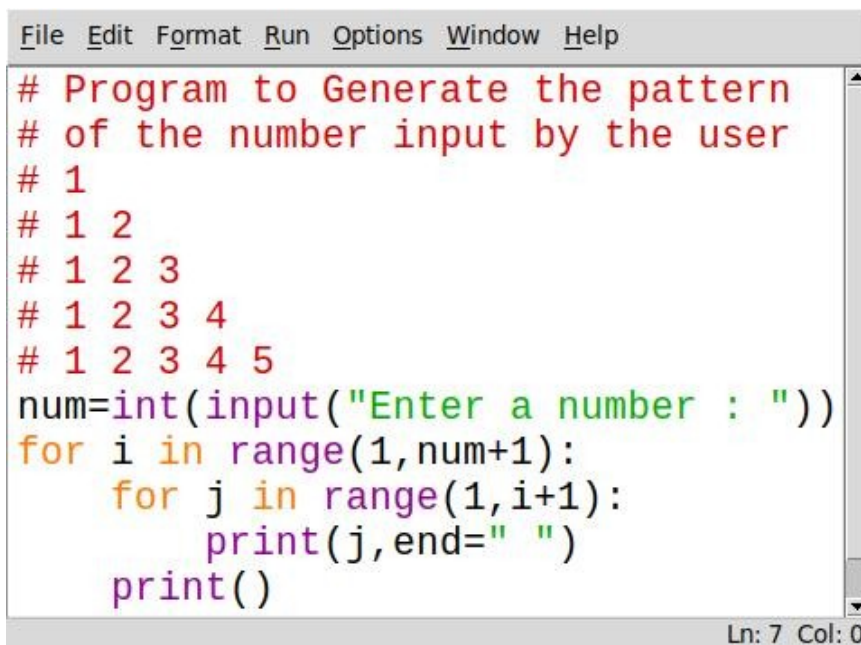
When you run the program, the output will be:

A screenshot of a shell window with a menu bar (File, Edit, Shell, Debug, Options, Window, Help). The output of the program is displayed in blue text. The prompt ">>>" is visible at the bottom left.

```
Using nested for loop
Big Apple
Big Guava
Big Mango
Tasty Apple
Tasty Guava
Tasty Mango
>>>
```

Example 1.17: Consider another program to print the pattern input by the user.

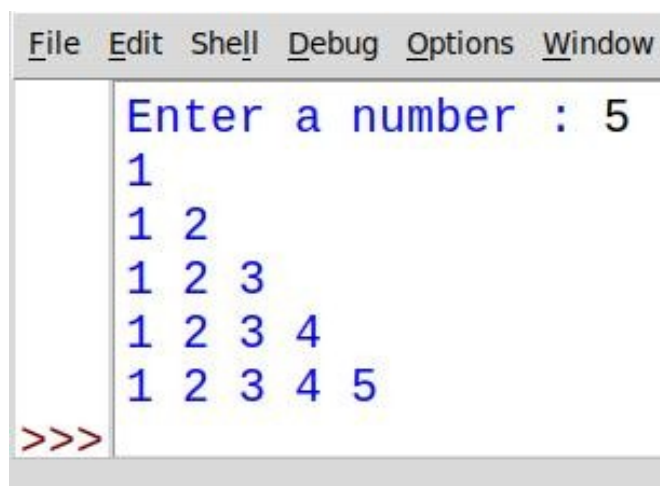
```
# Program to Generate the pattern
# of the number input by the user
# 1
# 1 2
# 1 2 3
# 1 2 3 4
# 1 2 3 4 5
num=int(input("Enter a number : "))
for i in range(1,num+1):
    for j in range(1,i+1):
        print(j,end=" ")
    print()
```

A screenshot of a Python IDE window. The menu bar at the top includes File, Edit, Format, Run, Options, Window, and Help. The code editor contains the same Python program as shown in the previous block. The code is color-coded: comments are red, the input prompt is green, and the loop variables and print statements are purple. The status bar at the bottom right indicates 'Ln: 7 Col: 0'.

```
# Program to Generate the pattern
# of the number input by the user
# 1
# 1 2
# 1 2 3
# 1 2 3 4
# 1 2 3 4 5
num=int(input("Enter a number : "))
for i in range(1,num+1):
    for j in range(1,i+1):
        print(j,end=" ")
    print()
```

Ln: 7 Col: 0

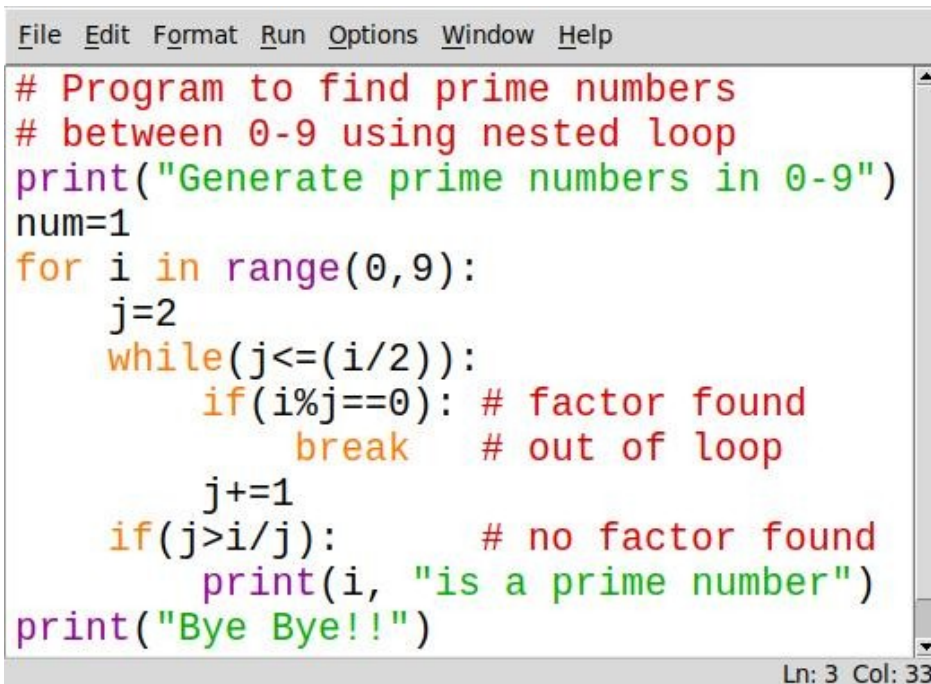
When you run the program, the output will be:

A screenshot of a Python IDE window showing the output of the program. The menu bar includes File, Edit, Shell, Debug, Options, and Window. The Shell window displays the input 'Enter a number : 5' and the resulting pattern of numbers. The prompt '>>>' is visible at the bottom left of the Shell window.

```
Enter a number : 5
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
>>>
```


Example 1.18: Consider another program to find the prime numbers between 1 to 10 using a nested loop.

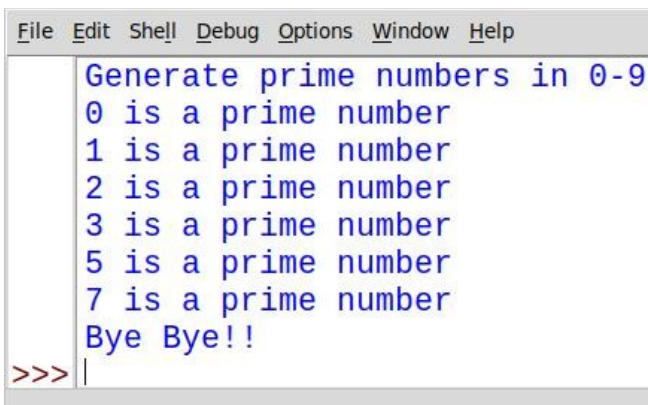
```
# Program to find prime numbers
# between 0-9 using nested loop
print("Generate prime numbers in 0-9")
num=1
for i in range(0,9):
    j=2
    while(j<=(i/2)):
        if(i%j==0): # factor found
            break    # out of loop
        j+=1
    if(j>i/j):      # no factor found
        print(i, "is a prime number")
print("Bye Bye!!")
```

A screenshot of a code editor window. The menu bar at the top includes File, Edit, Format, Run, Options, Window, and Help. The code is color-coded: red for comments, green for print statements, orange for loop keywords, and black for other code. The code is identical to the one in the previous block. The status bar at the bottom right shows "Ln: 3 Col: 33".

```
# Program to find prime numbers
# between 0-9 using nested loop
print("Generate prime numbers in 0-9")
num=1
for i in range(0,9):
    j=2
    while(j<=(i/2)):
        if(i%j==0): # factor found
            break    # out of loop
        j+=1
    if(j>i/j):      # no factor found
        print(i, "is a prime number")
print("Bye Bye!!")
```

Ln: 3 Col: 33

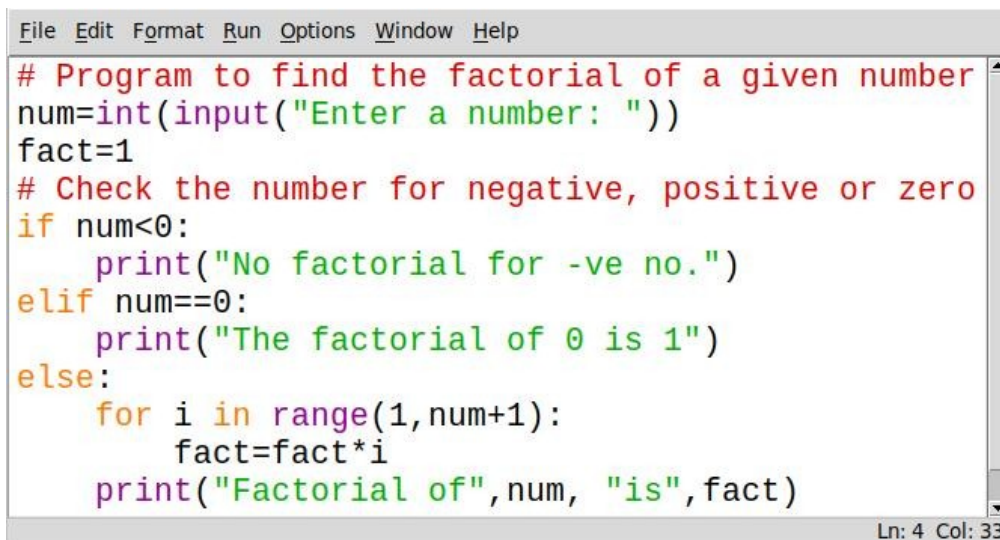
When you run the program, the output will be:

A screenshot of a shell window. The menu bar at the top includes File, Edit, Shell, Debug, Options, Window, and Help. The output is displayed in blue text. The prompt is >>>. The output is:

```
>>> | Generate prime numbers in 0-9
0 is a prime number
1 is a prime number
2 is a prime number
3 is a prime number
5 is a prime number
7 is a prime number
Bye Bye!!
```

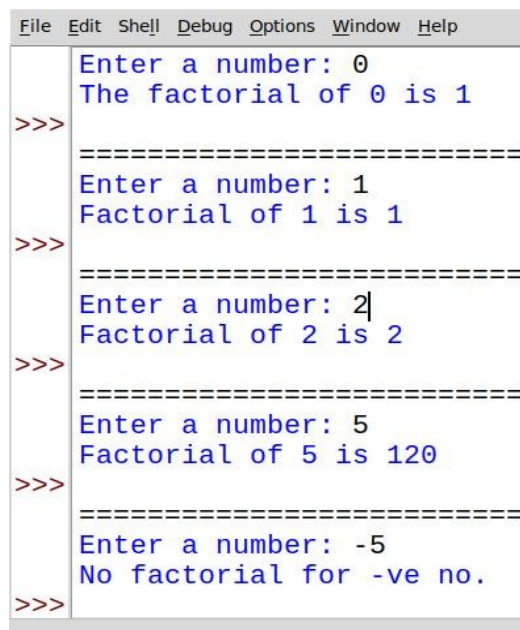
Example 1.19: Consider another program to find the factorial of a given number.

```
# Program to find the factorial of a given number
num=int(input("Enter a number: "))
fact=1
# Check the number for negative, positive or zero
if num<0:
    print("No factorial for -ve no.")
elif num==0:
    print("The factorial of 0 is 1")
else:
    for i in range(1,num+1):
        fact=fact*i
    print("Factorial of",num, "is",fact)
```



The screenshot shows a code editor window with a menu bar (File, Edit, Format, Run, Options, Window, Help) and a text area containing the same Python code as above. The code is color-coded: comments are red, keywords are orange, strings are green, and variables/numbers are black. The status bar at the bottom right indicates 'Ln: 4 Col: 33'.

When you run the program, the output will be:



The screenshot shows a terminal window with a menu bar (File, Edit, Shell, Debug, Options, Window, Help). The output of the program is displayed, showing prompts and results for inputs 0, 1, 2, 5, and -5. The output is color-coded: prompts are blue and results are black. The terminal shows the program running in a shell environment with a prompt '>>>>'.

Assignment

1. Write a program using nested loops to produce a rectangle of * with 6 rows and 20 * per row.
2. Write a program to print a pattern like:

a.	b.	c.	d.	e.
1	1	A	1	5 4 3 2 1
2 2	2 1	A B	1 2	4 3 2 1
3 3 3	3 2 1	A B C	1 2 3	3 2 1
4 4 4 4	4 3 2 1	A B C D	1 2 3 4	2 1
5 5 5 5 5	5 4 3 2 1	A B C D E	1 2 3 4 5	1

CHECK YOUR PROGRESS**A. Multiple Choice Questions**

1. Python supports two types of control structures (a) permutation and combination (b) module and library (c) built-in and user-defined (d) selection and repetition
2. In programming, the concept of decision making or selection is implemented with the help of (a) if else statement (b) for loop (c) while loop (d) Functions
3. In Python, the same level of indentation associate statements into (a) a single block of code (b) a single loop (c) a single function (d) a single control structure
4. Which key is used for each level of indentation (a) \t (b) \n (c) @ (d) #
5. Repetition of a set of statements in a program is made possible using (a) if-else (b) loops (c) data types (d) functions
6. When the condition associated with a loop becomes false, the loop (a) terminates (b) continues (c) fails (d) produce error
7. Which function is used to create a list containing a sequence of integers from the given start value up to stop value (excluding stop value), with a difference of the given step value (a) range() (b) random() (c) maths() (d) int()
8. The start and step parameters of range() function are (a) optional (b) mandatory (c) default (d) invalid
9. All parameters of range() function must be (a) integer (b) float (c) list (d) tuple
10. The step parameter of range() function can be a positive or a negative integer excluding (a) 0 (b) 1 (c) -1 (d) -2

11. Function `range(x)` will generate a sequence of numbers `[0,1,2,3,4,5,6,7,8,9,10]`. Then `x=?` (a) 10 (b) 11 (c) -10 (d) -11
12. Which of the following is not used as for loop in Python? (a) for loop (b) while loop (c) do-while loop (d) None of the above
13. In a Python program, a control structure: (a) defines program-specific data structures (b) directs the order of execution of the statements in the program (c) dictates what happens before the program starts and after it terminates (d) None of the above
14. How many times will the loop run? (a) 2 (b) 3 (c) 1 (d) 0

```
i=2
while(i>0):
    i=i-1
```

15. What will be the output of the following code? (a) 12 (b) 1, 2 (c) 0 (d) Error

```
x = 12
for i in x:
    print(i)
```

B. State whether True or False

1. There is no limit on the number of statements that can appear as a block under the if statement.
2. The statements in a loop are executed again and again as long as the particular logical condition remains false.
3. The range () is a built-in function in Python.
4. The while statement executes a block of code repeatedly as long as the control condition of the loop is true.
5. There is no limitation on the chaining of loops.
6. Keyword *"break"* can be used to bring control out of the current loop.
7. A loop becomes an infinite loop if a condition never becomes FALSE.
8. If the condition is TRUE the statements of if block will be executed otherwise the statements in the else block will be executed.
9. Do-while is a valid loop in Python.
10. Break and continue are jump statements.

C. Fill in the Blanks

1. The order of execution of the statements in a program is known as _____.
2. Python uses _____ for blocks as well as for nested block structures.

3. The interpreter checks indentation levels very strictly and throws up _____ errors if indentation is not correct.
4. The _____ clause can occur with an if as well as with loops.
5. The break statement _____ the current loop and resumes execution of the statement following that loop.
6. When a continue statement is encountered, the control _____ the execution of remaining statements inside the body of the loop for the current iteration and jumps to the beginning of the loop for the next iteration.
7. A loop inside another loop is called a _____ loop.
8. Common use of nested loops is to print various _____ and _____ pattern.
9. For working with _____ data structures, such as printing two-dimensional arrays, nested loops are typically used.
10. The statement to check if a is equal to b is if _____.

D. Programming Questions

1. Write a program that takes the input name and age and displays a message whether the user is eligible to apply for a driving license or not. The eligible age is 18 years.
2. Write a program to print the table of a given number entered by the user.
3. Write a program that prints the minimum and maximum of five numbers entered by the user.
4. Write a program to check if the year entered by the user is a leap year or not.
5. Write a program to generate the sequence: $-5, 10, -15, 20, -25, \dots$ up to n , where n is an integer input by the user.
6. Write a program to find the sum of $1 + \frac{1}{8} + \frac{1}{27} + \dots + \frac{1}{n}$, where n is entered by the user.
7. Write a program to find the sum of digits of an integer number, input by the user.
8. Write a function that checks whether an input number is a palindrome or not.
9. Write a program to find the largest number of a list of numbers entered through the keyboard.
10. Write a program to input N numbers and then print the second largest number.

Session 2. Functions in Python

In a marriage ceremony, a lot of work has to be done related to catering, decoration and other things. In general, we assign these different tasks to a particular group of people who are service providers to complete the work smoothly. For example, catering work is assigned to caterers and decoration work is assigned to decorators. This makes the process easy to manage. Similarly, in programming also we use functions to do a specific task to make our program modular and easy to read.



Fig. 2.1: Different tasks in a wedding ceremony

In programming, the use of function is one of the means to achieve modularity and reusability. Function can be defined as a named group of instructions that accomplish a specific task when it is invoked. Once defined, a function can be called repeatedly from different places of the program without writing all the codes of that function every time, or it can be called from inside another function, by simply writing the name of the function and passing the required parameters, if any. The programmer can define as many functions as desired while writing the code.

In this session, you will understand the concept of functions and the benefits of using functions. We will discuss user defined functions, flow of execution, scope of a variable and standard libraries in Python programming.

2.1 Python Functions

A function is a block of code that performs a specific task. Dividing a complex problem into smaller chunks makes our program easy to understand and reuse. Suppose we need to create a program to make a circle and color it. We can create two functions to solve this problem:

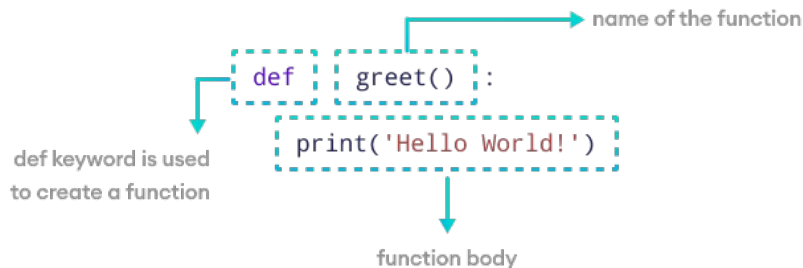
1. function to create a circle
2. function to color the shape

Creating a Function

Example 2.1: Let's create a function `greet()` to greet others.

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

The different parts of this function are as below:



Here, a function `greet()` is created to print **Hello World!**

Note: When writing a function, it is important to understand indentation. Indentation are the spaces at the start of a code line.

In the above code, the `print()` statement is intended to show it's part of the function body, distinguishing the function's definition from its body.

Calling a Function

In the above example, a function is declared with the name `greet()`.

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

The above code when executed will not produce any output. The function is created for executing code inside it. The function has to be called for its execution. The function can be called by just mentioning the function name with opening and closing parenthesis as below.

```
greet()
```

Example 2.2: Let us demonstrate to call a function with the following code

```
def greet():  
    print('Hello World!')  
# call the function  
greet()  
print('Outside function')
```

In the above example, a function named `greet()` is created. The flow of control is depicted as below:



When the function `greet()` is called, the program's control transfers to the function definition. The code inside the function is executed. The control of the program jumps to the next statement after the function call.

Python Function Arguments

Arguments are inputs given to the function. The value of the argument can be passed or mentioned itself in the function.

It is possible to pass different arguments in each call, making the function reusable and dynamic.

In the following example, the value 'Diya' is passed to the argument of the `greet()` function to display the output as 'Hello Diya'

This function is again called with another argument as 'Deepak' to print Hello Deepak as shown in the below code.

```
def greet(name):
    print("Hello", name)
```

```
# pass argument
greet("Diya")
```

```
greet("Deepak")
```

```

File Edit Shell Debug Options Window Help
>>> def greet(name):
...     print('Hello', name)
...
...
>>> # pass argument
>>> greet('Diya')
Hello Diya
>>> greet('Deepak')
Hello Deepak
>>>
Ln: 9 Col: 0

```

Example 2.3: Consider another example of a function to add two numbers.

```
File Edit Shell Debug Options Window Help
>>> # function with two arguments
>>> def add_numbers(num1, num2):
...     sum = num1 + num2
...     print("Sum : ", sum)
...
...
>>> # function call with two values
>>> add_numbers(4,5)
Sum : 9
>>> |
```

Ln: 25 Col: 0

```
# function with two arguments
def add_numbers(num1, num2):
    sum = num1 + num2
    print("Sum : ", sum)
# function call with two values
add_numbers(4,5)
```

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

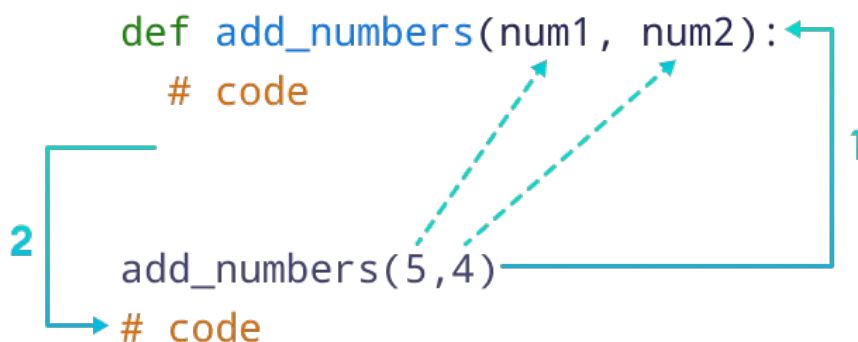


Fig. Python Function with Arguments

Parameters and Arguments

Parameters : Parameters are the variables listed inside the parentheses in the function definition. They act like placeholders for the data the function can accept when we call them.

Think of parameters as the **blueprint** that outlines what kind of information the function expects to receive.

```
def print_age(age): # age is a parameter
```



```
print(age)
```

In this example, the `print_age()` function takes `age` as its input. However, at this stage, the actual value is not specified.

The `age` parameter is just a placeholder waiting for a specific value to be provided when the function is called.

Arguments : Arguments are the actual values that are passed to the function when it is called. Arguments replace the parameters when the function executes.

```
print_age(21) # 21 is an argument
```

Here, during the function call, the argument **21** is passed to the function.

The return Statement

A value of the function is returned using the `return` statement as illustrated in

Example 2.4.

```
# Example 2.3
# function definition
def find_square(num):
    result = num * num
    return result

# function call
square = find_square(3)
print('Square:', square)
square = find_square(5)
print('Square:', square)
```

In Example 2.4, a function named `find_square()` is created. The function accepts a number and returns the square of the number.

```
def find_square(num):
    # code
    return result

square = find_square(3)
# code
```

Note: The `return` statement also denotes the end of function. Any code after `return` is not executed.

The pass Statement

The `pass` statement serves as a placeholder for future code, preventing errors from empty code blocks. It's typically used where code is planned but has yet to be written.

```
def future_function():  
    pass  
  
# this will execute without any action or error  
future_function()
```

Python Library Functions

Python provides some built-in functions that can be directly used in python programs. In such cases we don't need to create the function, but just need to call them.

Some Python library functions are:

`print()` : prints the string inside the quotation marks

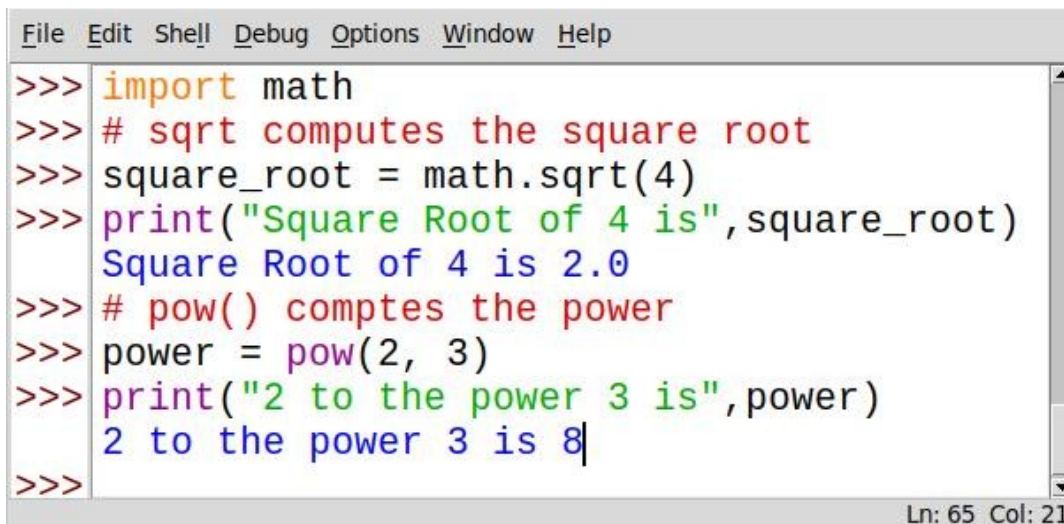
`sqrt()` : returns the square root of a number

`pow()` : returns the power of a number

These library functions are defined inside the module, and to use them, you must include the `math` module inside our program. For example, `sqrt()` is defined inside the `math` module.

Example 2.5 illustrate the use of **Python** library function.

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

A screenshot of a Python IDE window. The menu bar at the top includes File, Edit, Shell, Debug, Options, Window, and Help. The main text area contains the following Python code:

```
>>> import math
>>> # sqrt computes the square root
>>> square_root = math.sqrt(4)
>>> print("Square Root of 4 is",square_root)
Square Root of 4 is 2.0
>>> # pow() computes the power
>>> power = pow(2, 3)
>>> print("2 to the power 3 is",power)
2 to the power 3 is 8
>>>
```

The status bar at the bottom right indicates "Ln: 65 Col: 21".

User Defined Function Vs Standard Library Functions

In Python, functions are divided into two categories: *user-defined functions* and *standard library functions*. These two differ in several ways:

User-Defined Functions – These are the functions created by the user. They're designed for specific tasks to be performed as per our requirement. They're not part of Python's standard toolbox. A user. The user has freedom to tailor them exactly to their needs, adding a personal touch to code.

Standard Library Functions – These are Python's pre-packaged gifts. They come built-in with Python, ready to use. These functions cover a wide range of common tasks such as mathematics, file operations, working with strings, etc. These are tested by the Python community, ensuring efficiency and reliability.

2.2 User defined Functions

There are large number of functions already available in Python under the standard library. We can directly call these functions in our program without defining them. Functions that readily come with Python are called built-in functions. If we use functions written by others in the form of library, it can be termed as library functions.

However, in addition to the standard library functions, we can define our own functions while writing the program. Such functions are called *user defined functions*.

Functions that we define ourselves to do certain specific task are referred as user-defined functions.

Advantages of user-defined functions

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.

2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmers working on large project can divide the workload by making different functions.

Creating User Defined Function

A function definition begins with def (short for define). The syntax for creating a user defined function is as below.

Function Header

```
def <Function name> ([parameter1, parameter2, ...]) :
```

Function Body

```
set of instructions to be executed  
[return <value>]
```

The items enclosed in "[]" are called parameters and they are optional. Hence, a function may or may not have parameters. Also, a function may or may not return a value. Function header always ends with a colon (:). Function names should be unique. Rules for naming identifiers also apply for function naming. The statements outside the function indentation are not considered as part of the function.

Example 2.6 : Consider the problem of finding a maximum of two numbers a and b. Let us take a simple Python program for this problem.

```
# Find Maximum of two numbers  
# Without using function  
a=10  
b=20  
if a>b:  
    max=a  
else:  
    max=b  
    print("Maximum = ",max)
```

```
File Edit Shell Debug Options Window Help
>>> # Find Maximum of two numbers
>>> # Without using function
>>> a=10
>>> b=20
>>> if a>b:
...     max=a
... else:
...     max=b
...     print("Maximum = ",max)
...
...
Maximum = 20
>>>
```

In the above program, a built-in function `print ()` is used to print the value. Another approach to solve the above problem is to divide the program into different blocks of code and keep the block of code in a function which is used to do some specific task. The process of dividing a computer program into separate independent blocks of code or separate sub-problems with different names and specific functionalities is known as modular programming.

Let us solve the above problem using function. The above program is rewritten using user defined functions as shown below.

Example 2.7 :

```
# Find Maximum of two numbers
# By using function
def max(x,y):
    if x>y:
        return x
    else:
        return y

print("Maximum using function=",max(a,b))
```

```
File Edit Shell Debug Options Window Help
>>> # Find Maximum of two numbers
>>> # By using function
>>> def max(x,y):
...     if x>y:
...         return x
...     else:
...         return y
...
...
>>> print("Maximum using function=",max(a,b))
Maximum using function= 20
>>>
```

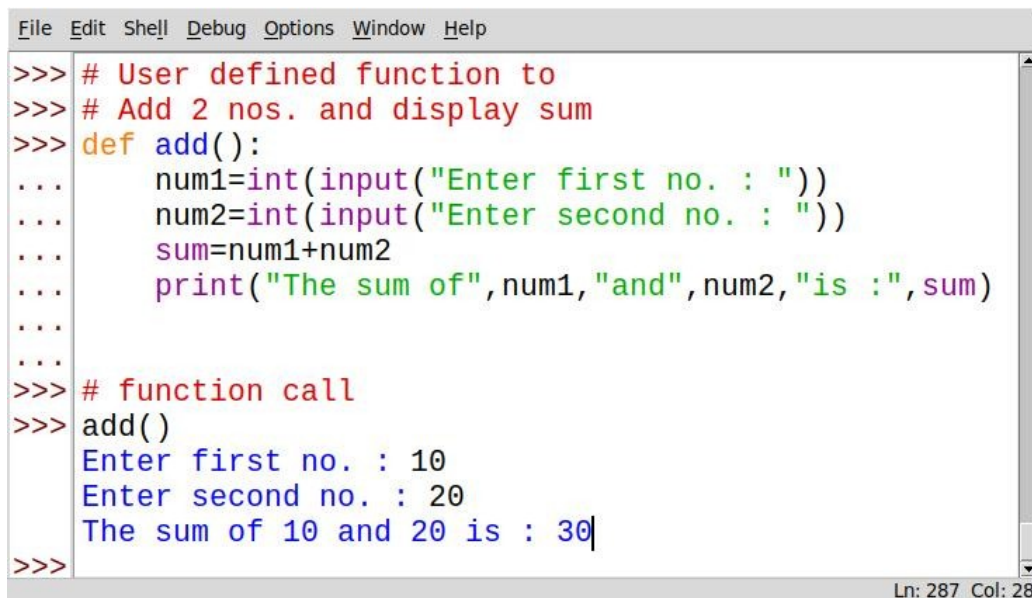
Comparing the above code to find the maximum by using function with that of without using function, it can be seen that the program using function looks

more organized and easier to read. A user defined function *max ()* used to find the maximum of two numbers. The *print () function* is a built-in function of Python which is used to print the value on the console. Thus, we have two types of functions in Python – *User defined functions* and *Built-in functions*.

Example 2.8: Following program illustrates how to write a user defined function to add two numbers and display their sum as a result.

```
# User defined function to add 2 nos. and display sum
def add():
    num1=int(input("Enter first no. : "))
    num2=int(input("Enter second no. : "))
    sum=num1+num2
    print("The sum of",num1,"and",num2,"is :",sum)

# function call
add()
```

A screenshot of a Python IDE window. The menu bar at the top includes File, Edit, Shell, Debug, Options, Window, and Help. The main text area shows the same code as the previous block, with syntax highlighting. The code is executed, and the output is visible: 'Enter first no. : 10', 'Enter second no. : 20', and 'The sum of 10 and 20 is : 30'. The status bar at the bottom right indicates 'Ln: 287 Col: 28'.

In the above program, **add()** is a user-defined function that takes two integer numbers as input, calculates the sum of two numbers and displays it. To execute this function, it is required to call this function by using function name **()**. The function **add()** is being called in the last line of code.

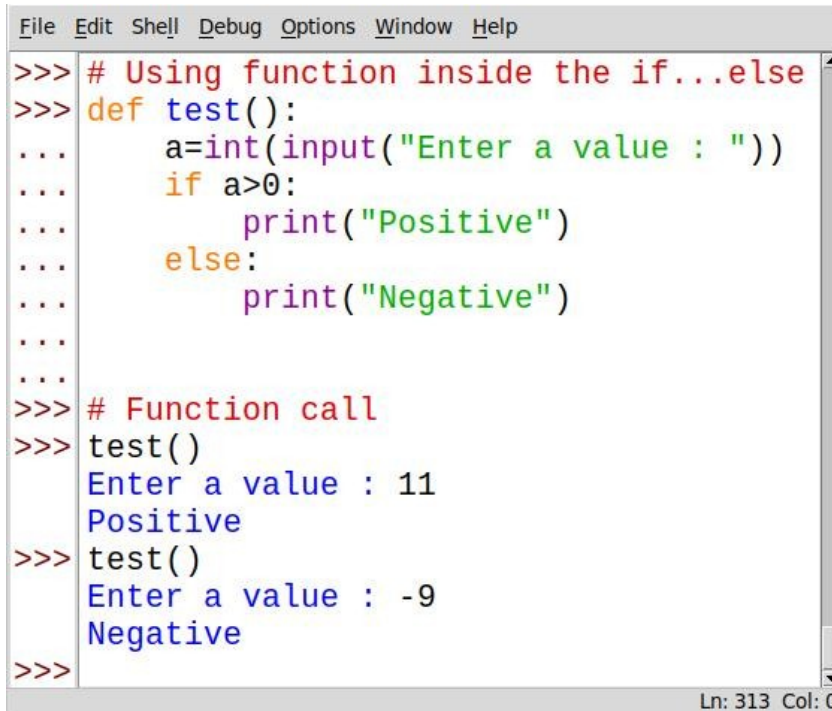
The *def* is the keyword used to define a function and the function name is written following the *def* keyword. After execution, it creates a new function object and assigns it a new name. It can be used inside any control structures like *if else*. **Example 2.8** illustrates this.

```
# Using function inside the if...else
def test():
    a=int(input("Enter a value : "))
```



```
if a>0:
    print("Positive")
else:
    print("Negative")
```

```
# Function call
test()
```

A screenshot of a Python IDE window. The menu bar at the top includes File, Edit, Shell, Debug, Options, Window, and Help. The main text area contains the following code:

```
>>> # Using function inside the if...else
>>> def test():
...     a=int(input("Enter a value : "))
...     if a>0:
...         print("Positive")
...     else:
...         print("Negative")
...
...
>>> # Function call
>>> test()
Enter a value : 11
Positive
>>> test()
Enter a value : -9
Negative
>>>
```

The status bar at the bottom right shows "Ln: 313 Col: 0".

Assignment 2.1

1. Write a program to find the maximum of three numbers using a user-defined function.
2. Write a program to Find Factorial of Number using a user-defined function.
3. Write a program to print the sum of digits of a user entered number using a user-defined function.
4. Write a program to print the day name by reading the day number from the user using a user-defined function.

Example 2.9 : Let's look at an example.

```
def greet(name, message="Hello"):
    print(message, name)

# calling function with both arguments
greet("Students", "Good Morning")

# calling function with only one argument
greet("Diya")
```

```
File Edit Shell Debug Options Window Help
>>> def greet(name, message="Hello"):
...     print(message, name)
...     # calling function with both arguments
...
>>> # calling function with both arguments
>>> greet("Students", "Good Morning")
Good Morning Students
>>> # calling function with only one argument
>>> greet("Diya")
Hello Diya
>>>
Ln: 79 Col: 0
```

2.3 Python Function Arguments

An argument is a value that is accepted by a function. In the above example, the values are accepted from the user within the function itself, but it is also possible for a user defined function to receive values when it is called. An argument is a value passed to the function during the function call which is received in the corresponding parameter defined in the function header.

Example 2.10: Write a python code to add two numbers using a function with arguments.

```
File Edit Shell Debug Options Window Help
>>> # Sum of 2 numbers by using function
>>> # with specified arguments
>>> def add_numbers(a, b):
...     sum = a + b
...     print('Sum using function with arguments',a,b,'=',sum)
...
...
>>> # Function call with arguments 4,5
>>> add_numbers(4,5)
Sum using function with arguments 4 5 = 9
>>> # Function call with arguments -4,5
>>> add_numbers(-4,5)
Sum using function with arguments -4 5 = 1
>>>
Ln: 145 Col: 35
```

```
# Sum of 2 numbers by using function
# with specified arguments
def add_numbers(a, b):
    sum = a + b
    print('Sum using function with arguments',a,b,'=',sum)

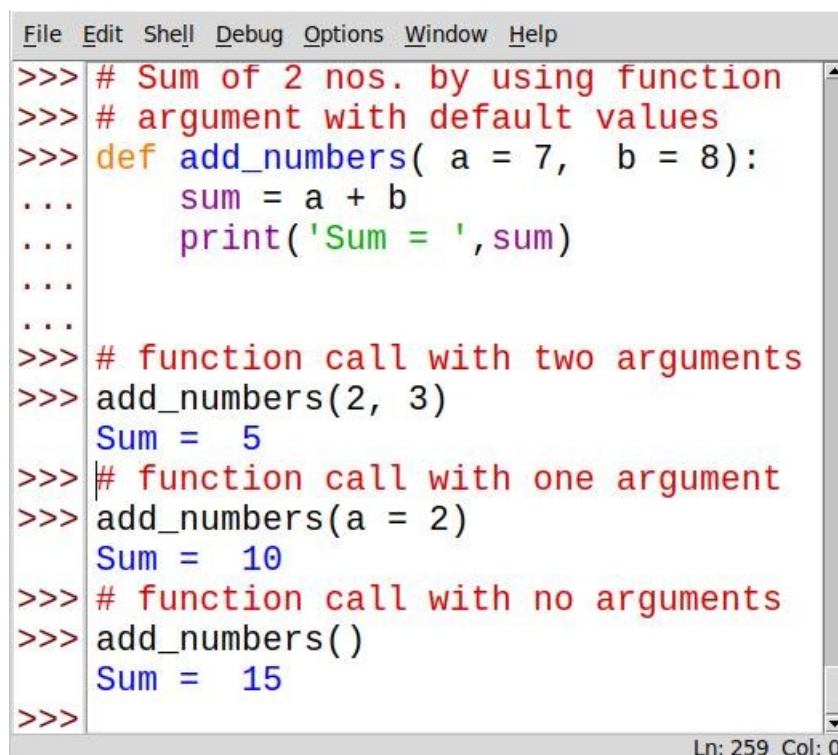
# Function call with arguments 4,5
add_numbers(4,5)
```

In the above example, the function `add_numbers()` takes two parameters: `a` and `b`. The function, `add_numbers(4, 5)` specifies that parameters `a` and `b` will get values **4** and **5** respectively. The function, `add_numbers(4, 5)` specifies that parameters `a` and `b` will get values **-4** and **5** respectively.

Function Argument with Default Values

Python allows assigning a default value to the parameter. Default arguments are used when no explicit values are passed to these parameters during a function call. The `=` operator is used to provide default values.

Example 2.11: For example, the above code can be modified as below.



```
File Edit Shell Debug Options Window Help
>>> # Sum of 2 nos. by using function
>>> # argument with default values
>>> def add_numbers( a = 7, b = 8):
...     sum = a + b
...     print('Sum = ',sum)
...
...
>>> # function call with two arguments
>>> add_numbers(2, 3)
Sum = 5
>>> # function call with one argument
>>> add_numbers(a = 2)
Sum = 10
>>> # function call with no arguments
>>> add_numbers()
Sum = 15
>>>
```

Ln: 259 Col: 0

```
# Sum of 2 nos. by using function
# argument with default values
def add_numbers( a = 7, b = 8):
    sum = a + b
    print('Sum = ',sum)

# function call with two arguments
add_numbers(2, 3)

# function call with one argument
add_numbers(a = 2)

# function call with no arguments
add_numbers()
```

In this example the default values **7** and **8** are provided for parameters a and b respectively. Let us see how this program works for three conditions

1. add_number(2, 3) : Both values are passed during the function call. Hence, these values are used instead of the default values.

2. add_number(2) : Only one value is passed during the function call. So, according to the positional argument **2** is assigned to argument a, and the default value is used for parameter b.

3. add_number() : No value is passed during the function call. Hence, default value is used for both parameters a and b.

Example 2.12: Consider another example to convert the denominator of the division into proper fraction using a user defined function.

```
# Program to convert denominator of division into
# proper fraction using user defined function
def mixedFraction(num, deno = 1):
    remainder = num % deno
    if remainder != 0:
        quotient = int(num/deno)
        print("The mixed fraction =
", quotient, '(', remainder, '/', deno, ')')
    else:
        print("It evaluates to whole number")
# Function ends
num = int(input("Enter the numerator : "))
deno = int(input("Enter the denominator : "))
print("The number is :", num, '/', deno)
if num > deno:                # Check for proper fraction
    mixedFraction(num, deno) # Function call
else:
    print("It is a proper fraction")
```

```
File Edit Format Run Options Window Help
# proper fraction using user defined funtion
def mixedFraction(num, deno = 1):
    remainder = num % deno
    if remainder != 0:
        quotient = int(num/deno)
        print("The mixed fraction = ", quotient, '(', remainder, '/', deno, ')')
    else:
        print("It evaluates to whole number")
# Function ends
num = int(input("Enter the nummerator : "))
deno = int(input("Enter the denominator : "))
print("The number is :", num, '/', deno)
if num > deno:
    mixedFraction(num, deno) # Check for proper fraction
else:
    print("It is a proper fraction")
Ln: 13 Col: 38
```

When you run the program for various inputs, the output is as below.

```
File Edit Shell Debug Options Window Help
>>> Enter the denominator : 3
>>> The number is : 20 / 3
>>> The mixed fraction = 6 ( 2 / 3 )
>>>
>>> ===== RESTART
>>> Enter the nummerator : 20
>>> Enter the denominator : 2
>>> The number is : 20 / 2
>>> It evaluates to whole number
>>>
>>> ===== RESTART
>>> Enter the nummerator : 6
>>> Enter the denominator : 20
>>> The number is : 6 / 20
>>> It is a proper fraction
>>>
```

In the above program, the denominator entered is 3, which is passed to the parameter "*deno*" so the default value of the argument *deno* is overwritten. Let us consider the following function call: `mixedFraction(9)`. Here, *num* will be assigned 25 and *deno* will use the default value 1.

A function argument can also be an expression, such as

`mixedFraction(num+5, deno+5)`

In such a case, the argument is evaluated before calling the function so that a valid value can be assigned to the parameter. The parameters should be in the same order as that of the arguments.

The default parameters must be the trailing parameters in the function header that means if any parameter is having default value then all the other parameters to its right must also have default values. For example,

```
def mixedFraction(num,deno = 1)
def mixedFraction(num = 2,deno = 1)
```

Let us consider few more function definition headers:

```
def calcInterest(principal = 1000, rate, time = 5):
```

Above header is incorrect as default must be the last #parameter. So, the correct function header should be as follows.

```
def calcInterest(rate, principal = 1000, time = 5):
```

One more point is important to note that a function header cannot have expressions. Therefore, following function headers will give error:

```
def mixedFraction(num+5,deno):
def mixedFraction(num+5,deno+5):
```

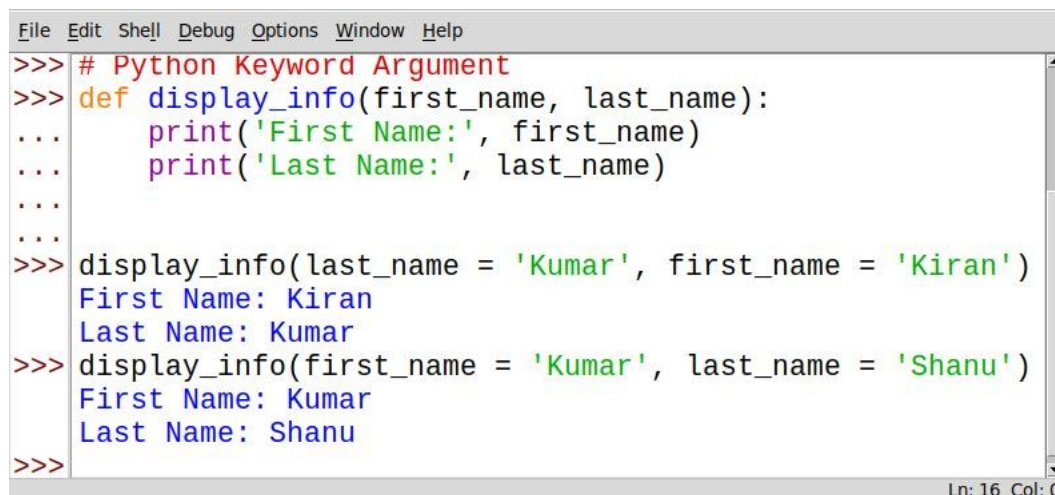
String as Parameters

In the above program only numeric types of arguments are passed while calling a function. However, it may also require passing string values as an argument, as illustrated in **Example 2.13**.

Example 2.13:

```
def display_info(first_name, last_name):
    print('First Name:', first_name)
    print('Last Name:', last_name)

display_info(last_name = 'Kumar', first_name = 'Kiran')
display_info(first_name = 'Kumar', last_name = 'Shanu')
```



The screenshot shows a Python IDE window with a menu bar (File, Edit, Shell, Debug, Options, Window, Help) and a code editor. The code in the editor is as follows:

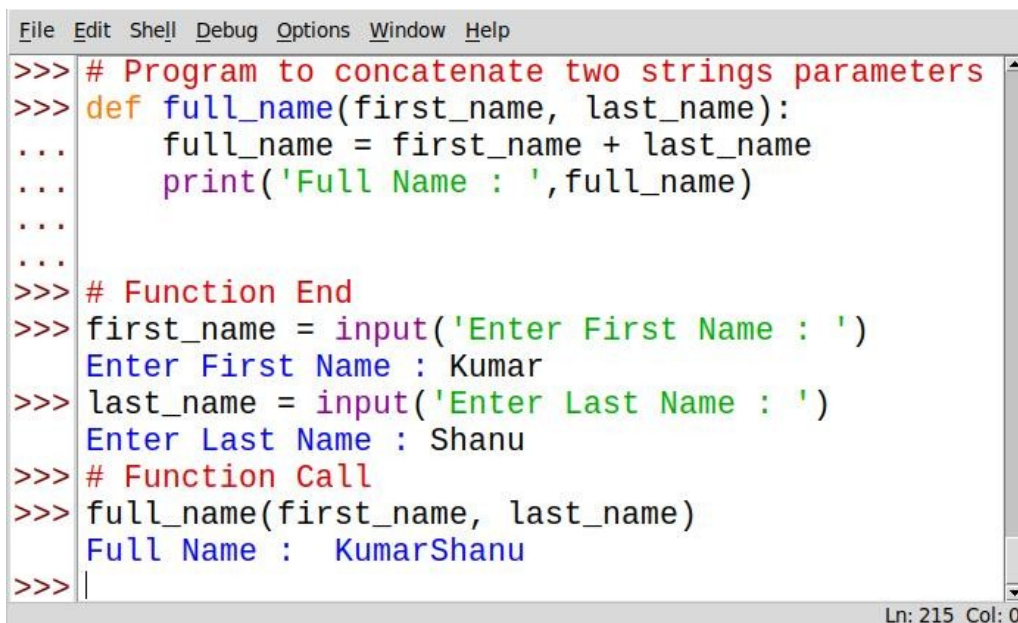
```
>>> # Python Keyword Argument
>>> def display_info(first_name, last_name):
...     print('First Name:', first_name)
...     print('Last Name:', last_name)
...
...
>>> display_info(last_name = 'Kumar', first_name = 'Kiran')
First Name: Kiran
Last Name: Kumar
>>> display_info(first_name = 'Kumar', last_name = 'Shanu')
First Name: Kumar
Last Name: Shanu
>>>
```

The output of the program is displayed in the shell area, showing the first and last names for both calls to the function. The status bar at the bottom right indicates 'Ln: 16 Col: 0'.

Here, the names to arguments are assigned during the function call. Hence, `first_name` in the function call is assigned to `first_name` in the function definition. Similarly, `last_name` in the function call is assigned to `last_name` in the function definition. In such cases, the position of arguments doesn't matter.

Example 2.14: Consider another example to display the full name by taking the input first name and last name using concatenation of two strings.

```
# Program to concatenate two strings parameters
# Function Start
def full_name(first_name, last_name):
    full_name = first_name + last_name
    print('Full Name : ',full_name)
# Function End
first_name = input('Enter First Name : ')
Enter First Name : Kumar
last_name = input('Enter Last Name : ')
Enter Last Name : Shanu
# Function Call
full_name(first_name, last_name)
```



```
File Edit Shell Debug Options Window Help
>>> # Program to concatenate two strings parameters
>>> def full_name(first_name, last_name):
...     full_name = first_name + last_name
...     print('Full Name : ',full_name)
...
...
>>> # Function End
>>> first_name = input('Enter First Name : ')
Enter First Name : Kumar
>>> last_name = input('Enter Last Name : ')
Enter Last Name : Shanu
>>> # Function Call
>>> full_name(first_name, last_name)
Full Name :  KumarShanu
>>> |
Ln: 215 Col: 0
```

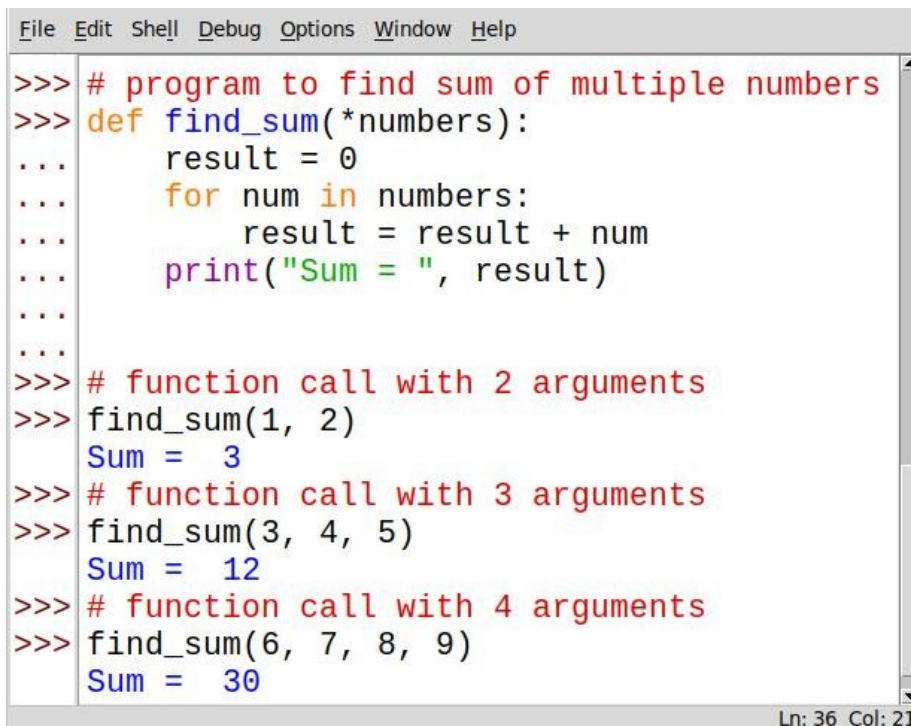
Python Function with Arbitrary Arguments

Sometimes, the number of arguments that will be passed into a function may not be known in advance. To handle such situations, arbitrary arguments can be used in Python. Arbitrary arguments allow us to pass a varying number of values during a function call. The asterisk (*) can be used before the parameter name to denote such an argument.

Example 2.15:

```
# program to find sum of multiple numbers
def find_sum(*numbers):
    result = 0
    for num in numbers:
        result = result + num
    print("Sum = ", result)

# function call with 2 arguments
find_sum(1, 2)
Sum = 3
# function call with 3 arguments
find_sum(3, 4, 5)
Sum = 12
# function call with 4 arguments
find_sum(6, 7, 8, 9)
Sum = 30
```



```
File Edit Shell Debug Options Window Help
>>> # program to find sum of multiple numbers
>>> def find_sum(*numbers):
...     result = 0
...     for num in numbers:
...         result = result + num
...     print("Sum = ", result)
...
...
>>> # function call with 2 arguments
>>> find_sum(1, 2)
Sum = 3
>>> # function call with 3 arguments
>>> find_sum(3, 4, 5)
Sum = 12
>>> # function call with 4 arguments
>>> find_sum(6, 7, 8, 9)
Sum = 30
Ln: 36 Col: 21
```

In Example 2.15, the function **find_sum()** accepts arbitrary arguments. Hence it is possible to call the same function with different arguments. After getting multiple values, numbers behave as an array so we are able to use the for loop to access each value.

2.4 Scope of a variable

A variable defined inside a function cannot be accessed outside it. Every variable has a well-defined accessibility. The part of the program where a variable is accessible can be defined as the scope of that variable.

In Python, we can declare variables in three different scopes: *local scope*, *global*, and *nonlocal scope*.

A variable scope specifies the region where we can access a variable. For example,

```
def add_numbers():  
    sum = 5 + 4
```

Here, the *sum* variable is created inside the function, so it can only be accessed within it (local scope). This type of variable is called a *local* variable.

Based on the scope, Python variables are classified into three types:

1. Local Variables
2. Global Variables
3. Nonlocal Variables

Local Variables

When we declare variables inside a function, these variables will have a local scope (within the function). It can be accessed only in the function or a block where it is defined. It exists only till the function executes.

These types of variables are called local variables. For example,

Example 2.16:

```
def greet():  
  
    # local variable  
    message = 'Hello'  
  
    print('Local', message)  
  
greet()  
  
# try to access message variable  
# outside greet() function  
print(message)
```

Output

Local Hello

NameError: name 'message' is not defined

Here, the *message* variable is local to the *greet()* function, so it can only be accessed within the function.

That's why we get an error when we try to access it outside the *greet()* function.

To fix this issue, we can make the variable named *message* global.

Global Variables

In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function. Any change made to the global variable will impact all the functions in the program where that variable can be accessed.

Example 2.17: illustrates how a global variable is created in Python.

```
# declare global variable
message = 'Hello'

def greet():
    # declare local variable
    print('Local', message)

greet()
print('Global', message)
```

Output

```
Local Hello
Global Hello
```



```
File Edit Shell Debug Options Window Help
>>> # declare global variable
>>> message = 'Hello'
>>> def greet():
...     # declare local variable
...     print('Local', message)
...
>>> greet()
Local Hello
>>> print('Global', message)
Global Hello
>>> |
Ln: 50 Col: 0
```

This time we can access the *message* variable from outside of the `greet()` function. This is because we have created the *message* variable as the global variable.

```
# declare global variable
message = 'Hello'
```

Now, *message* will be accessible from any scope (region) of the program.

Nonlocal Variables

In Python, the `nonlocal` keyword is used within nested functions to indicate that a variable is not local to the inner function, but rather belongs to an enclosing function's scope.

This allows you to modify a variable from the outer function within the nested function, while still keeping it distinct from global variables.

Example 2.18:

```
# outside function
def outer():
    message = 'local'

    # nested function
    def inner():

        # declare nonlocal variable
        nonlocal message

        message = 'nonlocal'
        print("inner:", message)

    inner()
    print("outer:", message)

outer()
```

Output

```
inner: nonlocal
outer: nonlocal
```

In the above example, there is a nested `inner()` function. The `inner()` function is defined in the scope of another function `outer()`.

We have used the `nonlocal` keyword to modify the `message` variable from the outer function within the nested function.

Note : If we change the value of a nonlocal variable, the changes appear in the local variable.

Global Keyword

In Python, the global keyword allows to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.

Access and Modify Python Global Variable

First let's try to access a global variable from the inside of a function,

Example 2.19:

```
c = 1 # global variable
def add():
    print(c)
add()

# Output: 1
```

Here, we can see that a global variable is accessed from the inside of a function.

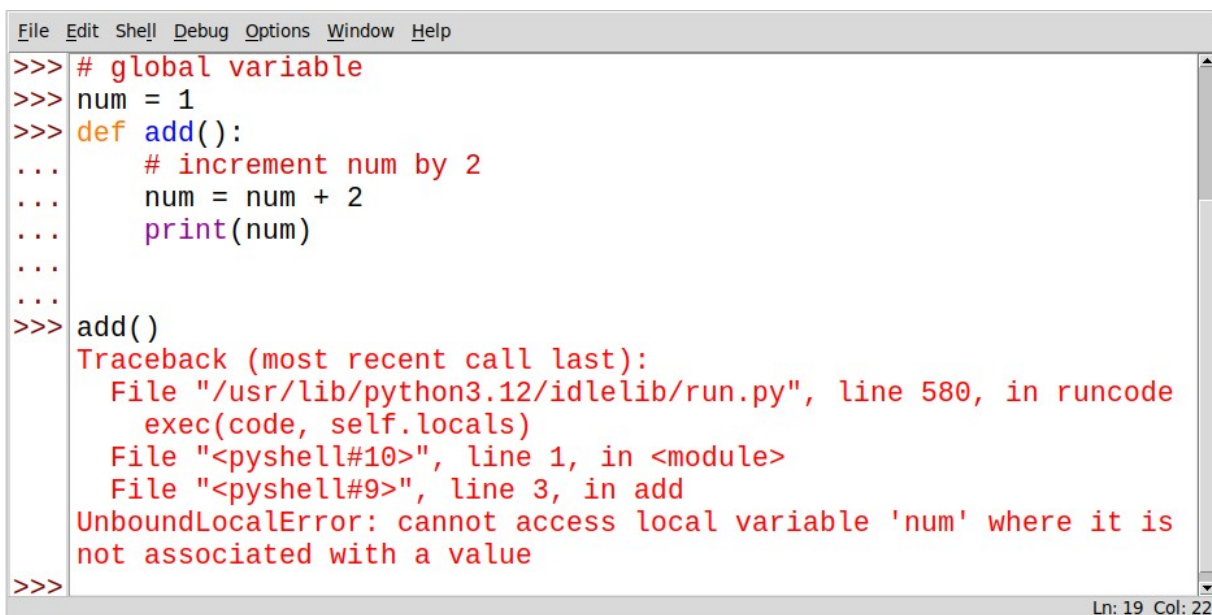
However, if we try to modify the global variable from inside a function as:

Example 2.20:

```
# global variable
c = 1
def add():
    # increment c by 2
    c = c + 2
    print(c)
add()
```

Output

UnboundLocalError: local variable 'c' referenced before assignment

A screenshot of a Python IDE window. The menu bar at the top includes File, Edit, Shell, Debug, Options, Window, and Help. The main text area shows a Python script with the following code:

```
>>> # global variable
>>> num = 1
>>> def add():
...     # increment num by 2
...     num = num + 2
...     print(num)
...
>>> add()
```

 Below the code, a red traceback message is displayed:

```
Traceback (most recent call last):
  File "/usr/lib/python3.12/idlelib/run.py", line 580, in runcode
    exec(code, self.locals)
  File "<pyshell#10>", line 1, in <module>
  File "<pyshell#9>", line 3, in add
UnboundLocalError: cannot access local variable 'num' where it is
not associated with a value
>>>
```

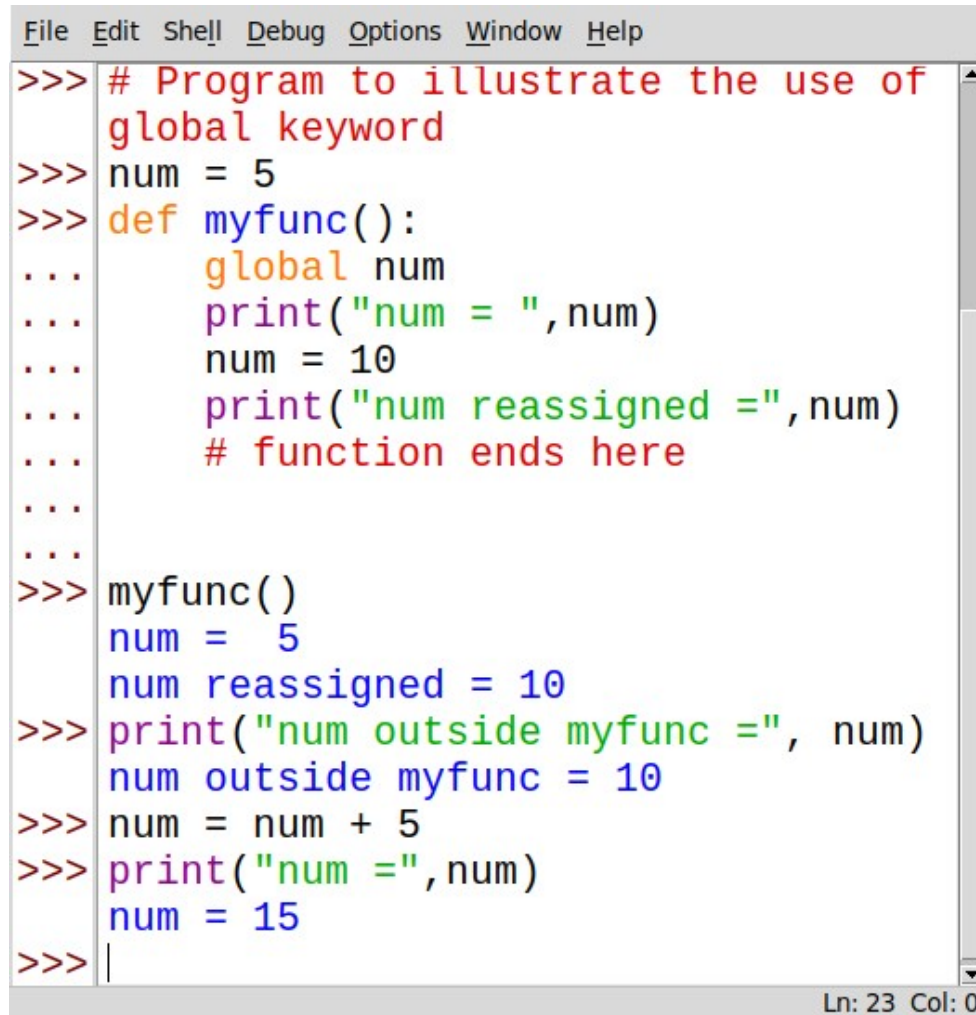
 The status bar at the bottom right indicates "Ln: 19 Col: 22".

This is because we can only access the global variable but cannot modify it from inside the function.

The solution for this is to use the global keyword.

Example 2.21 illustrates, changing global variable from inside a function using global.

Example 2.21:



```
File Edit Shell Debug Options Window Help
>>> # Program to illustrate the use of
>>> global keyword
>>> num = 5
>>> def myfunc():
...     global num
...     print("num = ", num)
...     num = 10
...     print("num reassigned =", num)
...     # function ends here
...
...
>>> myfunc()
num = 5
num reassigned = 10
>>> print("num outside myfunc =", num)
num outside myfunc = 10
>>> num = num + 5
>>> print("num =", num)
num = 15
>>> |
Ln: 23 Col: 0
```

In the above program's output, Global variable num is accessed as the ambiguity is resolved by prefixing the keyword global to it.

Rules of global Keyword

The basic rules for global keyword in Python are:

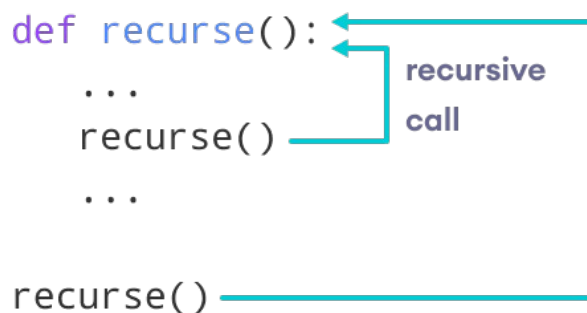
- When we create a variable inside a function, it is local by default.
- When we define a variable outside of a function, it is global by default. You don't have to use the global keyword.
- We use the global keyword to modify (write to) a global variable inside a function.
- Use of the global keyword outside a function has no effect.

2.5 Recursion

Recursion is the process of defining something in terms of itself. A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

Recursive Function

In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions. The following image shows the working of a recursive function called recurse.



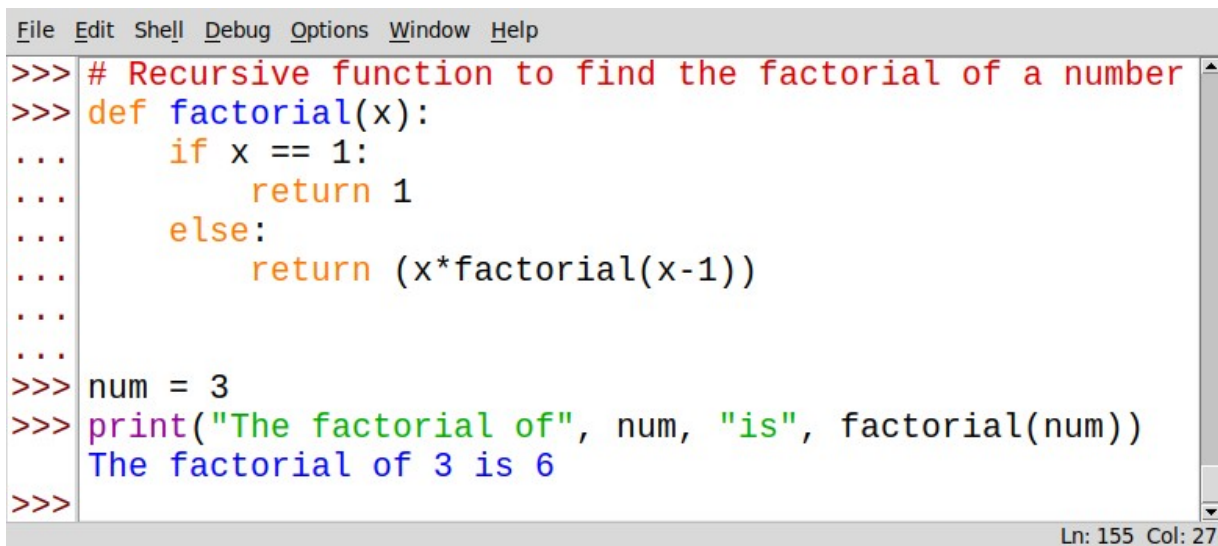
Following is an example of a recursive function to find the factorial of an integer. Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is $1*2*3*4*5*6 = 720$.

Example 2.22: Example of a recursive function

```
def factorial(x):  
    # This is a recursive function to  
    # find the factorial of an integer  
    if x == 1:  
        return 1  
    else:  
        return (x * factorial(x-1))  
num = 3  
print("The factorial of", num, "is", factorial(num))
```

Output

The factorial of 3 is 6

A screenshot of a Python IDE window. The menu bar at the top includes File, Edit, Shell, Debug, Options, Window, and Help. The main text area contains the following Python code:

```
>>> # Recursive function to find the factorial of a number
>>> def factorial(x):
...     if x == 1:
...         return 1
...     else:
...         return (x*factorial(x-1))
...
...
>>> num = 3
>>> print("The factorial of", num, "is", factorial(num))
The factorial of 3 is 6
>>>
```

The status bar at the bottom right indicates "Ln: 155 Col: 27".

In the above example, **factorial()** is a recursive function as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

factorial(3)	# 1st call with 3
3 * factorial(2)	# 2nd call with 2
3 * 2 * factorial(1)	# 3rd call with 1
3 * 2 * 1	# return from 3rd call as number=1
3 * 2	# return from 2nd call
6	# return from 1st call

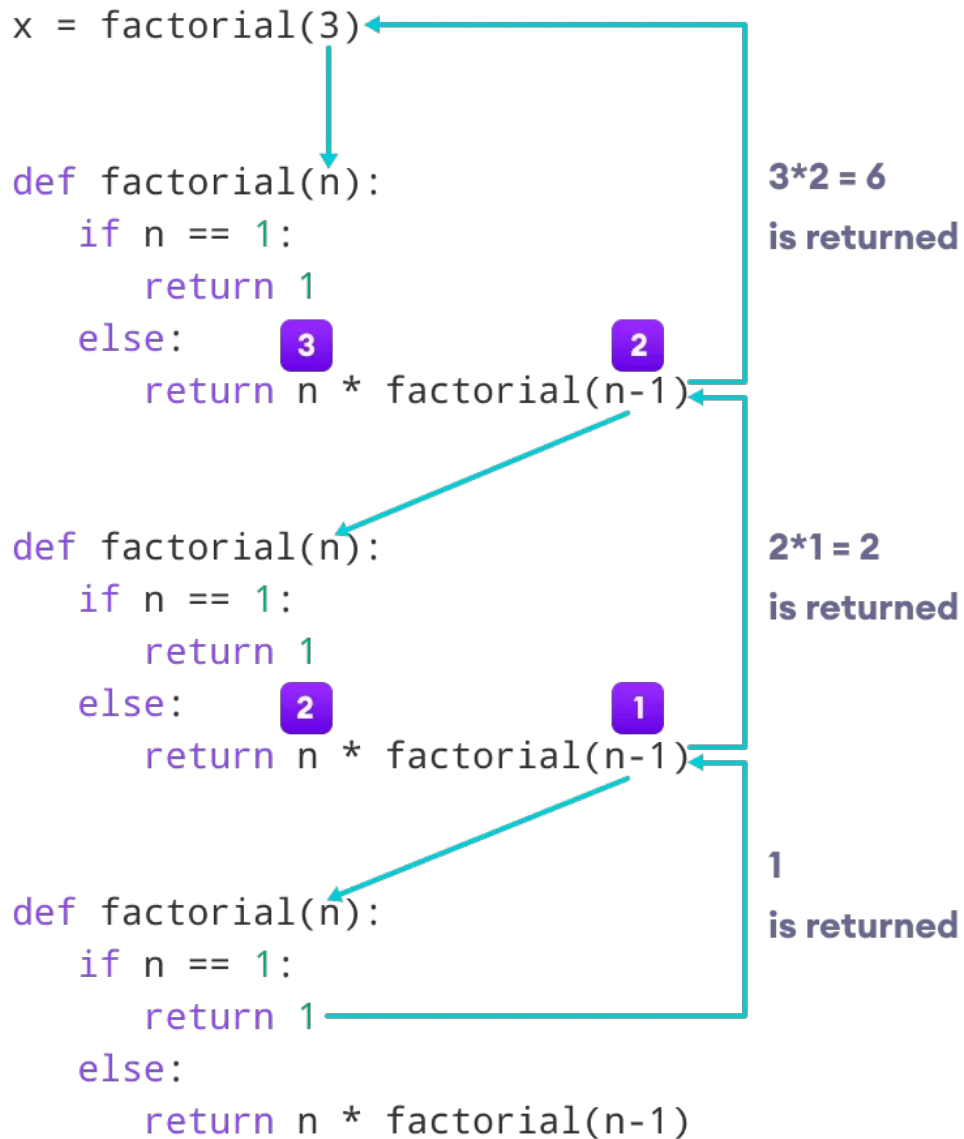
Let's look at an image that shows a step-by-step process of what is going on:

Our recursion ends when the number reduces to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.

By default, the maximum depth of recursion is 1000. If the limit is crossed, it results in `RecursionError`. Let's look at one such condition.



```
File Edit Shell Debug Options Window Help
>>> def recursor():
...     recursor()
...
...
>>> recursor()
Traceback (most recent call last):
  File "<pyshell#27>", line 2, in recursor
  File "<pyshell#27>", line 2, in recursor
  File "<pyshell#27>", line 2, in recursor
[Previous line repeated 997 more times]
RecursionError: maximum recursion depth exceeded
>>> |
```

Ln: 47 Col: 0

Advantages of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

Example 2.23: Write a program to calculate the factorial of a number using recursion.

```
# Python program to find the factorial of a number provided by the user.

# To take input from the user
num = int(input("Enter a number: "))

factorial = 1

# check if the number is negative, positive or zero
if num < 0:
    print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
    print("The factorial of 0 is 1")
else:
    for i in range(1,num + 1):
        factorial = factorial*i
    print("The factorial of",num,"is",factorial)
```

```
File Edit Format Run Options Window Help
# Python program to find the factorial of a number provided by the user.

# To take input from the user
num = int(input("Enter a number: "))

factorial = 1

# check if the number is negative, positive or zero
if num < 0:
    print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
    print("The factorial of 0 is 1")
else:
    for i in range(1,num + 1):
        factorial = factorial*i
    print("The factorial of",num,"is",factorial)
```

Ln: 12 Col: 6

You can run this program for different input as shown below.

```
File Edit Shell Debug Options Window Help
Enter a number: 0
The factorial of 0 is 1
>>>
=====
Enter a number: 1
The factorial of 1 is 1
>>>
=====
Enter a number: 2
The factorial of 2 is 2
>>>
=====
Enter a number: 3
The factorial of 3 is 6
>>>
=====
Enter a number: 5
The factorial of 5 is 120
>>>
```

CHECK YOUR PROGRESS

A. Multiple Choice Questions

1. A named group of instructions that accomplish a specific task when it is invoked is called a (a) string (b) control (c) tuple (d) Function
2. Which keyword is used for function? (a) fun (b) define (c) def (d) function
3. What are the two main types of functions? (a) Custom function (b) Built-in function & User defined function (c) User function (d) System function

4. Which of the following is the use of id () function in Python? (a) Id returns the identity of the object (b) Every object doesn't have a unique id (c) All of the mentioned (d) None of the mentioned
5. Which of the following refers to mathematical function? (a) sqrt (b) rhombus (c) add (d) minus
6. Which of the following is not true for Recursive functions in Python. (a) Recursive functions make the code look clean and elegant (b) A complex task can be broken down into simpler sub-problems using recursion (c) Recursive functions take up a lot of memory and time. (d) Recursive functions are easy to debug. (d)

B. State whether True or False

1. Use of function is one of the means to achieve modularity and reusability.
2. Function header always ends with a semicolon (;).
3. Lambda function contains return statements.
4. Lambda is an anonymous function in Python.
5. Module is a grouping of functions.
6. Once we import a module, we can directly use all the functions of that module.
7. To use the function when imported using "from statement" we do not need to precede it with the module name.
8. It is not possible to create your own function, besides the available modules in the Python standard library.
9. randrange () is a function of a random module.
10. fmod () is a function of the statistics module.

C. Fill in the Blanks

1. The process of dividing a computer program into separate independent blocks of code or separate sub-problems with different names and specific functionalities is known as _____ programming.
2. A function defined to achieve some tasks as per the programmer's requirement is called a _____ function.
3. An _____ is a value passed to the function during the function call which is received in the corresponding parameter defined in the function header.
4. The _____ statement returns the values from the function.
- 5.

6. Sequence generation is easier with _____ than using some nested iteration (recursion)

D. Programming Questions

1. Identify the errors if any in the following code.

(a) `def create (text, freq):`

```
    for i in range (1, freq):  
        print text  
    create (5)    #function call
```

(b) `from math import sqrt,ceil`

```
    def calc ():  
        print cos (0)  
    calc () #function call
```

(c) `mynum = 9 def add9():`

```
    mynum = mynum + 9  
    print mynum  
    add9() #function call
```

(d) `def findValue(val) = 1.1, val2, val3):`

```
    final = (val2 + val3)/ val1  
    print(final)  
    findvalue ()    #function call
```

(e) `def greet ():`

```
    return ("Good morning")  
    greet () = message    #function call
```

2. Write a program to check the divisibility of a number by 7 that is passed as a parameter to the user defined function.
3. Write a program that uses a user defined function that accepts name and gender (as M for Male, F for Female) and prefixes Mr./Ms. on the basis of the gender.
4. Write a Program that uses two user defined functions to convert temperatures to and from Celsius, Fahrenheit and Fahrenheit to Celsius.
5. Write a program that has a user defined function to accept the coefficients of a quadratic equation in variables and calculates its determinant. For example: if the coefficients are stored in the variables a, b, c then calculate the determinant as b^2-4ac . Write the appropriate condition to check determinants on positive, zero and negative and output appropriate results.
6. ABC School has allotted unique token IDs from (1 to 600) to all the parents for facilitating a lucky draw on the day of their annual day

function. The winner would receive a special prize. Write a program using Python that helps to automate the task. (Hint: use random module)

7. Write a program that implements a user defined function that accepts Principal Amount, Rate, Time, Number of Times the interest is compounded to calculate and displays compound interest. (Hint: $CI = P(1 + \frac{R}{N})^{NT}$)
8. Write a program that has a user defined function to accept 2 numbers as parameters, if number 1 is less than number 2 then numbers are swapped and returned, i.e., number 2 is returned in place of number 1 and number 1 is reformed in place of number 2, otherwise the same order is returned.
9. Write a program that contains user defined functions to calculate area, perimeter or surface area whichever is applicable for various shapes like square, rectangle, triangle, circle and cylinder. The user defined functions should accept the values for calculation as parameters and the calculated value should be returned. Import the module and use the appropriate functions.
10. Write a program that creates a GK quiz consisting of any five questions of your choice. The questions should be displayed randomly. Create a user defined function score () to calculate the score of the quiz and another user defined function remark (score value) that accepts the final score to display remarks as follows:

Marks	Remarks
5	Outstanding
4	Excellent
3	Good
2	Read more to score more
1	Needs to take interest
0	General knowledge will always help you. Take it seriously.

Appendix 1

Table 12.2 Commonly used functions in math module

Function Syntax	Arguments	Returns	Example Output
math.ceil(x)	x may be an integer or floating-point number	ceiling value of x	<pre>>>> math.ceil(-9.7) -9 >>> math.ceil (9.7) 10 >>> math.ceil(9) 9</pre>

<code>math.floor(x)</code>	x may be an integer or floating-point number	floor value of x	<pre>>>> math.floor(-4.5) -5 >>> math.floor(4.5) 4 >>> math.floor(4) 4</pre>
<code>math.fabs(x)</code>	x may be an integer or floating-point number	absolute value of x	<pre>>>> math.fabs(6.7) 6.7 >>> math.fabs(-6.7) 6.7 >>> math.fabs(-4) 4.0</pre>
<code>math.factorial(x)</code>	x is a positive integer	factorial of x	<pre>>>> math.factorial(5) 120</pre>
<code>math.fmod(x,y)</code>	x and y may be an integer or floating-point number	$x \% y$ with sign of x	<pre>>>> math.fmod(4,4.9) 4.0 >>> math.fmod(4.9,4.9) 0.0 >>> math.fmod(- 4.9,2.5) -2.4 >>> math.fmod(4.9,- 4.9) 0.0</pre>
<code>math.gcd(x,y)</code>	x, y are positive integers	gcd (greatest common divisor) of x and y	<pre>>>> math.gcd(10,2) 2</pre>
<code>math.pow(x,y)</code>	x, y may be an integer or floating-point number	x^y (x raised to the power y)	<pre>>>> math.pow(3,2) 9.0 >>> math.pow(4,2.5) 32.0 >>> math.pow(6.5,2) 42.25 >>> math.pow(5.5,3.2) 233.97</pre>
<code>math.sqrt(x)</code>	x may be a positive integer or floating-point number	square root of x	<pre>>>> math.sqrt(144) 12.0 >>> math.sqrt(.64) 0.8</pre>
<code>math.sin(x)</code>	x may be an integer or floating-point number in radians	sine of x in radians	<pre>>>> math.sin(0) 0 >>> math.sin(6) -0.279</pre>

Table 12.3 Commonly used functions in random module

Function Syntax	Argument	Return	Example Output
random. random()	No argument (void)	Random Real Number (float) in the range 0.0 to 1.0	>>> random.random() 0.65333522
random. randrange(x,y)	x and y are positive integers signifying the start and stop value	Random integer between x and y	>>> random.randrange(2,7) 2
random. randrange(y)	y is a positive integer signifying the stop value	Random integer between 0 and y	>>> random.randrange(5) 4

Table 12.4 Some of the function available through statistics module

Function Syntax	Argument	Return	Example Output
statistics.mean(x)	x is a numeric sequence	arithmetic mean	>>> statistics.mean([11,24,32,45,51]) 32.6
statistics.median(x)	x is a numeric sequence	median (middle value) of x	>>> statistics.median([11,24,32,45,51]) 32
statistics.mode(x)	x is a sequence	mode (the most repeated value)	>>> statistics.mode([11,24,11,45,11]) 11 >>> statistics.mode(["red","blue","red"]) 'red'

Module 2. Data Science

Data science is an evolving field which combines math, computer science, and domain expertise to tackle real-world challenges in a variety of fields. The study of data helps us derive useful insight for business decision making. Data Science is all about using tools, techniques, and creativity to uncover insights hidden within data. Data science is about using data to gain knowledge and make better decisions by collecting, cleaning, analyzing, and interpreting information to uncover patterns and insights. Data Science processes the raw data and solve business problems and even make prediction about the future trend or requirement.

Data science is about using data to gain knowledge and make better decisions by collecting, cleaning, analyzing, and interpreting information to uncover patterns and insights. Data science empowers the industries to make smarter, faster, and more informed decisions.

Python is a programming language widely used by Data Scientists. Python has in-built mathematical libraries and functions, making it easier to calculate mathematical problems and to perform data analysis. Numpy is a mathematical library in Python with powerful N-dimensional array object, linear algebra, Fourier transform, etc.

In this unit we will discuss the advanced tool **NumPy** that is commonly used for data science. This tool is commonly used for numerical analysis of large amount of data. Various mathematical and logical operations can be performed by using NumPy.

Session 1. Introduction to NumPy

In Python, the lists data structure serves the purpose of arrays, but they are slow to process. NumPy aims to provide an array object that is much faster than traditional Python lists. NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently. The elements of a NumPy array must all be of the same type, whereas the elements of a Python list can be of completely different types.

NumPy stands for *Numerical Python*. NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object. It provides functions for fast mathematical computation on arrays and matrices. NumPy objects are primarily used to create arrays or matrices that can be applied to Deep Learning or Machine Learning models. Pandas is used for creating heterogenous, two-dimensional data objects, NumPy makes N-dimensional homogeneous objects. Pandas functions return results in the form of NumPy array.

Why NumPy?

There are several reasons to use NumPy for data analysis as given below.

1. In NumPy we can create arrays for large amounts of data.
2. Various basic linear algebra operations, statistical operation, Fourier transform, sorting, searching, shape manipulation and random simulation operations can be performed by using NumPy.
3. NumPy package is the ndarray object. It means we can have n dimensional arrays of homogeneous data elements.
4. NumPy arrays have a fixed size. When the size is changed, then it will delete the original and a new array is created.
5. Consider a simple example of multiplying two arrays. In python we can write a code for the multiplication as given below.

```
c = [ ]
for i in range(len(a)):
    c.append(a[i]*b[i])
```

The above code when executed will produce the correct result, but will require time when both the arrays **a** and **b** will contain millions of numbers. The loop in python is inefficient when compared with the C programming language. When we use NumPy then the above multiplication can be achieved simply by writing `c=a*b`

Observe that NumPy gives the advantage of efficient programming.

6. NumPy is very fast because its code is more concise and easier to read. It contains fewer lines of codes therefore there are fewer bugs.
7. NumPy can be easily integrated with other scientific libraries of Python such as Pandas, Matplotlib, SciPy, and TensorFlow.

Installation of NumPy

To use NumPy, it should be installed. The best way to install NumPy on your system is by using a pre-build package for your operating system. Refer to the link provided below for installation of NumPy.

Installation of NumPy in Windows

NumPy can be installed on Windows by using the Anaconda toolbox or by using pip. If you don't have anaconda installed then install anaconda..

Installation of anaconda in Windows

To install anaconda in Windows computer, first download anaconda from the link <https://www.anaconda.com/download/success>

1. Locate the executable file of anaconda and download it for installation.
2. Click on the executable file **Anaconda3-2024.10-1-Windows-x86_64.exe** that is already downloaded on your computer system.
3. Click on this file and follow the simple instructions appearing on your screen.
4. Observe that the anaconda toolbox will get installed on C drive.
5. Now open the anaconda navigator as shown below.

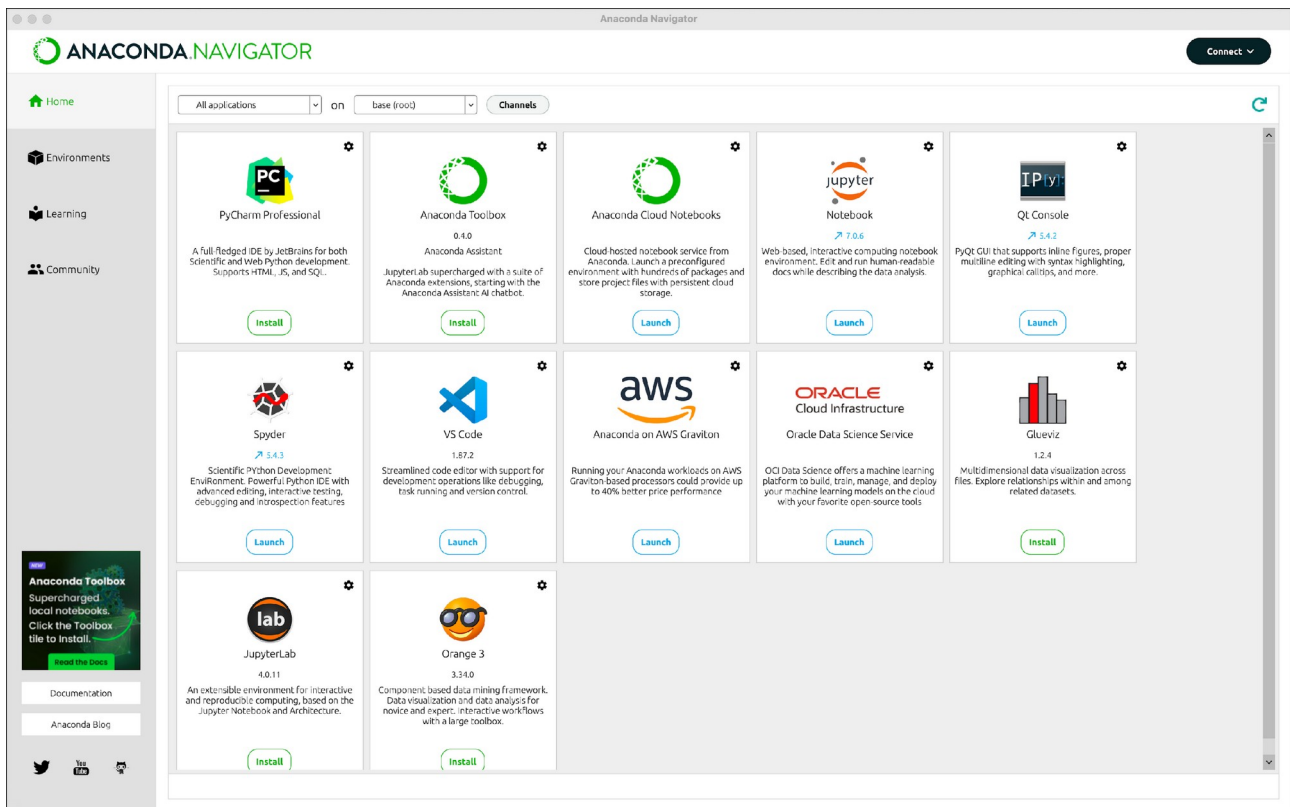


Fig. 1.1 : Anaconda navigator

From this navigator select the Anaconda prompt. Click on the launch button of Anaconda prompt as shown in Figure below.

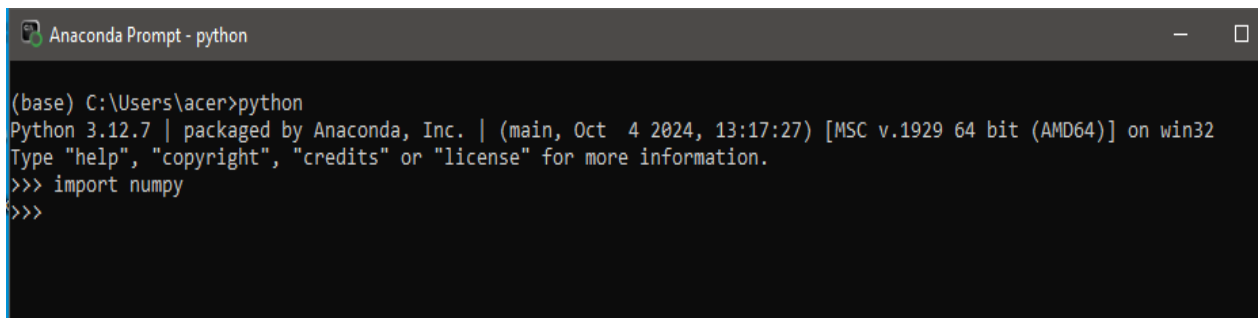


Fig. 1.2 : Anaconda prompt

On clicking the launch button, the anaconda prompt screen appears as shown in Figure 1.2.

On the prompt type

Python

Observe that the python version as below

Python 3.12.7 packaged by Anaconda.....

It means that Python is already installed by anaconda and now the python prompt >>> appears on the screen.

Now to check whether the NumPy is already installed on Python, type the command on the python prompt.

```
>>> import numpy
```

If no error message is displayed then it means that the anaconda toolbox has already installed the NumPy tool and it is ready for use.

Now you can execute NumPy code on this prompt.

Whenever you want to exit anaconda prompt type Ctrl+z and press enter

```
>>> ^Z
```

Now you can close your anaconda prompt.

Anaconda navigator can be closed by clicking on the close (x) window of the navigator.

Similarly, to check whether the Panda is already installed on Python, type the command on the python prompt.

```
>>> import panda
```

If no error message is displayed then it means that the anaconda toolbox has already installed the Panda tool and it is ready for use.

Installation of NumPy in Linux

Step 1. It is necessary to install Python before installing NumPy. If it is not installed, then first install Python by using the following link.

[Link to install python](#)

Step 2. To install the NumPy in Ubuntu Linux first open the terminal and check whether Python is installed or not on your computer by using the following command.

```
deepak@ds:~$ python3 --version
```

```
Python 3.12.7
```

```
deepak@ds:~$
```

This shows that Python 3.12.7 version is installed on your computer.

Step 3. Now check if the pip is installed or not by issuing the command pip on the \$ prompt.

```
deepak@ds:~$ pip
```

You will get something at the last \$ prompt. This shows that pip is installed.

Step 4. After that, write python3 on \$ prompt, and the python console will open. Then issue the command

```
>>> import numpy as nm
```

```
adam@adam-VirtualBox:~$ python3
Python 3.12.3 (main, Apr 10 2024, 05:33:47) [GCC 13.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as nm
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'numpy'
>>>
[1]+  Stopped                  python3
adam@adam-VirtualBox:~$ clear
```

Step 5. If numpy is not installed then it will show the error

ModuleNotFoundError: No module named 'numpy'. This shows that numpy is not installed. Then clear the terminal and proceed to install numpy.

Step 6. Now to install the numpy issue the following command and password.

```
deepak@ds:~$ sudo apt install python3-numpy
[sudo] password for deepak:
```

Step 7. It will download as well as install the numpy as shown in the following screenshot. Make sure that the internet must be connected during this process. When you receive the \$ prompt, numpy is installed.

```
adam@adam-VirtualBox:~$ sudo apt install python3-numpy
[sudo] password for adam:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Suggested packages:
  gfortran python3-pytest
The following NEW packages will be installed:
  python3-numpy
0 upgraded, 1 newly installed, 0 to remove and 39 not upgraded.
Need to get 4,437 kB of archives.
After this operation, 23.9 MB of additional disk space will be used.
Get:1 http://in.archive.ubuntu.com/ubuntu noble/main amd64 python3-numpy amd64 1:1.26.4+ds-6ubuntu1 [4,437 kB]
Fetched 4,437 kB in 5s (922 kB/s)
Selecting previously unselected package python3-numpy.
(Reading database ... 152202 files and directories currently installed.)
Preparing to unpack .../python3-numpy_1%3a1.26.4+ds-6ubuntu1_amd64.deb ...
Unpacking python3-numpy (1:1.26.4+ds-6ubuntu1) ...
Setting up python3-numpy (1:1.26.4+ds-6ubuntu1) ...
Processing triggers for man-db (2.12.0-4build2) ...
adam@adam-VirtualBox:~$
```

Step 8. You can check whether numpy is installed or not with the following.

```
deepak@ds:~$ python3
Python 3.12.3 (main, Nov 6 2024, 18:32:19) [GCC 13.2.0] on linux
```

Type "help", "In Python, the lists data structure serve the purpose of arrays, but they are slow to process. NumPy aims to provide an array object that is much faster than traditional Python lists. NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently. The elements of a NumPy array must all be of the same type, whereas the elements of a Python list can be of completely different types.

NumPy stands for Numerical Python is a Python library used for working with arrays. It provides functions for fast mathematical computation on arrays and matrices. NumPy objects are primarily used to create arrays or matrices that can be applied to Deep Learning or Machine Learning models. Pandas is used for creating heterogenous, two-dimensional data objects, NumPy makes N-dimensional homogeneous objects. Pandas functions return results in the form of NumPy array. copyright", "credits" or "license" for more information.

```
>>> import numpy as nm
```

```
>>> This shows that numpy is installed and you can do coding related to numpy.
```

Installation of anaconda in Ubuntu Linux

<https://docs.vultr.com/how-to-install-anaconda-on-ubuntu-24-04#install-anaconda-on-ubuntu-2404>

Anaconda is not available in the default Ubuntu package repositories. Follow the steps below to install the package by downloading a script from the Anaconda repository.

Step 1. Visit the Anaconda archives directory to check and download the latest installation script.

```
$ wget https://repo.anaconda.com/archive/Anaconda3-2024.10-1-Linux-x86_64.sh
```

Step 2. Run the script using Bash.

```
$ bash Anaconda3-2024.10-1-Linux-x86_64.sh
```

Press Enter when prompted to review the Anaconda license agreement.

```
Welcome to Anaconda3 2024.10-1
```

```
In order to continue the installation process, please review the
license
agreement.
```

```
Please, press ENTER to continue
```

```
>>>
```

Step 3. Press Space to browse through the license agreement.

Step 4. Enter yes and press Enter to accept the license agreement.

Version 4.0 | Last Modified: March 31, 2024 | ANACONDA TOS

Do you accept the license terms? [yes|no]

>>> yes

Step 5. Verify the default installation path, typically anaconda3 in your user home directory and press Enter to install Anaconda.

Anaconda3 will now be installed into this location:

/vultr/anaconda3

- Press ENTER to confirm the location
- Press CTRL-C to abort the installation
- Or specify a different location below

[/vultr/anaconda3] >>>

Step 6. Enter yes and press Enter to update your shell environment and initialize Conda.

Do you wish to update your shell profile to automatically initialize conda?

This will activate conda on startup and change the command prompt when activated.

If you'd prefer that conda's base environment not be activated on startup,

run the following command when conda is activated:

```
conda config --set auto_activate_base false
```

You can undo this by running `conda init --reverse \$SHELL`? [yes|no]

[no] >>> yes

Step 7. Reload the .bashrc shell configuration to apply the Anaconda changes to your environment.

```
$ source ~/.bashrc
```

4. View the installed Conda version.

```
$ conda -version
```

Output:

```
conda 24.5.0
```

Step 8. List all packages in the default Conda environment.

```
$ conda list
```

Output:

#	Name	Version	Build	Channel
	_anaconda_depends	2024.06	py312_mkl_2	
	_libgcc_mutex	0.1	main	
	_openmp_mutex	5.1	1_gnu	
	abseil-cpp	20211102.0	hd4dd3e8_0	
	aiobotocore	2.12.3	py312h06a4308_0	
	aiohttp	3.9.5	py312h5eee18b_0	
	aioitertools	0.7.1	pyhd3eb1b0_0	
	aiosignal	1.2.0	pyhd3eb1b0_0	
	alabaster	0.7.16	py312h06a4308_0	
	altair	5.0.1	py312h06a4308_0	
	anaconda-anon-usage	0.4.4	py312hfc0e8ea_100	
	anaconda-catalogs	0.2.0	py312h06a4308_1	
	anaconda-client	1.12.3	py312h06a4308_0	
	anaconda-cloud-auth	0.5.1	py312h06a4308_0	
	anaconda-navigator	2.6.0	py312h06a4308_0	
	anaconda-project	0.11.1	py312h06a4308_0	
	annotated-types	0.6.0	py312h06a4308_0	
			

Manage Conda Environments

A Conda environment contains specific dependencies, packages, and Python versions that match your project needs. The isolated environment is different from your global system environment. Follow the steps below to create and manage Conda environments.

1. Create a new myenv Conda environment with a specific Python version, such as 3.8.

```
$ conda create --name myenv python=3.8
```

Enter Y when prompted to install all necessary dependency packages in the new environment.

2. Activate the new Conda environment

```
$ conda activate myenv
```

Verify that your shell prompt changes to the new environment.

```
(myenv) vultr@Server:~$
```

3. List all outdated packages in your environment.

```
$ conda update -all
```

Enter Y when prompted to install new packages.

Step 9. Create a test environment by cloning the existing myenv environment.

```
$ conda create --name test --clone myenv
```

Step 10. Install a new package, such as requests in the environment.

```
$ conda install requests
```

Step 11. To install packages from different sources like conda-forge, run conda install -c conda-forge package_name and replace package_name with the name of the package that you want to install.

Step 12. Clear all unused packages and caches from your environment.

```
$ conda clean --all
```

Step 13. Export the Conda environment to a file like environment.yml.

```
$ conda env export > environment.yml
```

Step 14. Import the environment to reinstall all necessary packages and dependencies.

```
$ conda env create -f environment.yml
```

Step 15. Use conda run to execute commands in a specific environment without activating it. For example, run the script.py file in the myenv environment.

```
$ conda run -n myenv python -c "print('Hello, World!')"
```

Creating basic arrays using NumPy

NumPy is a main object for the creation of homogeneous multidimensional arrays. An array is like a list of elements. Homogeneous means all the elements are similar in nature. So a Numpy array consists of a table of elements. All the elements in the NumPy are indexed by a tuple of non negative integers. The number of dimensions of the array are called as axes in the numpy.

Figure... shows a one dimensional (1D), two dimensional (2D), three dimensional (3D) array.

Fig. 1.3 : 1D array, 2D array and 3D array

Such arrays can be created in numpy using the array class called as ndarray.

The important attributes of ndarray objects are as given below.

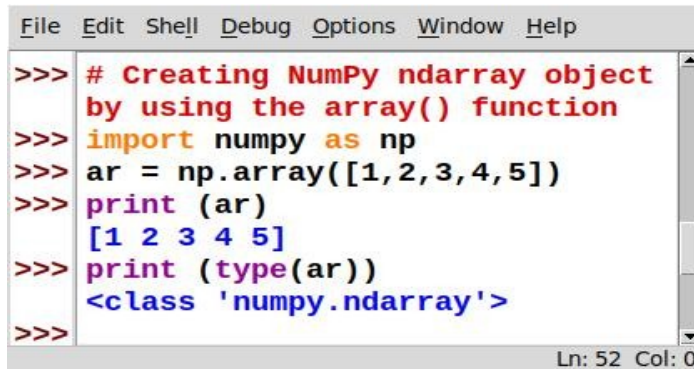
Creating Arrays in NumPy

There are 6 general mechanisms for creating arrays.

1. Conversion from other Python structures such as lists and tuples.
2. Intrinsic NumPy array creation functions (e.g. arrange, ones, zeros, etc.).
3. Replicating, joining, or mutating existing arrays.

4. Reading arrays from disk, either from standard or custom formats.
5. Creating arrays from raw bytes through the use of strings or buffers.
6. Use of special library functions such as random.

NumPy is used to work with arrays. The array object in NumPy is called ndarray. NumPy ndarray objects can be created by using the `array()` function as illustrated in **Example 1.1**.

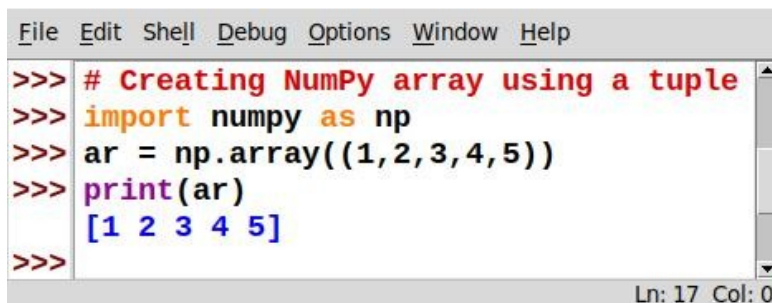


```
>>> # Creating NumPy ndarray object
>>> by using the array() function
>>> import numpy as np
>>> ar = np.array([1,2,3,4,5])
>>> print (ar)
[1 2 3 4 5]
>>> print (type(ar))
<class 'numpy.ndarray'>
>>>
```

Ln: 52 Col: 0

`type()` is a built-in Python function that displays the type of the object passed to it. Like in above code it shows that `arr` is `numpy.ndarray` type.

To create an ndarray, you can pass a list, tuple or any array-like object into the `array()` method, and it will be converted into an ndarray, as illustrated in the **Example 1.2**.



```
>>> # Creating NumPy array using a tuple
>>> import numpy as np
>>> ar = np.array((1,2,3,4,5))
>>> print(ar)
[1 2 3 4 5]
>>>
```

Ln: 17 Col: 0

Dimensions in Arrays

There are two forms of NumPy arrays – one dimensional array, known as *vectors*, and multidimensional arrays, known as *matrices*. NumPy has a whole sub module dedicated towards matrix operations called `numpy.mat`

Example 1.3, illustrates how to create 1D, 2D and 3D arrays.

```

File Edit Shell Debug Options Window Help
>>> # Creating 1-Dimensional array
>>> import numpy as np
>>> ar1 = np.array([1, 2, 3, 4, 5])
>>> print("1 Dimensional array created as : \n", ar1)
1 Dimensional array created as :
[1 2 3 4 5]
>>> # Creating 2-Dimensional array
>>> ar2 = np.array([[1, 2, 3], [4, 5, 6]])
>>> print("2 Dimensional array created as : \n", ar2)
2 Dimensional array created as :
[[1 2 3]
 [4 5 6]]
>>> # Creating 3-Dimensional array
>>> ar3 = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
>>> print("3 Dimensional array created as : \n", ar3)
3 Dimensional array created as :
[[[1 2 3]
  [4 5 6]]

 [[1 2 3]
  [4 5 6]]]
>>>

```

Ln: 24 Col: 0

```

File Edit Shell Debug Options Window Help
>>> # Checking the dimension of array
>>> import numpy as np
>>> a = np.array(25)
>>> print ("The array is :\n", a)
The array is :
25
>>> print ("The dimesion of array is :\n", a.ndim)
The dimesion of array is :
0
>>> b = np.array([1, 2, 3, 4, 5])
>>> print ("The array is :\n", b)
The array is :
[1 2 3 4 5]
>>> print ("The dimesion of array is :\n", b.ndim)
The dimesion of array is :
1
>>> c = np.array([[1, 2, 3], [4, 5, 6]])
>>> print ("The array is :\n", c)
The array is :
[[1 2 3]
 [4 5 6]]
>>> print ("The dimesion of array is :\n", c.ndim)
The dimesion of array is :
2
>>> d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
>>> print ("The array is :\n", d)
The array is :
[[[1 2 3]
  [4 5 6]]

 [[1 2 3]
  [4 5 6]]]
>>> print ("The dimesion of array is :\n", d.ndim)
The dimesion of array is :
3
>>>

```

Ln: 41 Col: 0

Checking Dimensions of Array

It is possible to check the dimension of the array. NumPy Arrays provides the `ndim` attribute that returns an integer value indicating the number of dimensions the array. **Example 1.4**, illustrates to check the dimension of array.

NumPy Arrays Vs Python Lists

Although NumPy arrays also hold elements like Python List, yet Numpy arrays are different data structures from Python lists. The key differences are as follows.

NumPy Array	Python List
It is required to install and import Numpy Library to access Numpy Arrays.	It is a built-in function of python available in the core library of python.
Once a NumPy array is created, it is not possible to change its size. It is required to create a new array or overwrite the existing one.	It is possible to append/insert values in List.
NumPy array contain elements of same type (homogeneous)	List contain elements of different type (heterogeneous)
Element wise operation is possible in the NumPy array.	Element wise operation is not possible on the list.
An equivalent NumPy array occupies much less space than a Python list.	List occupies much more space than an array.
It is faster as compared to lists.	It is slow as compared to NumPy Array.
NumPy array supports vectorized operations.	List does not support vectorized operations.

Example 1.5: It is possible to convert a python list into a NumPy array with the help of NumPy `array()` function. **Example 1.5** illustrates this.


```
File Edit Shell Debug Options Window Help
>>> # Converting List into Array using NumPy
>>> array() function
>>> import numpy as np
>>> L=[11,22,33,44,55]
>>> ar=np.array(L)
>>> print("Array values are : ",ar)
>>> Array values are : [11 22 33 44 55]
>>>
Ln: 9 Col: 0
```

In this example, L = [12,23,34,22] is listed. The statement aa=np.array(L) will convert this list into an array and store it into “aa”.

Example 1.6 : Following python code illustrates that NumPy array can perform vector addition but List cannot perform it.

```
File Edit Shell Debug Options Window Help
>>> # Program to illustrate the vector addition on NumPy Array but not possible on List
>>> import numpy as np
>>> L=[1,2,3,4]
>>> ar=np.array(L)
>>> print('Adding value to array adds the value to each element of array', '\n', ar+5)
>>> Adding value to array adds the value to each element of array
>>> [6 7 8 9]
>>> print('Adding value to list generates Error Message as follows', '\n', L+5)
>>> Traceback (most recent call last):
>>>   File "/usr/lib/python3.10/idlelib/run.py", line 578, in runcode
>>>       exec(code, self.locals)
>>>   File "<pyshell#5>", line 1, in <module>
>>> TypeError: can only concatenate list (not "int") to list
>>>
Ln: 16 Col: 0
```

Accessing Array Elements

It is possible to access an array element by referring to its index number. The indexes in NumPy arrays start with 0, meaning the first element has index 0, and the second has index 1 and so on. **Example 1.7** illustrates to access the array elements.

```
File Edit Shell Debug Options Window Help
>>> # Example to illustrate to access elements of NumPy array
>>> import numpy as np
>>> arr = np.array([1, 2, 3, 4])
>>> print('First element of array is : ', arr[0])
>>> First element of array is : 1
>>> print('Second element of array is : ', arr[1])
>>> Second element of array is : 2
>>> print('Third element of array is : ', arr[2])
>>> Third element of array is : 3
>>>
Ln: 12 Col: 0
```

A table is a 2-D array with rows and columns, where the dimension represents the row and the index represents the column. To access elements from 2-D arrays, a comma is used to separate integers representing the dimension and the index of the element.

Similarly, to access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

Example 1.8 illustrates to access the elements of a 2-Dim and 3-Dim array.

```
File Edit Shell Debug Options Window Help
>>> # Example to illustrate to acces the elements of 2-Dim and 3-Dim array
>>> import numpy as np
>>> ar2 = np.array([[1,2,3,4,5], [6,7,8,9,10]])
>>> print('Third element of first row is :', ar2[0,2])
Third element of first row is : 3
>>> print('Fifth element of second row is :', ar2[1,4])
Fifth element of second row is : 10
>>> ar3 = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
>>> print('Third element of second array is :', ar3[0,1,2])
Third element of second array is : 6
>>> print('Second element of second array is :', ar3[0,1,1])
Second element of second array is : 5
>>>
```

Ln: 15 Col: 0

Data Types in NumPy

There are a large number of data types that are supported by NumPy. For example, in NumPy we can have integer type, floating type, complex type, Boolean type, string type, object type, date and time type.

Integer Type

This type is used for storing whole numbers with varying bit sizes and signs.

Data Type	Description	Range	Example
int8	8-bit signed integer	-128 to 127	>>> np.array([127, -128], dtype=np.int8) array([127, -128], dtype=int8)
uint8	8-bit unsigned integer	0 to 255	>>> np.array([0,255], dtype=np.uint8) array([0, 255], dtype=uint8)
int16	16-bit signed integer	-32,768 to 32,767	>>> np.array([-32768, 32767], dtype=np.int16) array([-32768, 32767], dtype=int16)

Data Type	Description	Range	Example
uint16	16-bit unsigned integer	0 to 65,535	>>> np.array([0, 65535], dtype=np.uint16) array([0, 65535], dtype=uint16)
int32	32-bit signed integer	-2,147,483,648 to 2,147,483,647	>>> np.array([1, -2147483648], dtype=np.int32) array([1, -2147483648], dtype=int32)
uint32	32-bit unsigned integer	0 to 4,294,967,295	>>> np.array([0, 4294967285], dtype=np.uint32) array([0, 4294967285], dtype=uint32)
int64	64-bit signed integer	Very large range	>>> np.array([1, -9223372036854775808], dtype=np.int64) array([1, 9223372036854775808], dtype=int64)
uint64	64-bit unsigned integer	Very large range	>>> np.array([1, -9223372036854775808], dtype=np.uint64) array([1, 9223372036854775808], dtype=uint64)

2. Floating-Point Types

This type is used for numbers with decimal points.

Data Type	Description	Range	Example
float16	Half precision	Approx. $\pm 65,504$	>>> np.array([1.5, -2.5], dtype=np.float16) array([1.5, -2.5], dtype=float16)
float32	Single precision	Approx. $\pm 3.4 \times 10^{38}$	>>> np.array([1.234, -5.678], dtype=np.float32) array([1.234, -5.678], dtype=float32)
float64	Double precision	Very large range	>>> np.array([1.123456789, -9.87654321], dtype=np.float64) array([1.12345679, -9.87654321])

3. Complex Types

This type is for complex numbers with real and imaginary parts.

Data Type	Description	Example
complex 64	Complex number with two 32-bit floats	<pre>>>> np.array([1+2j, 3-4j], dtype=np.complex64) array([1.+2.j, 3.-4.j], dtype=complex64)</pre>
complex 128	Complex number with two 64-bit floats	<pre>>>> np.array([1.5+2.5j, -3.5+4.5j], dtype=np.complex128) array([1.5+2.5j, -3.5+4.5j])</pre>

4. Boolean Type

This type is used for True or False Boolean Type. It is used for logical operations.

Data Type	Description	Example
bool_	Boolean type (True/False)	<pre>>>> np.array([True, False], dtype=np.bool_) array([True, False])</pre>

5. String Types

This type is used for fixed-size strings.

Data Type	Description	Example
string_	Fixed-size ASCII string	<pre>>>> np.array(['a', 'bc'], dtype=np.string_) array([b'a', b'bc'], dtype='<S2')</pre>
unicode_	Fixed-size Unicode string	<pre>>>> np.array(['hello', 'world'], dtype=np.unicode_) array(['hello', 'world'], dtype='<U5')</pre>

6. Object Type

Used for arbitrary Python objects.

Data Type	Description	Example
object_	Python object type	<pre>>>> np.array([1, 'string', 3.14], dtype=np.object_) array([1, 'string', 3.14], dtype=object)</pre>

7. Datetime and Timedelta Types

This is used for handling dates, times, and durations.

Data Type	Description	Example
datetime64	Date and time representation	<pre>>>> np.array(['2024-12-17'], dtype=np.datetime64) array(['2024-12-17'], dtype='datetime64[D]')</pre>
timedelta64	Duration or time difference	<pre>>>> np.array([10, 20], dtype='timedelta64[D]') array([10, 20], dtype='timedelta64[D]')</pre>

Examples demonstrating Data Types

By default Python has string, integer, float, boolean data types. NumPy has some extra data types, and refers to data types with one character, like i for integers, u for unsigned integers. The NumPy array object has a property called dtype that returns the data type of the array.

Example 1.9 illustrates to create NumPy array of different data types.

```
File Edit Shell Debug Options Window Help
>>> # Example to check the data type of array
>>> import numpy as np
>>> ar1 = np.array([1,2,3,4])
>>> print ('Data type of array is : ', ar1.dtype)
Data type of array is : int64
>>> ar2 = np.array(['a', 'b', 'c'])
>>> print ('Data type of array is : ', ar2.dtype)
Data type of array is : <U1
>>> ar3 = np.array([1.1,2.2,3.3,4.4])
>>> print ('Data type of array is : ', ar3.dtype)
Data type of array is : float64
>>>
```

Creating Arrays with a Defined Data Type

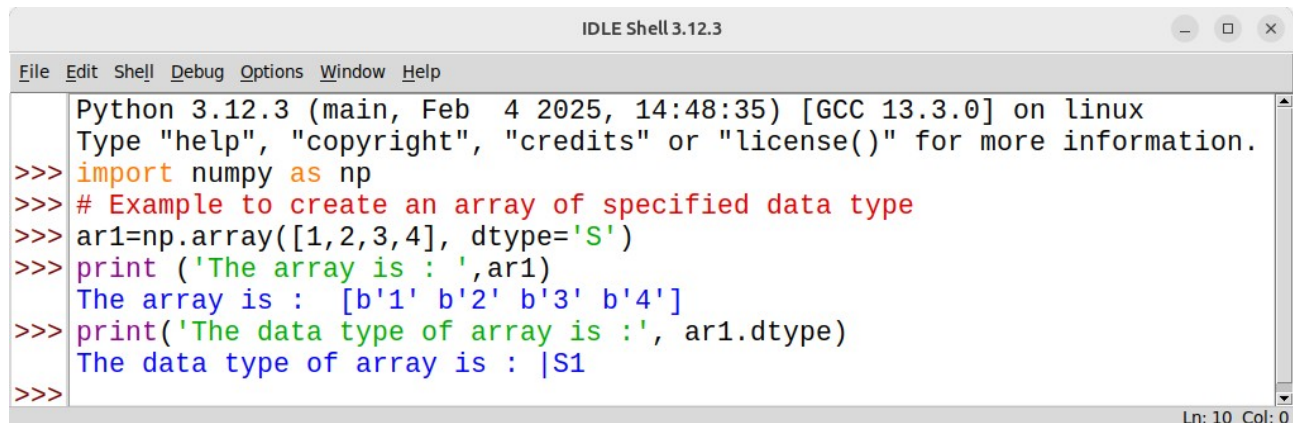
The array() function is used to create arrays that can take an optional argument as dtype that allows to define the expected data type of the array elements. If the data element is not of specified data type, it generates error.

Example 1.10 illustrates how to create an array with specified data types.

```
# Example to create an array of specified data type
>>> ar1=np.array([1,2,3,4], dtype='S')
>>> print ('The array is : ',ar1)
```



```
The array is : [b'1' b'2' b'3' b'4']
>>> print('The data type of array is :', ar1.dtype)
The data type of array is : |S1
```

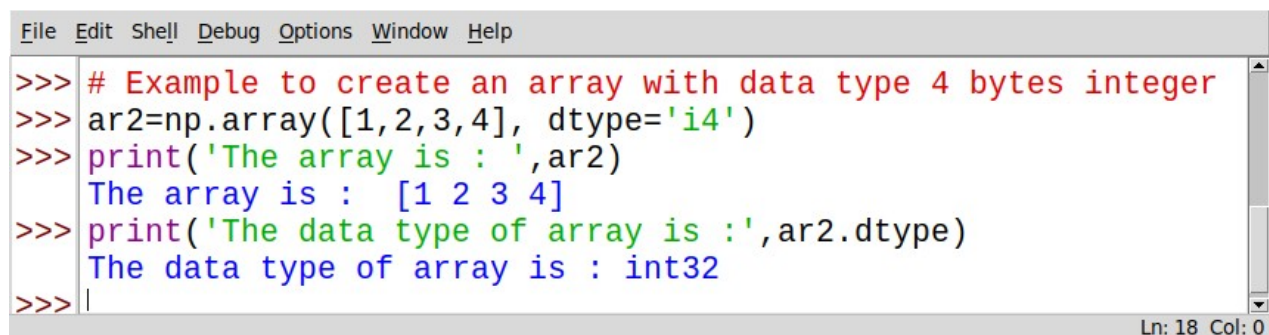


```
IDLE Shell 3.12.3
File Edit Shell Debug Options Window Help
Python 3.12.3 (main, Feb 4 2025, 14:48:35) [GCC 13.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> import numpy as np
>>> # Example to create an array of specified data type
>>> ar1=np.array([1,2,3,4], dtype='S')
>>> print ('The array is : ',ar1)
The array is : [b'1' b'2' b'3' b'4']
>>> print('The data type of array is :', ar1.dtype)
The data type of array is : |S1
>>>
```

In the above example, the original numeric array is converted into string by using dtype. In the output the prefix b indicates the **plain bytes** in front of the string. The data type of array is |S1. |S1 indicates the string of length 1.

Example 1.11 illustrates how to create an array with data type 4 bytes integer.

```
# Example to create an array with data type 4 bytes integer
>>> ar2=np.array([1,2,3,4], dtype='i4')
>>> print('The array is : ',ar2)
The array is : [1 2 3 4]
>>> print('The data type of array is :',ar2.dtype)
The data type of array is : int32
```



```
IDLE Shell 3.12.3
File Edit Shell Debug Options Window Help
>>> # Example to create an array with data type 4 bytes integer
>>> ar2=np.array([1,2,3,4], dtype='i4')
>>> print('The array is : ',ar2)
The array is : [1 2 3 4]
>>> print('The data type of array is :',ar2.dtype)
The data type of array is : int32
>>>
```

Example 1.12 illustrates to create an array with integer and string.

```
# Example to create an array with integer and string
>>> ar3=np.array(['1','2','a'], dtype='i')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
```

```
File Edit Shell Debug Options Window Help
>>> # Example to create an array with interger and string
>>> ar3=np.array(['1','2','a'], dtype='i')
Traceback (most recent call last):
  File "/usr/lib/python3.12/idlelib/run.py", line 580, in runcode
    exec(code, self.locals)
  File "<pyshell#15>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'a'
>>>
```

Ln: 26 Col: 31

In this example the literal cannot be converted into integer and hence it gives the error.

It is possible to create a NumPy array containing the data element of the same data type as well as the data element of different data types.

Converting Data Type on Existing Arrays

The data type of an existing array can be changed. To do this, make a copy of the array with the `astype()` method. The `astype()` function creates a copy of the array, and allows to specify the data type as a parameter. The data type can be specified using a string, like 'f' for float, 'i' for integer or you can use the data type directly like float for float and int for integer.

Example 1.12 illustrates to change data type from float to integer by using 'i' as parameter.

```
File Edit Shell Debug Options Window Help
>>> # Change data type from float to integer by using 'i' as parameter
>>> import numpy as np
>>> arr = np.array([1.1,2.2,3.3])
>>> print ('The array created as \n',arr)
The array created as
[1.1 2.2 3.3]
>>> arr1 = arr.astype('i')
>>> print('The array converted to integer is as \n',arr1)
The array converted to integer is as
[1 2 3]
>>> print('The data type of converted array using i is:',arr1.dtype)
The data type of converted array using i is: int32
>>> arr1 = arr.astype('int')
>>> print('The array converted to integer is as \n',arr1)
The array converted to integer is as
[1 2 3]
>>> print('The data type of converted array using int is:',arr1.dtype)
The data type of converted array using int is: int64
>>>
```

Ln: 15 Col: 0

NumPy Array Properties

A NumPy array is a powerful data structure in Python that provides many properties for working efficiently with large datasets. Here are some of its key properties:

1. Shape and Size Properties

ndarray.shape : Returns a tuple representing the dimensions of the array such as rows and columns. **Example 1.13** illustrates to print the shape of the array.

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]])
>>> print(a.shape)
(3, 2)
```

The output 3, 2 indicates that the array consists of 3 rows and 2 columns. This is the shape of an array.

ndarray.ndim : Returns the number of dimensions (axes) of the array. **Example 1.14** illustrates to print the dimension of the array.

```
>>> a=np.array([1, 2, 3])
>>> print(a.ndim)
1
```

The output 1 indicates the array of one dimension.

ndarray.size : Returns the total number of elements in the array. Following example illustrates the total number of elements in the array.

```
>>> a = np.array([[1, 2], [3, 4]])
>>> print(a.size)
4
```

The output 4 indicates that there are a total 4 elements in the array.

ndarray.itemsize : Returns the size (in bytes) of each element in the array. **Example 1.15** illustrates to print the size of array elements in bytes.

```
>>> a = np.array([1, 2, 3], dtype=np.int32)
>>> print(a.itemsize)
4
```

The output 4 indicates the size of array elements is 32 bits means 4 bytes.

ndarray.nbytes : Returns the total memory consumed by the array (in bytes). **Example 1.16** illustrates to print the total memory consumed by the array (in bytes).

```
>>> a = np.array([1, 2, 3])
>>> print(a.nbytes)
24
```

The output 24 indicates the total 24 bytes of memory consumed by the array. (assuming 64-bit integers)

2. Data Type Properties

ndarray.dtype : Returns the data type of the elements in the array. **Example 1.17** illustrates to display data type of the elements in the array.

```
>>> a = np.array([1, 2, 3])
>>> print(arr.dtype)
int64
```

The output 64 indicates the data type of the elements in the array is 64 bits integer.

ndarray.astype(dtype): Allows conversion of array elements to a specified type. **Example 1.18** illustrates to convert the data type of the array elements to the specified data type.

```
>>> a = np.array([4.3, 5.4])
>>> int_a = a.astype(int)
>>> print(int_a)
[4 5]
```

The output [4 5] shows that the float elements of array are converted into integer values.

3. Reshaping and Views

Reshaping means changing the shape of an array. The shape of an array is the number of elements in each dimension. Reshaping can change the shape of an array. It is possible to add or remove dimensions or change the number of elements in each dimension, provided the number of elements should be of exact count.

ndarray.reshape(shape): Returns a new array with the same data but a different shape. **Example 1.19** illustrates to reshape the array of the same data with different shapes.

```
>>> a = np.array([1, 2, 3, 4])
>>> reshaped = a.reshape(2, 2)
>>> print(reshaped)
[[1 2]
 [3 4]]
```

ndarray.ravel(): Flattens the array into a 1D array. **Example 1.20** illustrates to convert the 2D array to a 1D array with the same elements.

```
>>> a = np.array([[1, 2], [3, 4]])
>>> print(a.ravel())
[1 2 3 4]
```

The output [1 2 3 4] shows that the 2D array is converted to 1D array.

It is possible to convert 9 elements of 1 dimensional array to 2-dimensional array, but it is not possible to convert 8 elements of 1 dimensional array to 2-

dimensional array, it generates error. **Example 1.21** illustrates to reshaping the array from 1 dimension to 2 and 3 dimensions.

```
File Edit Shell Debug Options Window Help
>>> # Converting 1-Dim array with 12 elements into a 2-Dim array.
>>> import numpy as np
>>> ar1 = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
>>> ar2 = ar1.reshape(4, 3)
>>> print('1-Dim array converted to 2-Dim array is as \n',ar2)
1-Dim array converted to 2-Dim array is as
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
>>> # Converting 1-Dim array with 12 elements into a 3-Dim array.
>>> ar3 = ar1.reshape(2, 3, 2)
>>> print('1-Dim array converted to 3-Dim array is as \n',ar3)
1-Dim array converted to 3-Dim array is as
[[[ 1  2]
 [ 3  4]
 [ 5  6]]

 [[ 7  8]
 [ 9 10]
 [11 12]]]
>>> # Converting 1D array with 8 elements to a 2D array with 3 elements in each d
imension (will raise an error):
>>> ar = np.array([1, 2, 3, 4, 5, 6, 7, 8])
>>> ar4 = arr.reshape(3, 3)
Traceback (most recent call last):
  File "/usr/lib/python3.10/idlelib/run.py", line 578, in runcode
    exec(code, self.locals)
  File "<pyshell#20>", line 1, in <module>
NameError: name 'arr' is not defined. Did you mean: 'ar1'?
>>>
```

4. Accessing and Slicing

Slicing means taking elements from one given index to another given index. It is possible to slice instead of an index like this: `[start:end]`. It is possible to define the step, like this: `[start:end:step]`. If start or end is not passed, by default the start is considered 0, and end is considered length of array in that dimension. If a step is not passed it is considered as 1.

NumPy arrays support advanced slicing and indexing. **Example 1.22** illustrates to to access the elements of an array and slice the elements of an array.

```
>>> a = np.array([10, 20, 30, 40, 50])
>>> print(a[1:4])
[20 30 40]

>>> print(a[:2])
[10 30 50]
>>>
```

The first output shows the accessing the array elements from second to fifth element considering the index as 0.

The second output shows the slicing of the array in step 2 to print the first, third and fifth element of the array.

Example 1.22 illustrates the array slicing.

Negative Slicing

```
File Edit Shell Debug Options Window Help
>>> # Example to slice elements from index 2 to index 5 from the array
>>> import numpy as np
>>> arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> print('The array created is as \n',arr)
The array created is as
[1 2 3 4 5 6 7 8 9]
>>> print('The array sliced from index 2 to 5 is as \n',arr[2:5])
The array sliced from index 2 to 5 is as
[3 4 5]
>>> print('The array sliced from index 5 is as \n',arr[5:])
The array sliced from index 5 is as
[6 7 8 9]
>>> print('The array sliced from beginning to index 4 is as \n',arr[:4])
The array sliced from beginning to index 4 is as
[1 2 3 4]
>>>
```

The index of the array when referred from the end by using the minus operator, then it is negative indexing. Slicing can be done in steps by specifying steps.

Example 1.23 illustrates the negative indexing performed on the array and slicing in steps.

```
File Edit Shell Debug Options Window Help
>>> # Slice from the index 5 to 2 from the end
>>> import numpy as np
>>> arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> print ('Slice from the index 5 to 2 from the end\n', arr[-5:-2])
Slice from the index 5 to 2 from the end
[5 6 7]
>>> # Returning element of array in the step of 2
>>> print ('Element of array from index 2 to 8 in step of 2\n', arr[2:8:2])
Element of array from index 2 to 8 in step of 2
[3 5 7]
>>> print ('Element of array in step of 2 from entire array\n', arr[::2])
Element of array in step of 2 from entire array
[1 3 5 7 9]
>>>
```

Example 1.24 illustrates some more programs.

```
File Edit Shell Debug Options Window Help
>>> # Python code to create an array with integer values [24,46,57,14,68,34,89,92] and display array values from index 2 to 6 and negative index from 7 to 3
>>> import numpy as np
>>> A=np.array([24,46,57,14,68,34,89,92])
>>> print('Array from index 2 to 6 is \n', A[2:6])
Array from index 2 to 6 is
[57 14 68 34]
>>> print('Array from negative index 7 to 3 is \n', A[7:3:-1])
Array from negative index 7 to 3 is
[92 89 34 68]
>>>
>>> # Python code to create an array with integer values from 0 to 9 and display its values in reverse order
>>> import numpy as np
>>> arr= np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> print('Array created with values \n', arr)
Array created with values
[0 1 2 3 4 5 6 7 8 9]
>>> print('Array in reverse order is \n', arr[::-1])
Array in reverse order is
[9 8 7 6 5 4 3 2 1 0]
>>>
>>> # Python code to create 2 arrays as 'a' and 'b' and display sum of array values.
>>> import numpy as np
>>> a=np. array([2,3,4,5])
>>> b=np.array([6,7,8,9])
>>> print('Sum of values of two array is \n', a + b )
Sum of values of two array is
[ 8 10 12 14]
>>>
>>> # Python code to create an array as 'a' and increase array values by 10.
>>> import numpy as np
>>> a=np. array([ 2,3,4,5 ])
>>> print('Array values after adding 10 \n', a + 10)
Array values after adding 10
[12 13 14 15]
>>>
```

Ln: 37 Col: 0

5.

Mathematical and Logical Properties

NumPy arrays enable element-wise operations:

```
>>> a = np.array([1, 2, 3])
>>> print(a * 2)
[2 4 6]
```

The output shows that each element of the array is multiplied by 2.

6. Broadcasting

Arrays can operate with different shapes using **broadcasting** rules.

```
>>> a1 = np.array([1, 2, 3])
>>> a2 = np.array([[1], [2], [3]])
>>> a = a1 + a2
>>> print (a)
[[2 3 4]
 [3 4 5]
 [4 5 6]]
```

The output shows that each element of array a1 is added to all the elements of array a2 to form the new array.

7. Boolean Properties

ndarray.all() : Returns True if all elements are non-zero or True.

```
>>> a = np.array([1, 2, 3])
>>> print(a.all())
True
```

The output shows that all the elements array are non-zero.

ndarray.any() : Returns True if at least one element is non-zero or True.

```
>>> a = np.array([0, 0, 1])
>>> print(a.any())
True
```

The output shows that at least one element of the array is non-zero

8. Copy vs View

ndarray.view() : Creates a new array object with shared data.

ndarray.copy() : Creates a new array with a copy of the data.

These properties make NumPy arrays efficient and versatile for numerical and scientific computing.

Session 2. Array Manipulation using NumPy

NumPy is a foundation library for scientific computations in Python. It contains sophisticated functions and tools for integrating with other programming languages as well. During data analysis, it is widely used to handle arrays as it offers a powerful n-dimensional array object, faster than a traditional list in Python. In this session, we will discuss different array manipulation techniques.

Joining and splitting

Joining and splitting arrays are fundamental operations in NumPy. Here's a detailed explanation of how to perform these operations:

Joining Arrays

Joining merges multiple arrays into one. NumPy provides several functions to join arrays along specified axes.

1. Concatenation of Arrays

np.concatenate() : This function is used to join two or more arrays along an existing axis. Here, arrays must have the same shape along the axis being joined. If the axis is None, arrays are flattened before use. Default is 0. *Example 2.1* illustrates the joining of two arrays.

Example 2.1:

```
>>> import numpy as np
# Concatenation of Arrays
>>> a1=np.array([[1,2],[3,4]])
>>> a2=np.array([[5,6],[7,8]])
>>> # Join along rows (axis=0)
>>> result = np.concatenate((a1,a2), axis=0)
>>> print (result)
```

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

$$a1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$a2 = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

$$\text{result} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}$$


```
File Edit Shell Debug Options Window Help
>>> import numpy as np
>>> a1=np.array([[1,2],[3,4]])
>>> a2=np.array([[5,6],[7,8]])
>>> # Join along rows (axis=0)
>>> result = np.concatenate((a1,a2), axis=0)
>>> print (a1)
[[1 2]
 [3 4]]
>>> print (a2)
[[5 6]
 [7 8]]
>>> print (result)
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
>>>
```

Ln: 6 Col: 26

In the output, observe that array a1 and a2 are joined together to produce a resultant array consisting of all elements of array a1 and a2. Observe that when axis=0 then joining will be a long row as illustrated in the *Example 2.2*.

Example 2.2:

```
>>> # Join along columns (axis=1)
>>> result = np.concatenate((a1, a2), axis=1)
>>> print (result)
[[1 2 5 6]
 [3 4 7 8]]
```

Observe that when axis=1 then arrays are joined along columns.

$$a1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$
$$a2 = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$
$$result = \begin{bmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \end{bmatrix}$$

```
File Edit Shell Debug Options Window Help
>>> # Join along columns (axis=1)
>>> result = np.concatenate((a1, a2), axis=1)
>>> print (result)
[[1 2 5 6]
 [3 4 7 8]]
>>>
```

Ln: 50 Col: 0

2. Vertical Stack

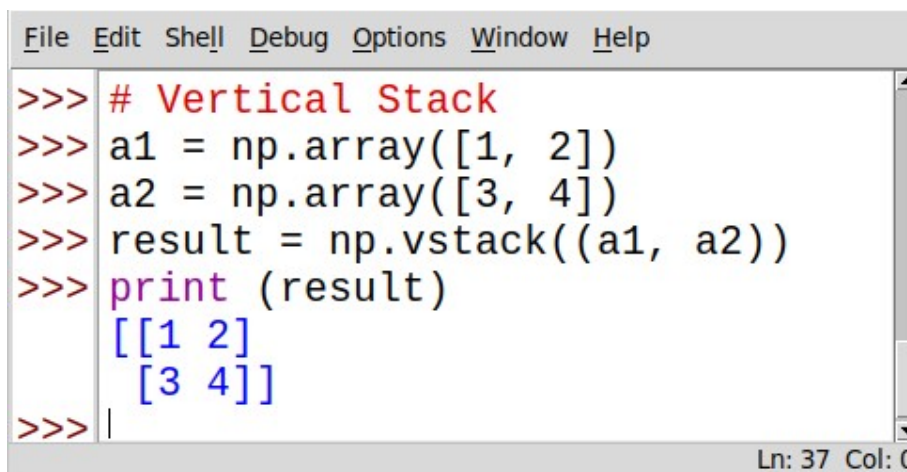
np.vstack() : This function is used to stack arrays vertically, that is, row-wise. This is equivalent to concatenation along the first axis after 1-D arrays of shape $(N,)$ have been reshaped to $(1,N)$.

Example 2.3:

```
# Vertical Stack
>>> a1 = np.array([1, 2])
>>> a2 = np.array([3, 4])
>>> result = np.vstack((a1, a2))
>>> print (result)
[[1 2]
 [3 4]]
```

In *Example 2.3*, observe that the array `a1` and `a2` are stacked vertically.

```
a1 = [1 2]
a2 = [3 4]
result =  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ 
```

A screenshot of a Jupyter Notebook interface. The top bar shows menu items: File, Edit, Shell, Debug, Options, Window, Help. The main area contains a code cell with the following text: >>> # Vertical Stack, >>> a1 = np.array([1, 2]), >>> a2 = np.array([3, 4]), >>> result = np.vstack((a1, a2)), >>> print (result). The output of the print statement is displayed below the code: [[1 2], [3 4]]. The status bar at the bottom right indicates 'Ln: 37 Col: 0'.

3. Horizontal Stack

np.hstack() : This function is used to stack arrays horizontally, means, column-wise. This is equivalent to concatenation along the second axis, except for 1-D arrays where it concatenates along the first axis. *Example 2.4* illustrates how to stack the array horizontally.

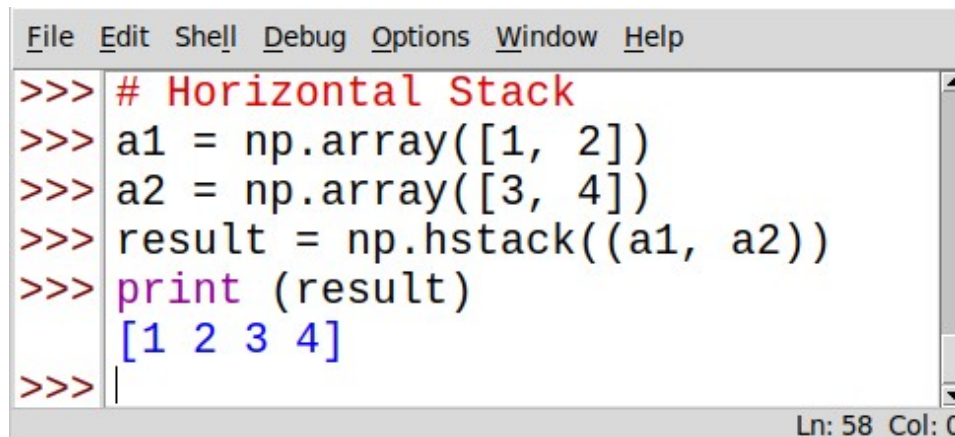
Example 2.4:

```
# Horizontal Stack
>>> a1 = np.array([1, 2])
>>> a2 = np.array([3, 4])
```

```
>>> result = np.hstack((a1, a2))
>>> print (result)
[1 2 3 4]
```

In this output, you can observe that array a1 and a2 are stacked horizontally.

```
a1 = [1 2]
a2 = [3 4]
result = [1 2 3 4]
```



```
File Edit Shell Debug Options Window Help
>>> # Horizontal Stack
>>> a1 = np.array([1, 2])
>>> a2 = np.array([3, 4])
>>> result = np.hstack((a1, a2))
>>> print (result)
[1 2 3 4]
>>> |
Ln: 58 Col: 0
```

4. Depth Stack

np.dstack() : This function is used to stack arrays along the third dimension, that is, depth-wise. This is equivalent to concatenation along the third axis after 2-D arrays of shape (M, N) have been reshaped to $(M, N, 1)$ and 1-D arrays of shape $(N,)$ have been reshaped to $(1, N, 1)$. **Example 2.5** illustrates how to stack the array depthwise.

Example 2.5:

Depth Stack

```
>>> a1 = np.array([[1, 2], [3, 4]])
>>> a2 = np.array([[5, 6], [7, 8]])
>>> result = np.dstack((a1, a2))
>>> print(result)
[[[1 5]
  [2 6]]
 [[3 7]
  [4 8]]]
```

$$a1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$a2 = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

$$\text{result} = \begin{bmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{bmatrix}$$

```
File Edit Shell Debug Options Window Help
>>> # Depth Stack
>>> a1 = np.array([[1, 2], [3, 4]])
>>> a2 = np.array([[5, 6], [7, 8]])
>>> result = np.dstack((a1, a2))
>>> print(result)
[[[1 5]
   [2 6]]

  [[3 7]
   [4 8]]]
>>>
```

Ln: 109 Col: 0

5. Column Stack

np.column_stack() : This function stacks 1D arrays as columns into a 2D array. Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with `hstack`. 1-D arrays are turned into 2-D columns first. **Example 2.6** illustrates how to stack the array in a column.

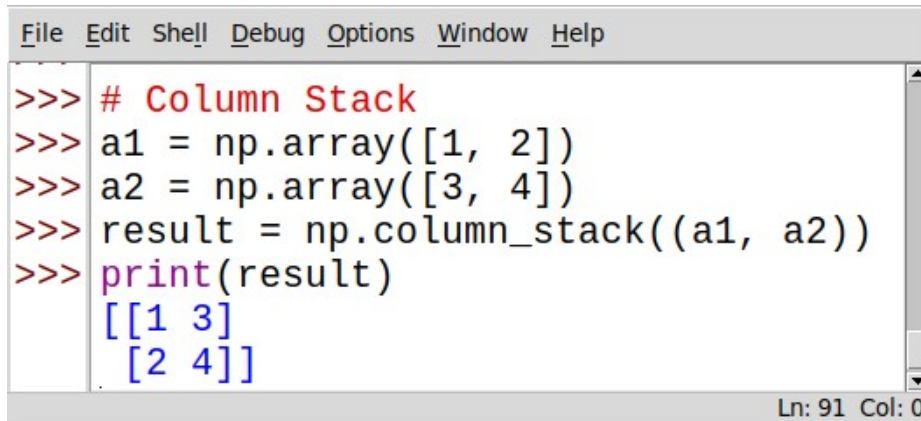
Example 2.6:

```
# Column Stack
>>> a1 = np.array([1, 2])
>>> a2 = np.array([3, 4])
>>> result = np.column_stack((a1, a2))
>>> print(result)
[[1 3]
 [2 4]]
```

Observe that in *Example 2.6*, row of array `a1` and `a2` is inserted as column in `result`. Result is a two-dimensional array.

$$a1 = [1 \ 2]$$

$$a2 = [3 \ 4]$$

$$\text{result} = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$


```
File Edit Shell Debug Options Window Help
>>> # Column Stack
>>> a1 = np.array([1, 2])
>>> a2 = np.array([3, 4])
>>> result = np.column_stack((a1, a2))
>>> print(result)
[[1 3]
 [2 4]]
Ln: 91 Col: 0
```

6. Row Stack

np.row_stack() : This function stacks 1D arrays as rows into a 2D array. It is similar to `vstack`. It stack arrays in sequence vertically, that is, row wise. This is equivalent to concatenation along the first axis after 1-D arrays of shape (N) have been reshaped to $(1,N)$. **Example 2.7** illustrates to stack the array in a column.

Example 2.7

```
# Row Stack
>>> a1 = np.array([1, 2])
>>> a2 = np.array([3, 4])
>>> result = np.row_stack((a1, a2))
>>> print(result)
[[1 2]
 [3 4]]
```

Observe that one dimensional arrays `a1` and `a2` are stacked as rows in `result`. Result is a two-dimensional array.

$$a1 = [1 \ 2]$$

$$a2 = [3 \ 4]$$

$$\text{result} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
File Edit Shell Debug Options Window Help
>>> # Row Stack
>>> a1 = np.array([1, 2])
>>> a2 = np.array([3, 4])
>>> result = np.row_stack((a1, a2))
>>> print(result)
[[1 2]
 [3 4]]
>>> |
Ln: 119 Col: 0
```

Splitting Arrays

Splitting is the reverse operation of Joining. Joining merges multiple arrays into one and Splitting breaks one array into multiple. NumPy provides functions to split arrays into multiple sub-arrays.

1. Split an Array

np.split() : This function is used to Split an array into multiple sub-arrays along a specified axis. Its general format is: `numpy.split(ary, indices_or_sections, axis=0)`.

If `indices_or_sections` are an integer, `N`, the array will be divided into `N` equal arrays along the axis. If such a split is not possible, an error is raised.

If `indices_or_sections` are a 1-D array of sorted integers, the entries indicate where along the axis the array is split. **Example 2.7** illustrates the splitting of array.

Example 2.7

Split an Array

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
>>> # Split into 3 equal parts
>>> result = np.split(a, 3)
>>> print(result)
[array([1, 2]), array([3, 4]), array([5, 6])]
```

Observe that in the above example array is split into three equal parts. Each part contains two elements. If the array cannot be split equally, it raises an error.

```
a = [1 2 3 4 5 6]
result = [[1 2] [3 4] [5 6]]
```

2. Unequal Splitting

np.array_split() : This function splits an array into multiple sub-arrays but allows unequal splitting. Its general format is: `numpy.array_split(ary, indices_or_sections, axis=0)`.

This function allows indices or sections to be an integer that does not equally divide the axis. For an array of length l that should be split into n sections, it returns $l \% n$ sub-arrays of size $l // n + 1$ and the rest of size $l // n$.

Example 2.7 illustrates the splitting of array.

Splitting of array

```
>>> a = np.array([1, 2, 3, 4, 5])
>>> # Split into 3 parts (unequal)
>>> result = np.array_split(a, 3)
>>> print(result)
[array ([1, 2]), array ([3, 4]), array ([5])]
```

Observe that the given array contains five elements. It is splitted into three arrays. The first two arrays contain two elements each and the third array contain only one element.

$a = [1 \ 2 \ 3 \ 4 \ 5]$

$result = [[1 \ 2] \ [3 \ 4] \ [5]]$

3. Vertical Split

np.vsplit() : This function splits an array vertically, meaning, row-wise. It works on arrays with at least 2 dimensions. This is equivalent to split with `axis=0` by default. The array is always split along the first axis regardless of the array dimension.

Example 2.8 illustrates the vertical splitting of array.

Vertical splitting of array.

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]])
>>> result = np.vsplit(a, 3)
>>> print(result)
[array([[1, 2]]), array([[3, 4]]), array([[5, 6]])]
```

Observe that the array `a` is vertically splitted into three arrays each containing two elements.

$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$

$result = [[1 \ 2] \ [3 \ 4] \ [5 \ 6]]$

4. Horizontal Split

np.hsplit() : This function splits an array horizontally, meaning column-wise. It works on arrays with at least 2 dimensions. It is equivalent to split with axis=1, the array is always split along the second axis except for 1-D arrays, where it is split at axis=0.

Example 2.9 illustrates the horizontal splitting of array.

```
# Horizontal splitting of array
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> result = np.hsplit(a, 3)
>>> print(result)
[array([[1],
        [4]]), array([[2],
        [5]]), array([[3],
        [6]])]
```

Observe that array a is split into three arrays horizontally.

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$
$$\text{result} = \begin{bmatrix} 1 & 2 \\ 4 & 2 \\ 5 & 3 \\ 6 \end{bmatrix}$$

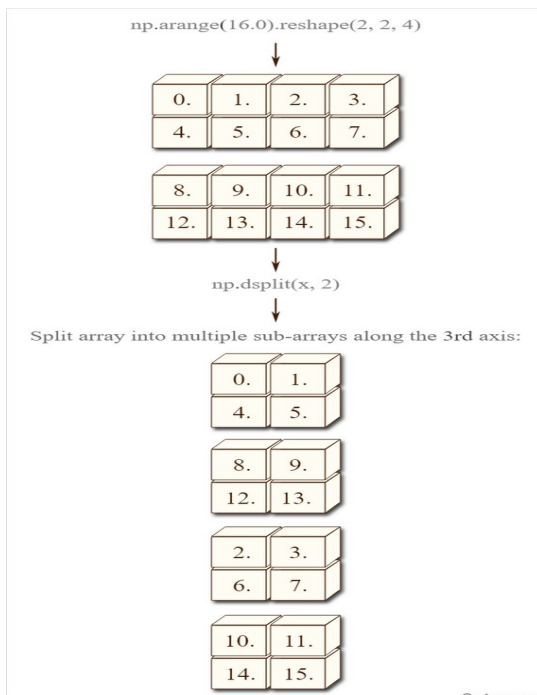
5. Depth Split

np.dsplit() : This function splits an array along the third dimension, that is, depth-wise. It works on arrays with at least 3 dimensions. It is equivalent to split with axis=2, the array is always split along the third axis provided the array dimension is greater than or equal to 3. **Example 2.10** illustrates the splitting of array in depth.

```
# Splitting of array in depth
>>> a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
>>> result = np.dsplit(a, 2)
>>> print(result)
[array([[[1],
        [3]],
        [[5],
        [7]]]), array([[[2],
        [4]],
        [[6],
```

[8]]]]]

Observe that the array `a` is a three-dimensional array. It is split into two arrays.



Ref Fig. (Modify it as per example)

Reshaping Arrays

Reshaping means changing the shape of an array. The shape of an array is the number of elements in each dimension. By reshaping we can add or remove dimensions or change the number of elements in each dimension.

Reshape Function

`np.reshape()` : This function creates a new array with the same data but a different shape. The total number of elements must remain the same. You can specify one dimension as -1, and NumPy will infer its value.

The general format is:

`numpy.reshape (a, /, shape=None, order='C', *, newshape=None, copy=None).`

Syntax

```
np.reshape(array, new_shape)
```

Example 2.10 illustrates the reshapping of array.

```
# Reshapping of array
```

```
>>> import numpy as np
```

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

```
>>> # Reshape into a 2x3 array
```

```
>>> reshaped = np.reshape(a, (2, 3))
```

```
>>> print(reshaped)
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

```
>>> # Infer one dimension using -1
>>> reshaped = np.reshape(a, (-1, 2))
>>> print(reshaped)
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

```
>>>
```

Observe that in the above example, the given array `arr` is reshaped into two arrays with the same number of elements. Each array contains three elements. When you infer one dimension using `-1` then each array contains 2 elements.

2. `ndarray.reshape()` : It works like `np.reshape()` but is a method of the array object.

```
a = np.array([1, 2, 3, 4])
reshaped = a.reshape(2, 2)
print(reshaped)
```

Output:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Observe that in the above example, the given array is reshaped into two arrays with the same elements.

3. Flattening Arrays

Flattening means to convert an array into a one-dimensional array. Flattening an array is the process of taking the nested elements within an array and putting them into a single array, in other words converting a multi-dimensional array into a single one-dimensional array. There are two functions to flatten the array.

`ndarray.ravel()` : Returns a flattened view if possible.

`ndarray.flatten()` : Returns a copy of the flattened array.

Example 2.11 illustrates the Flattening an array.

```
>>> Flattening an array
>>> a = np.array([[1, 2], [3, 4]])
>>> print(a.ravel())
[1 2 3 4]
```

```
>>> print(a.flatten())  
[1 2 3 4]
```

Observe that in the above example a two-dimensional array is converted into one dimensional array.

```
a =  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$   
result = [1 2 3 4]
```

Resizing Arrays

To resize an array, you can create a new array with a larger capacity, copy the elements from the old array to the new one, and then replace the old array with the new one.

np.resize() : It creates a new array with a specified shape. If the new shape has more elements, the array is repeated to fill it. If the new shape has fewer elements, the array is truncated.

Syntax:

```
np.resize(array, new_shape)
```

Example 2.12 illustrates the to resize the array.

```
>>> # Resizing Array  
>>> a = np.array([1, 2, 3, 4])  
>>> # Resize to a 2x3 array (data repeats)  
>>> resized = np.resize(a, (2, 3))  
>>> print(resized)  
a = [1 2 3 4]  
result =  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 1 & 2 \end{bmatrix}$   
>>>  
>>> # Resize to a smaller shape (truncated)  
>>> resized = np.resize(arr, (2, 2))  
>>> print(resized)  
a = [1 2 3 4]  
result =  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ 
```

>>> In the above example the given array arr is resized into a 2x3 array with data repeated. Further it is resized to a smaller shape.

2. In-Place Resizing

ndarray.resize() : This function modifies the array itself to match the new shape. If the new shape is larger, the array is filled with default values usually 0.

Syntax:

```
ndarray.resize(new_shape)
```

Example 2.13 illustrates how to resize the array in-place.

```
# Resize the array in place
>>> a = np.array([1, 2, 3, 4])
>>> # Resize the array in-place
>>> a.resize((2, 3))
>>> print(a)
a = [1 2 3 4]
result =  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 0 \end{bmatrix}$ 
>>>
```

Observe that in the above example the given array arr is resized into a two-dimensional array.

Key Differences Between Reshape and Resize

Feature	Reshape	Resize
Returns/ Modifies	Returns a new array (view or copy).	Modifies the array in-place.
Size Match	Total elements must match before reshaping.	May add or truncate elements.
Behaviour on Change	Keeps the data intact, just rearranges it.	May repeat or truncate elements.

Changing Dimensions

1. Adding Dimensions

We can add or remove dimensions to an array by using Numpy functions.

np.newaxis : Adds a new dimension to an array.

np.expand_dims() : Explicitly adds a new axis at a specific position.

Example 2.14 illustrates how to add a new dimension to array.

```
>>> # adding dimension to array.
>>> a = np.array([1, 2, 3])
>>> # Add a new dimension
```

```

>>> print(a[np.newaxis, :])
[[1 2 3]]
>>>
>>> print(np.expand_dims(a, axis=1))
a = [1 2 3]
result =  $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ 

```

In the above example the given one dimensional array is converted into a two-dimensional array.

2. Removing Dimensions

np.squeeze() : Removes dimensions of size 1.

Example 2.15 illustrates how to remove new dimension of array.

```

>>> # Removing dimension of array.
>>> a = np.array([[[[11, 22, 33]]]])
>>> # Shape: (1, 1, 3)
>>> # Remove size-1 dimensions
>>> squeezed = np.squeeze(a)
>>> print(squeezed)
a = [[11 22 33]]
result = [11 22 33]

```

In the above example observe that the dimension of array a is reduced.

Session 3. Array Computation using NumPy

Arithmetic operations such as addition, subtraction, multiplication, division and exponentiation can be performed on arrays in Numpy.

Operations such as sum, product, mean, standard deviation, dot product and matrix multiplication can be performed on arrays in Numpy.

NumPy supports efficient element-wise arithmetic operations on arrays. These operations are performed on corresponding elements of the arrays and are faster than traditional Python loops.

Arithmetic Operations

The primary arithmetic operations supported by NumPy are:

1. Addition

Addition corresponding elements of two arrays or an array and a scalar. Operator used for addition is `+`. Function for addition is `np.add()`.

Example 3.1 illustrates the addition of two arrays.

```
# Addition of two arrays
>>> import numpy as np
>>> a1 = np.array([11, 22, 33])
>>> a2 = np.array([10, 20, 30])
>>> # Element-wise addition
>>> result = a1 + a2
>>> print(result)
[21 42 63]
```

In the above example, observe that column wise addition of two arrays `a1` and `a2` takes place.

```
a1 = [11 22 33]
a2 = [10 20 30]
result = [21 42 63]
```

It is also possible to add a scalar in the array. By adding a scalar in the array, the scalar number will get added to each element of the array in the resultant array.

Example 3.2 illustrates to add a scalar to array.

```
>>> # Adding a scalar to array
>>> result = a1 + 10
>>> print(result)
[21 32 43]
```


In the above Example 3.2, a scalar 10 is added into each element of array a1.

```
a1 = [11 22 33]
result = a1 + 10
      [21 32 43]
```

Numpy.add() Function

NumPy's **numpy.add()** is a function that performs element-wise addition on NumPy arrays. This means it adds the corresponding elements between two arrays, element by element, instead of treating them as single values.

numpy.add() function is used when we want to compute the addition of two arrays. It adds arguments element-wise. If the shape of two arrays is not the same, that is `a1.shape != a2.shape`, they must be broadcastable to a common shape.

Example 3.3 illustrates to use numpy add function.

```
# Using numpy add function.
>>> import numpy as np
>>> # Numpy array and a scalar
>>> a1 = np.array([11, 22, 33])
>>> # Using numpy.add() with an array and a scalar
>>> result = np.add(a1, scalar)
>>> print("Result of adding array and scalar:", result)
Result of adding array and scalar: [15 26 37]
a1 = [11 22 33]
result = [15 26 37]
```

2. Subtraction

Subtraction corresponds to the difference of elements of two arrays or a scalar from an array. Operator used for subtraction is `-` and the function for subtraction is `np.subtract()`.

Example 3.4 illustrates the subtraction on arrays.

```
# Subtraction on arrays.
>>> import numpy as np
>>> a1 = np.array([10, 20, 30])
>>> a2 = np.array([40, 50, 60])
>>> result = a2 - a1
>>> print(result)
```

```
[30 30 30]
```

In the above output, observe the element wise difference of two arrays `a1` and `a2`.

```
a1 =[10 20 30]
a2 =[40 50 60]
result = a2 - a1
      [30 30 30]
```

It is also possible to subtract a number from an array. It will subtract the specified number from each element of the array.

Example 3.5 illustrates to subtract a specified number from arrays.

```
>>> Subtracting a specified number from arrays
>>> result = a1 - 11
>>> print(result)
[-1  9 19]
```

In the above example, the number 11 is subtracted from each element of the array and gives the resultant array.

```
a1 =[10 20 30]
result = a1 - 11
      [-1  9 19]
```

Numpy.subtract Function

numpy.subtract() function is used when we want to compute the difference of two arrays. It returns the difference of `arr1` and `arr2`, element-wise.

Example 3.6 illustrates to use `numpy.subtract()` function.

```
>>> # use numpy.subtract() function
>>> import numpy as np
>>> a1_in = np.array([[21, -40, 45], [-16, 12, 10]])
>>> a2_in = np.array([[10, -15, 25], [20, -22, 19]])
>>> print ("1st Input array :\n", a1_in)
1st Input array :
[[ 21 -40  45]
 [-16  12  10]]
>>> print ("1st Input array :\n", a2_in)
1st Input array :
[[ 10 -15  25]
```

```

[ 20 -22  19]]
>>> a_out = np.subtract(a1_in, a2_in)
>>> print ("Output array:\n", a_out)
Output array:
[[ 11 -25  20]
 [-36  34  -9]]
>>>

```

```


$$a1 = \begin{bmatrix} 21 & -40 & 45 \\ -16 & 12 & 10 \end{bmatrix}$$


$$a2 = \begin{bmatrix} 10 & -15 & 25 \\ 20 & -22 & 19 \end{bmatrix}$$


$$result = a1 - a2$$


$$\begin{bmatrix} 11 & -25 & 20 \\ -36 & 34 & -9 \end{bmatrix}$$


```

3. Multiplication

This operation multiplies corresponding elements of two arrays or scales an array by a scalar. Operator used for multiplication is *. Function used for multiplication is np.multiply().

Example 3.7 illustrates the multiplication of arrays.

```

>>> # Multiplication of arrays
>>> import numpy as np
>>> a1 = np.array([11, 22, 33])
>>> a2 = np.array([3, 2, 1])
>>> result = a1 * a2
>>> print(result)
[33 44 33]

```

Observe that element wise multiplication of two arrays a1 and a2 will take place.

```


$$a1 = [11 \ 22 \ 33]$$


$$a2 = [3 \ 2 \ 1]$$


$$result = a1 \times a2$$


$$[33 \ 44 \ 33]$$


```

Similarly it is possible to multiply the array with scalar. The scalar will get multiplied by each element of the array.

```
>>> result = a1 * 2
>>> print(result)
[22 44 66]
```

Observe the output the scalar 2 is multiplied with all elements of array a1.

```
a1 = [11 22 33]
result = a1 × 2
      [22 44 66]
```

Numpy.Multiplication

numpy.multiply() function is used when we want to compute the multiplication of two arrays. It returns the product of arr1 and arr2, element-wise.

Example 3.8 illustrates to use of the numpy.multiply() function.

```
>>> # Python program explaining numpy.multiply() function
>>> import numpy as np
>>> a1_in = np.array([[21, -17, 15], [-6, 12, 10]])
>>> a2_in = np.array([[10, -7, 8], [15, -2, 19]])
>>> print ("1st Input array : \n", a1_in)
1st Input array :
[[ 21 -17  15]
 [ -6  12  10]]
>>> print ("2nd Input array : \n", a2_in)
2nd Input array :
[[10 -7  8]
 [15 -2 19]]
>>> out_arr = np.multiply(a1_in, a2_in)
>>> print ("Resultant output array: \n", a_out)
Resultant output array:
[[ 11 -25  20]
 [-36  34  -9]]
>>>
```

$$a1 = \begin{bmatrix} 21 & -17 & 15 \\ -6 & 12 & 10 \end{bmatrix}$$

$$a2 = \begin{bmatrix} 10 & -7 & 8 \\ 15 & -2 & 19 \end{bmatrix}$$

$$\begin{bmatrix} 11 & -25 & 20 \\ -36 & 34 & -9 \end{bmatrix}$$

4. Division

This operation divides corresponding elements of two arrays or divides an array by a scalar. Operator used for division is / and the function used for division is np.divide().

Example 3.9 illustrates the the division of arrays.

```
>>> # Python program for division of array
>>> import numpy as np
>>> a1 = np.array([11, 22, 33])
>>> a2 = np.array([44, 55, 66])
>>> result = a2 / a1
>>> print(result)
[4.  2.5 2. ]
>>> Observe that elements of array a2 are divided by elements of array a1.
```

$$a1 = [11 \ 22 \ 33]$$

$$a2 = [44 \ 55 \ 66]$$

$$\text{result} = a2 \div a1$$

$$[4 \ 2.5 \ 2]$$

Division by Scalar

It is possible to multiply the array with scalar. The scalar will get multiplied by each element of the array.

Example 3.9 illustrates the division by saclar

```
>>> result = a1 / 2
>>> print(result)
[ 5.5 11. 16.5]
```

In the above example all the elements of array a1 are divided by 2.

```
a1 = [11 22 33]
result = a1 ÷ 2
[5.5 11 16.5]
```

NumPy.divide()

Array element from the first array is divided by elements from the second element (all happens element-wise). Both arr1 and arr2 must have the same shape and element in arr2 must not be zero; otherwise it will raise an error.

Example 3.10 illustrates to use of the `numpy.divide()` function.

```
>>> # Python program explaining divide () function
>>> import numpy as np
>>> # input_array
>>> a1 = [22, 27, 12, 21, 23]
>>> a2 = [2, 3, 4, 5, 6]
>>> out = np.divide(a1, a2)
>>> print ("a1: \n", a1)
a1: [22, 27, 12, 21, 23]
>>> print ("a2: \n", a2)
a2: [2, 3, 4, 5, 6]
>>> print(out)
[11.          9.          3.          4.2          3.83333333]
>>>
```

```
a1 = [22 27 12 21 23]
a2 = [2 3 4 5 6]
result = [11 9 3 4.2 3.834]
```

Divide by zero error

If any element of the second array is 0 then it is not possible to divide the array and will raise the division by zero error as illustrated below.

Example 3.11 illustrates the division by zero error .

```
>>> # Python program explaining divide() function
>>> import numpy as np
>>> # input_array
>>> a1 = [22, 27, 12, 21, 23]
>>> a2 = [2, 3, 0, 5, 6]
>>> print ("a1: \n", a1)
```



```
a1: [22, 27, 12, 21, 23]
>>> print ("a2: \n", a2)
a2: [2, 3, 0, 5, 6]
>>> out = np.divide(a1, a2)
<stdin>:1: RuntimeWarning: divide by zero encountered in divide
```

In this output, observe that error is generated on division by 0.

```
a1 = [22 27 12 21 23]
a2 = [2 3 4 5 6]
result = [0]
```

5. Floor Division

This operation performs integer division, that is, it truncates the decimal. Operator used for floor division is `//` and the function used is `np.floor_divide()`.

Example 3.12 illustrates the floor division of arrays.

```
>>> # Python code illustrates the floor division of arrays.
>>> import numpy as np
>>> a1 = np.array([10, 20, 30])
>>> a2 = np.array([44, 59, 62])
>>> result = a2 // a1
>>> print(result)
[4 2 2]
```

Observe that in the above output the Fractional part of division does not appear in the result.

```
a1 = [10 20 30]
a2 = [44 59 62]
result = a2 ÷ a1
[4 2 2]
```

Np.floor_divide() Function

Array element from the first array is divided by the elements from the second array element-wise. Both array 1 and array 2 must have the same shape. It is equivalent to the Python `//` operator.

Example 3.13 illustrate to use `np.floor_divide()` function.

```
# Python program illustrating np.floor_divide() function
import numpy as np
>>> a1 = [2, 2, 2, 3, 3]
```

```

>>> a2 = [2, 3, 4, 3, 6]
>>> out = np.floor_divide(a1, a2)
>>> print ("array 1: \n ", a1)
array 1: [2, 2, 2, 3, 3]
>>> print ("array2: \n ", a2)
array2: [2, 3, 4, 3, 6]
>>> print ("\nOutput array :\n", out)
Output array :
[1 0 0 1 0]
>>>
a1 =[2 2 2 3 3]
a2 =[2 3 4 3 6]
result =[1 0 0 1 0]

```

6. Modulus (Remainder)

This operation calculates the remainder of division for each element. Operator used is % and the function used is np.mod().

Example 3.14 illustrate the use of Modulus.

```

>>> # Python program to illustrate Modulus (Reminder)
>>> import numpy as np
>>> a1 = np.array([10, 20, 30])
>>> a2 = np.array([42, 55, 66])
>>> result = a2 % a1
>>> print(result)
[ 2 15  6]
>>>
a1 =[10 20 30]
a2 =[42 55 66]
result =a2%a1
[2 15 6]

```

Np.mod() Function

This function returns element-wise remainder of division between two arrays a1 and a2 i.e. $a1 \% a2$. It returns 0 when a2 is 0 and both a1 and a2 are (arrays of) integers.

Example 3.15 illustrate the use of Np.mod() function.

```

>>> # Python program to illustrate numpy.mod() function
>>> import numpy as np
>>> a1_in = np.array([2, -4, 7])
>>> a2_in = np.array([2, 3, 4])
>>> a_out = np.mod(a1_in, a2_in)
>>> print ("Dividend array : ", a1_in)
Dividend array : [ 2 -4  7]
>>> print ("Divisor array : ", a2_in)
Divisor array : [2 3 4]
>>> a_out = np.mod(a1_in, a2_in)
>>> print ("Output remainder array:", a_out)
Output remainder array: [0 2 3]
>>>
a1 =[2 -4 7]
a2 =[2 3 4]
result =[0 2 3]

```

7. Exponentiation

This operation raises each element of an array to the power of corresponding elements in another array or a scalar. Operator used for operation is ****** and the function used is **np.power()**.

Example 3.16 illustrate the Exponentiation operation.

```

>>> # Python program to illustrate exponentiation operation
>>> import numpy as np
>>> a1 = np.array([0, 1, 2])
>>> a2 = np.array([4, 5, 6])
>>> a1exp = a1 ** 2
>>> a2exp = a2 ** 3
>>> print(a1exp)
[0 1 4]
a1 =[0 1 2]
a2 =[4 5 6]
result =(a1)2
      [0 1 4]
>>> print(a2exp)

```

```
[ 64 125 216]
a1 =[0  1  2]
a2 =[4  5  6]
result =(a2)3
[64 125 216]
```

>>> Observe that all elements of array a1 are raised to 2 and a2 raised by 3.

```
>>> result = np.power(a1, a2)
>>> print(result)
[ 0  1 64]
>>>
```

```
a1 =[0  1  2]
a2 =[4  5  6]
result =(a1)a2
[0  1 64]
```

Here all elements of arr1 are raised by elements of arr2. $1^{**}4=1$, $2^{**}5=32$, $3^{**}6=729$.

Np.power() Function

Array element from the first array is raised to the power of the element from the second element, element-wise. Both arr1 and arr2 must have the same shape and each element in arr1 must be raised to corresponding +ve value from arr2; otherwise it will raise a ValueError.

Example 3.17 illustrate the use of np.power function

```
>>> # Python program illustrating power() function
>>> import numpy as np
>>> # input_array
>>> a1 = [2, 2, 2, 2, 2]
>>> a2 = [2, 3, 4, 5, 6]
>>> out = np.power(a1, a2)
>>> print ("array 1 : ", a1)
array 1 :  [2, 2, 2, 2, 2]
>>> print ("array 2 : ", a2)
array 2 :  [2, 3, 4, 5, 6]
>>> # output_array
```

```
>>> print ("\nOutput array : ", out)
Output array : [ 4  8 16 32 64]
>>>
a1 = [2  2  2  2  2]
a2 = [2  3  4  5  6]
result = (a1)a2
[4  8 16 32 64]
```

Broadcasting in Arithmetic

NumPy uses broadcasting to perform operations on arrays of different shapes, provided they are compatible.

Example 3.18 illustrate the Broadcasting in Arithmetic

```
>>> # Python program illustrating broadcasting in arithmetic
>>> a = np.array([[1, 2], [3, 4]])
>>> # Broadcasting a scalar
>>> result = a + 12
>>> print(result)
[[13 14]
 [15 16]]
a =  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ 
result = a + 12
 $\begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix}$ 
>>> # Broadcasting a 1D array
>>> result = a + np.array([1, 2])
>>> print(result)
[[2 4]
 [4 6]]
>>>
```

$$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$\text{result} = a + \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 4 \\ 4 & 6 \end{bmatrix}$$

Aggregate Arithmetic Operations

These functions compute aggregate results across elements of the array.

Sum

np.sum() function returns the sum of array elements over the specified axis.

Example 3.19 illustrate the use of np.sum() function.

```
>>> # Python code to illustrate np.sum() function
>>> import numpy as np
>>> a = np.array([3, 5, 6, 7])
>>> print(np.sum(a))
21
```

$$a = [3 \ 5 \ 6 \ 7]$$

$$\text{result} = [3 + 5 + 6 + 7]$$

$$21$$

Observe that all elements of array are added together to get result 10.

Product

np.prod() function is used to return the product of array elements over a given axis.

Example 3.19 illustrate the use of np.prod() function.

```
>>> # Python code to illustrate np.prod() function
>>> import numpy as np
>>> a = np.array([1, 2, 3, 4])
>>> print(np.prod(a))
24
```

>>> All elements of the array are multiplied together to return value 24.

$$a = [1 \ 2 \ 3 \ 4]$$

$$\text{result} = [1 \times 2 \times 3 \times 4]$$

$$24$$

Mean

The function `np.mean()` returns the arithmetic mean along the specified axis.

Example 3.20 illustrate the use of `mean()` function.

```
>>> # Python code to illustrate np.mean() function
>>> import numpy as np
>>> a = np.array([5, 6, 3, 4])
>>> print(np.mean(a))
4.5
```

>>> Observe that all 4 elements of the array are added together and then it is divided by 4 to get the result.

$$a = [1 \ 2 \ 3 \ 4]$$
$$\text{result} = \mu = \frac{\sum X_i}{N}$$
$$\left[\frac{3 + 4 + 5 + 6}{4} \right]$$
$$4.5$$

Standard Deviation

The function `np.std(a)` returns the standard deviation of the given array elements along the specified axis. Standard Deviation (SD) is measured as the spread of data distribution in the given data set.

Example 3.21 illustrate the use of `np.std()` function.

```
>>> # Python code to illustrate np.std() function
>>> import numpy as np
>>> a = np.array([1, 2, 3, 4])
>>> # Output
>>> print(np.std(a))
1.118033988749895
>>>
```

$$a = [1 \ 2 \ 3 \ 4]$$
$$\text{result} = \sigma^2 = \frac{\sum (X_i - \mu)^2}{N}$$
$$\sigma^2 = \frac{2.25 + 0.25 + 0.25 + 2.25}{4}$$
$$\sigma^2 = \frac{5}{4} = 1.25$$
$$\sigma = \sqrt{1.25} \approx 1.118$$

Cumulative Sum

The function `np.cumsum(a)` returns the cumulative sum of the elements along a given axis.

Example 3.22 illustrate the use of numpy cumulative sum function.

```
>>> # Python code to illustrate np.cumsum() function
>>> import numpy as np
>>> a = np.array([1, 2, 3, 4])
>>> # Output
>>> print(np.cumsum(a))
[ 1  3  6 10]
>>>
```

```
a = [1 2 3 4]
result = [1 3 6 10]
```

Cumulative Product

The function `np.cumprod(a)` is used to return cumulative products of elements along a given axis.

Example 3.23 illustrate the use of numpy cumulative product function.

```
>>> # Python code to illustrate np.cumprod() function
>>> import numpy as np
>>> a = np.array([1, 2, 3, 4])
>>> # Output
>>> print(np.cumprod(a))
[ 1  2  6 24]
>>>
```

```
a = [1 2 3 4]
result = [1 2 6 24]
```

Matrix Operations

For matrix-specific operations, NumPy provides a specialized function `np.dot()` for dot Product. It can handle 2D arrays but considers them as matrices and will perform matrix multiplication. For N dimensions it is a sum-product over the last axis of a and the second-to-last of b.

Example 3.24 illustrate the matrix operations .

```
>>> # Python code to illustrate matrix operations
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6], [7, 8]])
```

```
>>> result = np.dot(a, b)
>>> print(result)
[[19 22]
 [43 50]]
```

These arithmetic operations make NumPy highly versatile for mathematical and scientific computations.

```
a =  $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ 
b =  $\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$ 
result =  $\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$ 
```

Trigonometric Functions

NumPy provides a comprehensive set of trigonometric functions to perform operations involving angles and trigonometric calculations. Here's a detailed explanation of these functions:

Basic Trigonometric Functions

These functions compute the trigonometric values of each element in an array. Angles must be in radians unless specified otherwise.

1. Sine

Sine function computes the sine of each element in the array. The function used is `np.sin()`.

Example 3.27 illustrate to compute Sine using `np.sin()` function.

```
>>> # Python code to illustrate the Sine function
>>> import numpy as np
>>> angles = np.array([0, np.pi/2, np.pi])
>>> result = np.sin(angles)
>>> print('Output \n',result)
```

Output

```
[0.00000000e+00 1.00000000e+00 1.2246468e-16]
```

Observe that in the result the first value is 0, second value is 1 and third value is close to 0.

2. Cosine

Cosine function computes the cosine of each element. The function used is `np.cos()`.

Example 3.26 illustrate the Cosine function.

Example 3.27 illustrate to compute Cosine using `np.cos()` function.

```
>>> # Python code to illustrate the Cosine function
>>> import numpy as np
>>> angles = np.array([0, np.pi/2, np.pi])
>>> result = np.cos(angles)
>>> print('Output \n',result)
```

Output

```
[ 1.0000000e+00  6.123234e-17 -1.0000000e+00]
```

Observe that in the result the first value is 1, second value is close to 0 and third value is -1.

3. Tangent

Tangent function computes the tangent of each element. The function used is `np.tan()`.

Example 3.27 illustrate to compute Tangent using `np.tan()` function.

```
>>> # Python code to illustrate the Tangent function
>>> angles = np.array([0, np.pi/2, np.pi])
>>> result = np.tan(angles)
>>> print('Output \n',result)
```

Output

```
[ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

>>> Observe that in the result the first value is 0, second value is 1 and third value is close to 0.

Inverse Trigonometric Functions

These functions compute the inverse trigonometric values, returning angles in **radians**.

1. Arcsine

This function computes the arcsine of each element. The function used is `np.arcsin()`.

Example 3.28 illustrate to compute Arcsine using `np.arcsin()` function.

```
>>> # Python code to illustrate the Arcsine function
>>> values = np.array([0,1,-1])
>>> result = np.arcsin(values)
>>> print('Output \n',result)
```

Output

```
[ 0.          1.57079633 -1.57079633]
```

```
>>>
```

2. Arccosine

This function computes the arccosine of each element. The function used is `np.arccos()`.

Example 3.29 illustrate to compute Arccosine using `np.arccos()` function.

```
>>> # Python code to illustrate the Arcsine function
```

```
>>> values = np.array([0,1,-1])
```

```
>>> result = np.arccos(values)
```

```
>>> print('Output \n',result)
```

Output

```
[1.57079633 0.          3.14159265]
```

```
>>>
```

3. Arctangent

This function computes the arctangent of each element. The function used is `np.arctan()`.

Example 3.30 illustrate to compute Arctangent using `np.arctan()` function.

```
>>> # Python code to illustrate the Arctangent function
```

```
>>> values = np.array([0,1,-1])
```

```
>>> result = np.arctan(values)
```

```
>>> print('Output \n',result)
```

Output

```
[ 0.          0.78539816 -0.78539816]
```

```
>>>
```

4. Arctangent2 (Two-argument Arctangent)

This function computes the arctangent of y/x considering the quadrant of the point (x,y) . The function used is `np.arctan2()`.

Example 3.31 illustrate to compute Arctangent with two arguments using `np.arctan2()` function.

```
>>> # Python code to illustrate the Arctangent function with two arguments
```

```
>>> y = np.array([1, -1])
```

```
>>> x = np.array([1, 1])
```

```
>>> result = np.arctan2(y, x)
```

```
>>> print('Output \n',result)
```

Output

```
[ 0.78539816 -0.78539816]
```

```
>>>
```

Hyperbolic Trigonometric Functions

These functions compute hyperbolic trigonometric values.

1. Hyperbolic Sine

This function is used to compute hyperbolic sine. The function used is `np.sinh()`.

Example 3.32 illustrate to compute Hyperbolic sine using using `np.sinh()` function.

```
>>> # Python code to illustrate Hyperbolic Sine function
```

```
>>> values = np.array([0, 1, -1])
```

```
>>> result = np.sinh(values)
```

```
>>> print('Output \n',result)
```

Output

```
[ 0.          1.17520119 -1.17520119]
```

```
>>>
```

2. Hyperbolic Cosine

This function is used to compute hyperbolic cosine. The function used is `np.cosh()`.

Example 3.33 illustrate to compute Hyperbolic Cosine using using `np.cosh()` function.

```
>>> # Python code to illustrate Hyperbolic Cosine function
```

```
>>> values = np.array([0, 1, -1])
```

```
>>> result = np.cosh(values)
```

```
>>> print('Output \n',result)
```

Output

```
[1.          1.54308063  1.54308063]
```

```
>>>
```

3. Hyperbolic Tangent

This function is used to compute hyperbolic tangents. The function used is `np.tanh()`.

Example 3.34 illustrate to compute Hyperbolic tangent using using `np.tanh()` function.

```
>>> # Python code to illustrate Hyperbolic Tangent function
```

```
>>> values = np.array([0, 1, -1])
```

```
>>> result = np.tanh(values)
```

```
>>> print('Output \n',result)
```

Output

```
[ 0.          0.76159416 -0.76159416]
>>>
```

4. Inverse Hyperbolic Functions

The function `np.arcsinh()` is an inverse sine hyperbolic function. The function used is `np.arcsinh()`. The function `np.arccosh()` is an inverse cosine hyperbolic function. The function used is `np.arccosh()`. The function `np.arctanh()` is an inverse tangent hyperbolic function. The function used is `np.arctanh()`.

Example 3.35 illustrate to compute Inverse Hyperbolic sine using using `np.arcsinh()` function.

```
>>> # Python code to illustrate Inverse Hyperbolic Tangent function
>>> values = np.array([0, 1, -1])
>>> result = np.arcsinh(values)
>>> print('Output \n',result)
Output
[ 0.          0.88137359 -0.88137359]
>>>
```

Angle Conversion

NumPy includes functions to convert between radians and degrees.

1. Convert Degrees to Radians

The Function `np.radians()` is used to convert degrees to radians.

Example:

```
>>> # Python code to convert Degrees to Radians
>>> degrees = np.array([0, 90, 180])
>>> result = np.radians(degrees)
>>> print('Output \n',result)
Output
[0.          1.57079633  3.14159265]
>>>
```

2. Convert Radians to Degrees

The function `np.degrees()` is used to convert radians to degrees.

Example 3.36 illustrate to convert radians to degrees using `np.degrees()` function.

```
>>> # Python code to convert Radians Degrees
>>> radians = np.array([0, np.pi/2, np.pi])
>>> result = np.degrees(radians)
>>> print('Output \n',result)
Output
```

```
[ 0.  90. 180.]
```

```
>>>
```

Other Trigonometric Utilities

1. Compute Hypotenuse

This function computes the hypotenuse of a right triangle given the lengths of the two perpendicular sides. The function used is `np.hypot()`.

Example 3.37 illustrate to compute Hypotenuse using `np.hypot()` function.

```
>>> # Python code to compute Hypotenuse
```

```
>>> x = np.array([3, 5])
```

```
>>> y = np.array([4, 12])
```

```
>>> result = np.hypot(x, y)
```

```
>>> print('Output \n',result)
```

Output

```
[ 5. 13.]
```

These trigonometric functions make NumPy a powerful tool for mathematical computations involving angles and periodic functions.

Module 3. Data Analysis

Session 1. Introduction to Pandas

Pandas is a powerful and popular open-source data analysis and manipulation library in Python. It is built on top of NumPy and provides easy-to-use data structures and data analysis tools. Pandas is widely used in data science, machine learning, and analytics due to its efficiency and flexibility.

The term Pandas is derived from “Panel Data System”, which is an econometric term for multidimensional, structured dataset.

Pandas are an Open Source, library specially built for Python Programming language. Pandas offer high performance, easy to use data structure and data analysis tools for real world need of individual or any organization. The main author of Pandas is Wes McKinney.

Key Features of Pandas

- Pandas, is the most popular library in Scientific Python ecosystem for data analysis.
- Quick and efficient data manipulation and analysis.
- It has functionality to find and fill missing data.
- It allows you to apply operations to independent groups within the data.
- It supports reshaping of data into different forms.
- It supports visualization by integrating matplotlib.
- Pandas is best for handling huge tabular (like excel, mysql) data sets comprising different data formats.
- Pivoting and reshaping data sets
- Easy handling of missing data (represented as NaN) in both floating point and non-floating-point data.
- Represents the data in tabular form.
- It provides time-series functionality.
- Effective grouping by functionality for splitting, applying, and combining data sets.

Overview of Pandas in Data Analysis

Pandas are used in data analysis for following 4 reasons.

1. Easy-to-use syntax for data manipulation and transformation.
2. Efficient handling of large datasets.

3. Seamless integration with other Python libraries such as Matplotlib and NumPy.
4. Broad support for varied data formats such as CSV, SQL and Excel.

Pandas optimizes memory usage and computation speed. Supports filtering, aggregating, and transforming data with minimal code. Pandas' intuitive syntax makes complex tasks straightforward. Data loading and storage is easy in Pandas. Data exploration, data cleaning, data transformation, data merging, data aggregation and grouping can be done in Pandas with ease. The time series analysis can also be done in Pandas.

Difference between Numpy and Pandas

NumPy and Pandas are two of the most widely used libraries in Python for data manipulation and analysis. While they have overlapping functionalities, they are designed for different purposes and have distinct features. Below is a detailed comparison:

Key Differences Between NumPy and Pandas

Feature	NumPy	Pandas
Primary Purpose	Numerical computations with n-dimensional arrays.	Data manipulation and analysis with labeled data structures.
Data Structure	ndarray (multi-dimensional array).	Series (1D) and DataFrame (2D).
Flexibility	Efficient for numeric and homogeneous data.	Handles mixed data types (e.g., numeric, strings, and dates).
Data Indexing	Indexed by integer positions.	Indexed with labels (row/column names) and integers.
Ease of Use	Requires more manual work for data manipulation.	Provides high-level methods for cleaning, transforming, and exploring data.
File I/O	Limited support for reading/writing files.	Extensive support for formats like CSV, Excel, SQL, and JSON.
Performance	Faster for numerical operations and larger datasets.	Slightly slower due to higher-level abstractions.
Data	Cannot handle missing	Built-in support for missing

Feature	NumPy	Pandas
Handling	data directly (e.g., NaN requires extra handling).	data (e.g., NaN, None).

Installation of Pandas

You can install Pandas using Python's package manager, pip, or via conda if you're using the Anaconda distribution.

Using pip

The simplest way to install Pandas is via pip.

Installation Command

```
pip install pandas
```

Upgradation Command

If Pandas is already installed but you want to update it to the latest version:

```
pip install --upgrade pandas
```

Verify Installation

To ensure Pandas is installed and check its version:

```
python
import pandas as pd
print(pd.__version__)
```

Using conda

If you're using the Anaconda or Miniconda distribution, you can install Pandas via conda.

Installation Command

```
conda install pandas
```

To Update Pandas

```
conda update pandas
```

Installing Specific Versions

If you need a particular version of Pandas:

```
pip install pandas==1.5.3
```

Common Installation Issues

Missing Dependencies: Ensure you have Python installed version 3.7 or later is recommended.

Environment Conflicts: Use a virtual environment to avoid conflicts:

```
python -m venv myenv
source myenv/bin/activate # On macOS/Linux
myenv\Scripts\activate    # On Windows
```

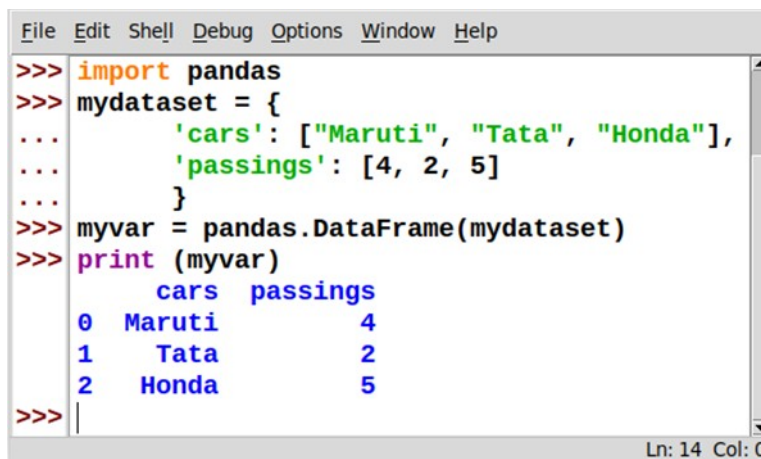
```
pip install pandas
```

Import Pandas

Once Pandas is installed, import it in your applications by adding the import keyword as follows:

```
import pandas
```

Now Pandas are imported and ready to use. Let us test it using the following code.



```
>>> import pandas
>>> mydataset = {
...     'cars': ["Maruti", "Tata", "Honda"],
...     'passings': [4, 2, 5]
... }
>>> myvar = pandas.DataFrame(mydataset)
>>> print (myvar)
   cars  passings
0  Maruti         4
1   Tata         2
2  Honda         5
>>>
```

Pandas as pd

In Python alias are an alternate name for referring to the same thing. Pandas is usually imported under the pd alias. To create an alias with the as keyword while importing as.

```
import pandas as pd
```

Now the Pandas package can be referred to as pd instead of pandas. Here pd is an object of Pandas library to which you can use in your program.

Data Structures in Pandas

Pandas provides two primary data structures that make data manipulation and analysis convenient: Series and Dataframe

1. Series

A Series is a one-dimensional labeled array capable of holding data of any type such as integer, string, and float. It is similar to a column in a spreadsheet or a one-dimensional array in NumPy.

Key Features are labeled indexes similar to dictionary keys and homogeneous data, that is, all elements are of the same type.

2. Dataframe

A Data Frame is a two-dimensional, tabular, labeled data structure. It is analogous to a spreadsheet or SQL table and can hold heterogeneous data types across columns.

Key Features are labeled rows and columns where each column can have a different data type. And flexible indexing for rows and columns.

Series			Series			DataFrame	
	apples			oranges		apples	oranges
0	3		0	0		0	3
1	2	+	1	3	=	1	2
2	0		2	7		2	0
3	1		3	2		3	1

Sample Fig

Comparison of Series and DataFrame

Feature	Series	DataFrame
Dimensions	One-dimensional	Two-dimensional
Indexing	Single index	Row and column indexing
Data Type	Homogeneous (same type for all data)	Heterogeneous (different types for columns)
Structure	Similar to a single column or array	Similar to a spreadsheet or SQL table

Apart from series and dataframe we can also have index and panel as data structures in Pandas.

3. Index (Shared by Both Series and DataFrame)

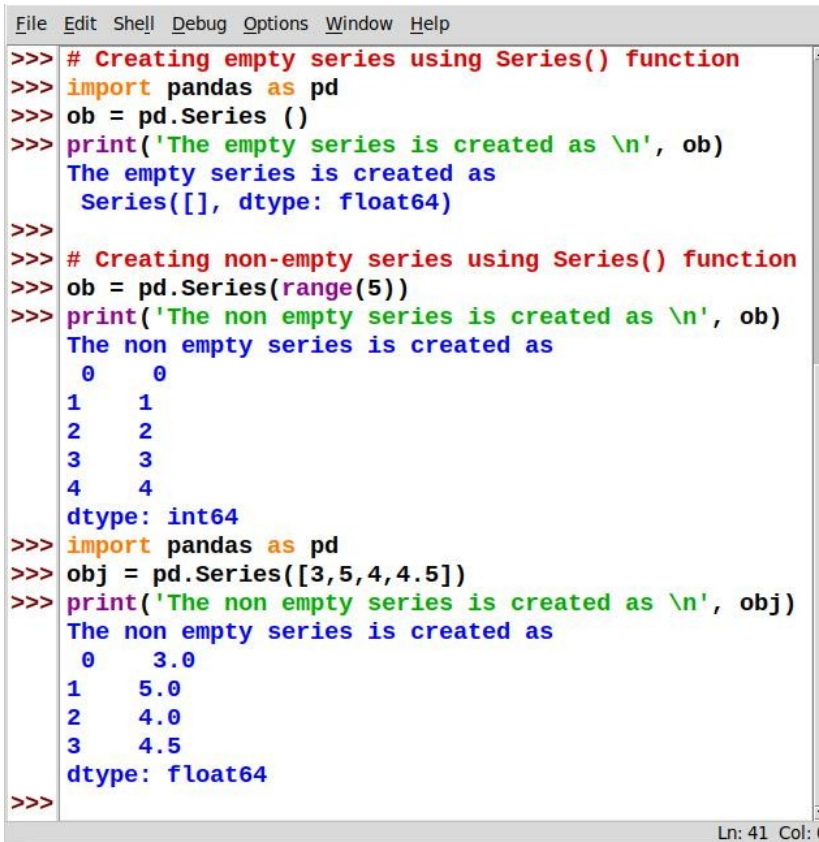
The Index in Pandas provides labels for rows and/or columns, allowing for easy data alignment and access.

4. Panel (Deprecated)

Pandas previously had a Panel data structure for three-dimensional data. It has been deprecated in favor of using MultiIndex DataFrames or external libraries like xarray.

Session 2. Coding with Pandas

Now we can start writing small codes in Pandas as given below:



```
>>> # Creating empty series using Series() function
>>> import pandas as pd
>>> ob = pd.Series ()
>>> print('The empty series is created as \n', ob)
The empty series is created as
Series([], dtype: float64)

>>> # Creating non-empty series using Series() function
>>> ob = pd.Series(range(5))
>>> print('The non empty series is created as \n', ob)
The non empty series is created as
0    0
1    1
2    2
3    3
4    4
dtype: int64

>>> import pandas as pd
>>> obj = pd.Series([3,5,4,4.5])
>>> print('The non empty series is created as \n', obj)
The non empty series is created as
0    3.0
1    5.0
2    4.0
3    4.5
dtype: float64

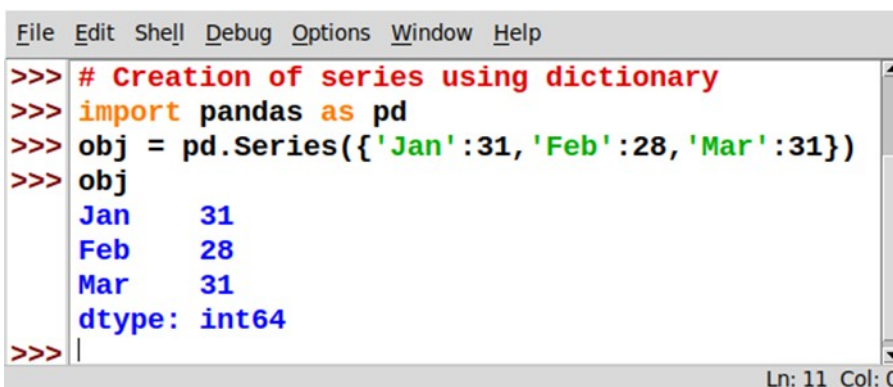
>>>
```

Here in this list, three values are integers and one is float type, so finally the series type will be float.

Creation of series using dictionary

Example 2.1 illustrates how to create a series with the help of a dictionary.

Dictionary's keys act as a series index and the dictionary's value's act as a series value.



```
>>> # Creation of series using dictionary
>>> import pandas as pd
>>> obj = pd.Series({'Jan':31, 'Feb':28, 'Mar':31})
>>> obj
Jan    31
Feb    28
Mar    31
dtype: int64

>>> |
```

In this example, “Jan”, “Feb”, “Mar” is called series index and 31, 28, 31 are as series values and the type of series is int type.

Creation of series with Scalar value

Let us understand different ways to create a series using the pandas library using the following code.

Example 2.2

```
File Edit Shell Debug Options Window Help
>>> # Creation of series with Scalar value
>>> import pandas as pd
>>> a=pd.Series(10,index=range(0,3))
>>> a
0      10
1      10
2      10
dtype: int64
>>>
```

Ln: 11 Col: 0

In this code snippet, the range (0, 3) function will generate indexes 0, 1 and 2 given by the programmer. There is only one value “10” which will store in all indexes.

Example 2.3

```
File Edit Shell Debug Options Window Help
>>> b=pd.Series(15,index=range(1,6,2))
>>> b
1      15
3      15
5      15
dtype: int64
>>>
```

Ln: 19 Col: 0

In this code snippet, range () functions are used to create index 1, 3 and 5. Here the index range is 1 to 6, which will start from 1 and increment by 2 and goes up to 5 only. There is only one value “15” given by the programmer which will store in all indexes.

Example 2.4

```
File Edit Shell Debug Options Window Help
>>> a=pd.Series('Welcome to My School',
index=['Hema', 'Rahul','Anup'])
>>> a
Hema      Welcome to My School
Rahul     Welcome to My School
Anup      Welcome to My School
dtype: object
>>>
```

Ln: 39 Col: 0

Indexes can also be given in the form list that is 'Hema', 'Rahul', 'Anup'. This is given by the programmer and there is only one value "Welcome to CIVE" which will store in all indexes.

Example 2.5 demonstrate the use of Pandas library to demonstrate mathematical function/Expression in series.

```
File Edit Shell Debug Options Window Help
>>> import pandas as pd
>>> import numpy as np
>>> a=np.arange(9,13)
>>> a
array([ 9, 10, 11, 12])
>>> |
Ln: 52 Col: 0
```

arange() is numpy library function which stored 9 to 12 numbers into "a" object and we can use this "a" object values as series index and "data= a * 3" for data values as shown in **Example 2.6**.

Example 2.6

```
File Edit Shell Debug Options Window Help
>>> import pandas as pd
>>> import numpy as np
>>> a=np.arange(9,13)
>>> a
array([ 9, 10, 11, 12])
>>> od=pd.Series(index=a, data=a*3)
>>> od
9      27
10     30
11     33
12     36
dtype: int64
>>> |
Ln: 16 Col: 0
```

Series Object Attributes

Series Attribute	Description
Series.index	Returns the index of a series
Series.values	Returns values of series in the form of ndarray.
Series.dtype	Returns the data type of the data object
Series.shape	Returns tuple of the shape of underlying data
Series.nbytes	Return number of bytes of underlying data. The formula is: number of element in series * data type size.

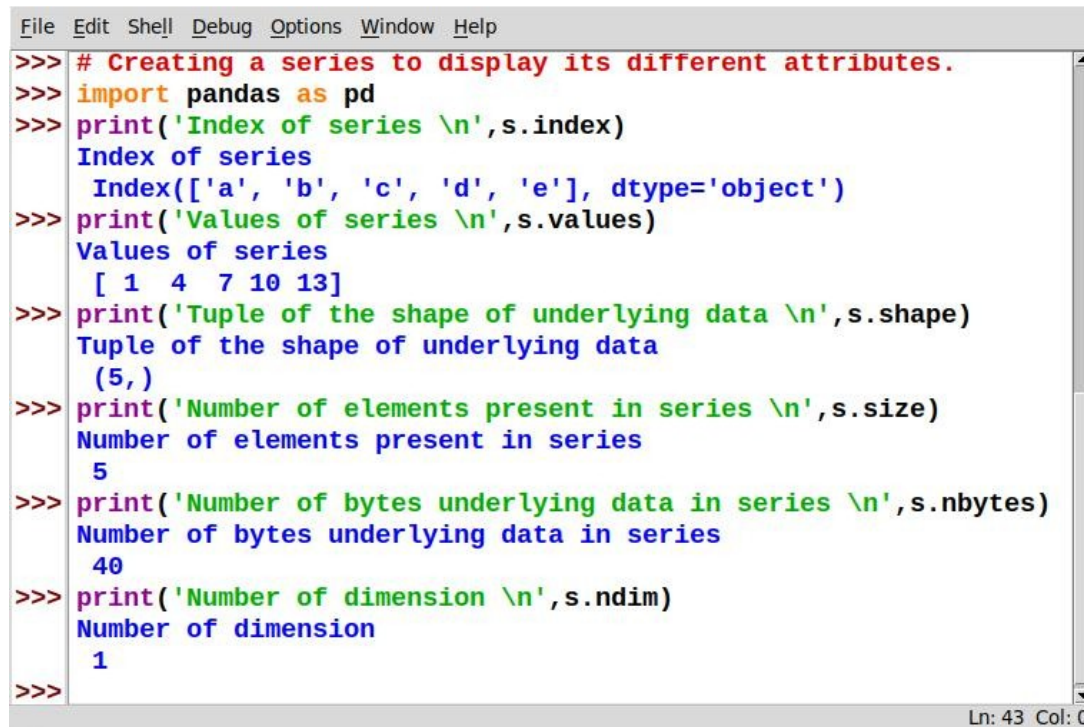
Series.ndim	Returns the number of dimensions in series.
Series.size	Returns number of elements present in series

Example 2.7

Let us create a series to display its different attributes.

Create a series s:

```
s=pd.Series(range (1,15,3), index=[x for x in 'abcde'])
```



```
File Edit Shell Debug Options Window Help
>>> # Creating a series to display its different attributes.
>>> import pandas as pd
>>> print('Index of series \n',s.index)
Index of series
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
>>> print('Values of series \n',s.values)
Values of series
[ 1  4  7 10 13]
>>> print('Tuple of the shape of underlying data \n',s.shape)
Tuple of the shape of underlying data
(5,)
>>> print('Number of elements present in series \n',s.size)
Number of elements present in series
5
>>> print('Number of bytes underlying data in series \n',s.nbytes)
Number of bytes underlying data in series
40
>>> print('Number of dimension \n',s.ndim)
Number of dimension
1
>>>
```

Here,

s.index: It will display index of the series i.e. Index(['a', 'b', 'c', 'd', 'e'] dtype = 'object')

s.values: It will display values of the series i.e. array([1,4,7,10,13], dtype = int64).

s.shape: It will display tuple of the shape of underlying data i.e. (5,)

s.size: It will display number of elements present in series i.e. 5

s.nbytes: It will display number of bytes of underlying data(formula is: number of element present in series * (multiply by) size of individual element) i.e. 40

s.ndim: It will display the number of dimension(1-D, 2-D) i.e. 1

Consider another example,

Example 2.8

```

>>> s=pd.Series([1,2,3,4,5])
>>> print('Index of series\n',s.index)
Index of series
RangeIndex(start=0, stop=5, step=1)
>>> print('Values of series\n',s.values)
Values of series
[1 2 3 4 5]
>>> print('Bytes of series\n',s.nbytes)
Bytes of series
40
>>> print('Dimension of series\n',s.ndim)
Dimension of series
1
>>> _

```

In the above example:

s.index: It will display index of the series

s.values: It will display values of the series i.e. array([1,2,3,4,5]).

s.nbytes: It will display number of bytes of underlying data(formula is: number of element

present in series * (multiply by) size of individual element) i.e. 40

s.ndim: It will display the number of dimension(1-D, 2-D) i.e. 1

It is possible to display series data values as per user need. It is known as Series Slicing. For this we need to pass three parameters i.e. [<start>:<stop>:<step>] It is illustrated in the **Example 2.9**.

Example 2.9

```
File Edit Shell Debug Options Window Help
>>> import pandas as pd
>>> import numpy as np
>>> a=np.arange(9,13)
>>> ob=pd.Series(index=a, data=a**2)
>>> print('Values in the series are \n',ob)
Values in the series are
 9      81
10     100
11     121
12     144
dtype: int64
>>> print('Value at index 10 are \n',ob[10])
Value at index 10 are
100
>>> print('Value at index 10 are \n',ob[2:4])
Value at index 10 are
11     121
12     144
dtype: int64
>>> print('Series values starting at index 1 are \n',ob[1:])
Series values starting at index 1 are
10     100
11     121
12     144
dtype: int64
>>> print('Series values starting at index 0 upto with step 2\n',ob[0::2])
Series values starting at index 0 upto with step 2
 9      81
11     121
dtype: int64
>>> print('Series values in reverse order\n',ob[::-1])
Series values in reverse order
12     144
11     121
10     100
 9      81
dtype: int64
>>>
```

In the above example,

ob – It will display all values in the series.

Ob[10] – It will display the value at index 10 means 100.

ob[2 : 4] – It will display values 121, 144 at 11, 12 as per given index by user. Python automatically generates index 0, 1, 2, 3 internally against user's indexes 9, 10, 11 and 12. So in this code snippet indexes will be treated as 2, 3 means 11 and 12 only. Index no 4 is not included here.

ob[1:] – It will display series values which start with “1” in index up-to end in normal order i.e. 100, 121 and 144

ob[0 : 2] – Here the starting index is 0, there is no stop point but step is 2. So, it will display alternative values 81 and 121.

ob[: -1] – It will display series values in reverse order, because there is no start and stop point. The step is given as -1 means the index will increase negatively. This is similar to the list slicing concept.

Example 2.10: Write a python code to modify/update a data series.

```
File Edit Shell Debug Options Window Help
>>> # Example to modify/update a data series
>>> import pandas as pd
>>> s=pd.Series(range(1,15,3),index=[x for x in 'abcde'])
>>> print('The data series is created with the elemens as \n',s)
The data series is created with the elemens as
a      1
b      4
c      7
d     10
e     13
dtype: int64
>>> s['c']=25
>>> print('The data series after replacing the value of c to 25\n',s)
The data series after replacing the value of c to 25
a      1
b      4
c     25
d     10
e     13
dtype: int64
>>>
```

Ln: 46 Col: 0

Observe that the value at index 'c' is modified to 25.

Example 2.11: Write a python code to modify / update indexes of a data series.

```
File Edit Shell Debug Options Window Help
>>> # Example to modify/update a data series
>>> import pandas as pd
>>> s=pd.Series(range(1,15,3),index=[x for x in 'abcde'])
>>> print('The data series is created with the elemens as \n',s)
The data series is created with the elemens as
a      1
b      4
c      7
d     10
e     13
dtype: int64
>>> s.index=['u', 'v', 'w', 'x', 'y']
>>> print('The data series after modifying index is\n',s)
The data series after modifying index is
u      1
v      4
w      7
x     10
y     13
dtype: int64
>>>
```

Ln: 88 Col: 0

Here in this example, the series is created with the index using a for loop as ['a', 'b', 'c', 'd', 'e']. Then the index of the series is changed to ['u', 'v', 'w', 'x', 'y']. So the new indexes in series are u, v, w, x and y.

head() and tail () Function

Python provides two important functions to access the data values from the series directly. You can access starting values using head() function and or last values using tail() function.

head (<n>): The head () function is used to display first/top n rows from a Series. By default it will display the top/first 5 rows.

tail(<n>): The tail () function is used to display last/bottom n rows from a Series. By default it will display the last / bottom 5 rows.

Example 2.12: Program illustrates the use of head() and tail() functions.

```
File Edit Shell Debug Options Window Help
>>> # Program to illustrate the head() and tail() function
>>> import pandas as pd
>>> s=pd.Series(range(1,20,2),index=[x for x in range(0,10)])
>>> print('Display the series values from 1 to 20 with step 2\n',s)
Display the series values from 1 to 20 with step 2
  0    1
  1    3
  2    5
  3    7
  4    9
  5   11
  6   13
  7   15
  8   17
  9   19
dtype: int64
>>> print('Display default top values of series\n',s.head())
Display default top values of series
  0    1
  1    3
  2    5
  3    7
  4    9
dtype: int64
>>> print('Display top 3 values of series \n',s.head(3))
Display top 3 values of series
  0    1
  1    3
  2    5
dtype: int64
>>> print('Display default bootom values of series\n',s.tail())
Display default bootom values of series
  5   11
  6   13
  7   15
  8   17
  9   19
dtype: int64
>>> print('Display bottom 2 values of series \n',s.tail(2))
Display bottom 2 values of series
  8   17
  9   19
dtype: int64
>>>
```

Vector Operations and Arithmetic Operations on series

You can do various arithmetic operations on series data in python just like you perform it on normal variables.

Example 2.13: Program illustrates vector operations and arithmetic operations in series. For this first create the three series as shown in the following code.

```
File Edit Shell Debug Options Window Help
>>> import pandas as pd
>>> s1=pd.Series(range(11,18))
>>> s2=pd.Series(range(41,48))
>>> s3=pd.Series(range(101,106), index=[10,20,30,40,50
])
>>> print('Series s1 created with values ranging from
11 to 18 from index 0 to 6 \n',s1)
Series s1 created with values ranging from 11 to 1
8 from index 0 to 6
  0    11
  1    12
  2    13
  3    14
  4    15
  5    16
  6    17
dtype: int64
>>> print('Series s2 created with values ranging from
41 to 48 from index 0 to 6 \n',s2)
Series s2 created with values ranging from 41 to 4
8 from index 0 to 6
  0    41
  1    42
  2    43
  3    44
  4    45
  5    46
  6    47
dtype: int64
>>> print('Series s3 created with values ranging from
101 to 106 with index as 10,20,30,40,50\n',s3)
Series s3 created with values ranging from 101 to
106 with index as 10,20,30,40,50
  10    101
  20    102
  30    103
  40    104
  50    105
dtype: int64
Ln: 37 Col: 26
```

Now perform arithmetic operation (+, -, * and /) on s1 and s2 as the indexes are the same in both but not using s3 as it has different indexes.

Example 2.14 illustrates the arithmetic operation and vector operation in series.

```
File Edit Shell Debug Options Window Help
>>> # Arithmetic operation of adding elements of series s1 and s2
>>> print('Arithmetic operation to add the elements of series s1 and s2 to create a new series \n', s1+s2)
Arithmetic operation to add the elements of series s1 and s2 to create a new series
  0    52
  1    54
  2    56
  3    58
  4    60
  5    62
  6    64
dtype: int64
>>> # Vector operation of adding a vector value to series s1
>>> print('Vector value 2 is added to series s1 to form a new series \n', s1+2)
Vector value 2 is added to series s1 to form a new series
  0    13
  1    14
  2    15
  3    16
  4    17
  5    18
  6    19
dtype: int64
>>> # Vector operation to multiply each elements of series s1 by 2
>>> print('Vector value 2 is multiplied to series s1 to form a new series \n', s1*2)
Vector value 2 is multiplied to series s1 to form a new series
  0    22
  1    24
  2    26
  3    28
  4    30
  5    32
  6    34
dtype: int64
>>> # Addition of series with different index is not possible
>>> print('Arithmetic operation to add the elements of series s1 and s3 is not possible due to different index\n', s1+s3)
Arithmetic operation to add the elements of series s1 and s3 is not possible due to different index
  0    NaN
  1    NaN
```

In the above code the arithmetic and vector operations performed on series are as follows.

$s1+s2$: This operation will add each element of series $s1$ and $s2$. It will successfully be done as both the series are similar in nature in terms of their index number.

$s1 + 2$: This operation will add 2 to each item of the data series. So we can get 13, 14, 15, 16, 17, 18, and 19 instead of 11, 12, 13, 14, 15, 16 and 17.

$s1 * 2$: This operation will multiply each item of the data series by 2. So we will get 22, 24, 26, 28, 30, 32 and 34 instead of 11, 12, 13, 14, 15, 16 and 17.

$s1 + s3$: This operation will not do appropriately as both series had different types of indexes. The index of series $s1$ is [0,1,2,3,4,5,6] while series $s3$ has index [10,20,30,40,50] . if indexes are not matched then Python will result in NaN (Not a number) in Output.

Relational Operations on series

It is also possible to perform various relational operations (>, <, >=, <=, ==, !=) on series data in python to generate Boolean results in the form of True/False. These operations are also known as filtration in python.

First create a series and then perform the relational operations and delete data from data Series as shown in **Example 2.15**.

```
File Edit Shell Debug Options Window Help
>>> # Creating a Series to do relational operations
>>> import pandas as pd
>>> s1=pd.Series(data=[12,76,9,34,65])
>>> print('Series s1 created with specified data values with index
from 0 to 4 is \n',s1)
Series s1 created with specified data values with index from 0
to 4 is
 0    12
 1    76
 2     9
 3    34
 4    65
dtype: int64
>>> print('Data values in Series s1 which satisfy the conditon gene
rates the result as True else False\n',s1<50)
Data values in Series s1 which satisfy the conditon generates t
he result as True else False
 0    True
 1   False
 2    True
 3    True
 4   False
dtype: bool
>>> print('Data values in Series s1 which satisfy the conditon gene
rates the result as original values. The data values which dono
t satisfy the condition will not be displayed\n',s1[s1<50])
Data values in Series s1 which satisfy the conditon generates t
he result as original values. The data values which donot satis
fy the condition will not be displayed
 0    12
 2     9
 3    34
dtype: int64
>>> print('Drop function delete data value at the specified index n
umber\n',s1.drop(3))
Drop function delete data value at the specified index number
 0    12
 1    76
 2     9
 4    65
dtype: int64
>>>
```

Ln: 165 Col: 0

Session 3. Data Visualisation using Matplotlib

Data visualization is the graphical representation of data or information. It is used to display data in a more expressive way. Data visualization in the form of charts, graphs, animation, and maps are very easy and simple to understand the trends, outliers, and patterns in data. Data visualization techniques for such big data are very important for the purpose of analysis of data.

Data Visualization is the process of representing data graphically to help users understand patterns, trends, and insights more effectively. It transforms raw data into visual formats like charts, graphs, and maps, making it easier to analyze and communicate information.

Importance of Data Visualization

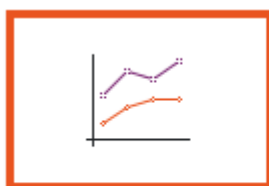
- It simplifies Complex Data, that is, Large datasets become easier to understand.
- It reveals Trends & Patterns, that is, helps identify relationships within the data.
- It improves Decision-Making, that is, Visual insights assist in strategic planning.
- It enhances Communication, that is, makes data accessible to both technical and non-technical users.

Common Types of Data Visualizations

There are different forms of data visualization such as basic graphs and advanced graphs

Basic Charts & Graphs

1. **Bar Chart:** It compares categorical data. A bar plot or bar chart is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent. The bar plots can be plotted horizontally or vertically.
2. **Line Chart:** It shows trends over time. Line charts are used to represent the relation between two data X and Y on a different axis.
3. **Pie Chart:** It represents proportions of a whole. A Pie Chart is a circular statistical plot that can display only one series of data. The area of the chart is the total percentage of the given data. The area of slices of the pie represents the percentage of the parts of the data. The slices of pie are called wedges.



Line chart



Bar Chart



Pie Chart

Fig. 3.1: Basic Charts (a) Bar Chart (b) Line Chart (c) Pie Chart

4. Advanced Visualizations

Maps: It is used for geographic data such as heatmaps. It uses colors, symbols, and lines to represent different data points on a geographical map.

Scatter Plot: It displays relationships between two variables. A scatter plot is a graph that shows the relationship between two variables in a data set. It uses Cartesian coordinates to plot data points on a two-dimensional plane.

Histogram: It represents data distribution. A histogram is a graph that shows how quantitative data is distributed. It's made up of bars that represent the number of values that fall into each interval.

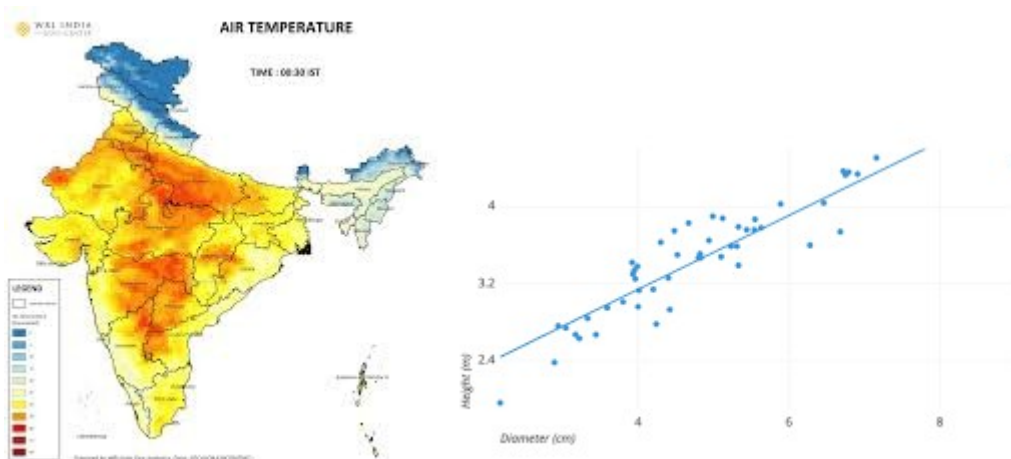
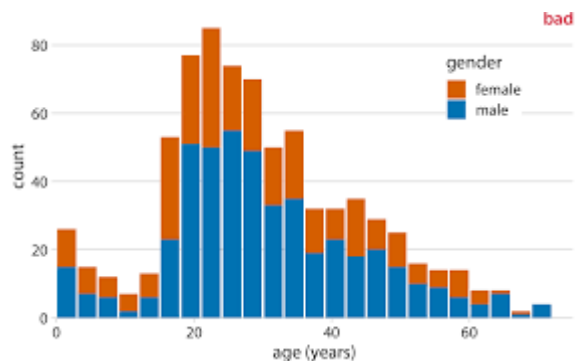


Fig. 3.2 : Advanced Visualizations (a) Map (b) Scatter Plot



(c) Histogram

Data Visualization Tools

Some popular data visualization tools are as given below:

1. Matplotlib (Python) – For static plots.
2. Tableau & Power BI – Interactive dashboards.
3. Google Data Studio – Web-based visualization.
4. D3.js – JavaScript library for dynamic visualizations.

In this session we will discuss the Matplotlib tool available with Python.

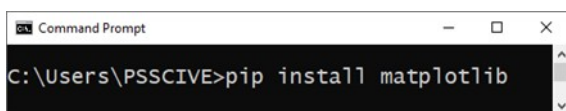
Matplotlib

The Matplotlib is a python library that provides many interfaces and functionality for 2D-graphics similar to MATLAB. Python scripts can be used to create 2D graphs and plots using the Matplotlib module. With features to control line styles, font attributes, formatting axes, and other features, it offers a module named pyplot that makes things simple for plotting. It offers a huge range of graphs and plots, including error charts, bar charts, power spectra, and histograms. It is combined with NumPy to provide a powerful open source MatLab substitute environment.

Installing Matplotlib

To install Matplotlib library, you need to open the command prompt with administrator rights and make sure internet connectivity is on. Matplotlib library and its dependencies can be easily downloaded as a binary file (pre-compiled) package from the internet very easily.

To install Matplotlib in the Windows operating system, issue the following command on the command prompt.



It will give the message for successful installation of Matplotlib library.

To install Matplotlib in Ubuntu Linux run the following command in the command prompt.

```
pip install matplotlib
```

It will start downloading and installing packages related to the matplotlib library.

To verify that matplotlib is successfully installed, execute the following command in the Python idle. If matplotlib is successfully installed, the version of matplotlib installed will be displayed.

To find out version of Matplotlib, open the Python Idle and find the version using the following command. The version of Matplotlib is displayed as '3.5.1'

```
>> import matplotlib
>> matplotlib.__version__
'3.5.1'
```

The version of Matplotlib is displayed as '3.5.1'

You can find what directory Matplotlib is installed in by importing it and printing the `__file__` attribute:

Importing Pyplot

Pyplot is a set of functions in the Matplotlib library. A figure's elements can be changed by using its functions, which include constructing a figure, a plotting area, plot lines, and adding plot labels. Keep in mind that you can alter the line's colour and style by including the arguments `linecolor` and `linestyle`. You need to use the `import` keyword as under to use it.

```
import matplotlib.pyplot as pp
```

Here `pp` is a user defined object for `pyplot`. You can use all the functions in the `pyplot` library using this object as per your need.

CREATING LINE CHART

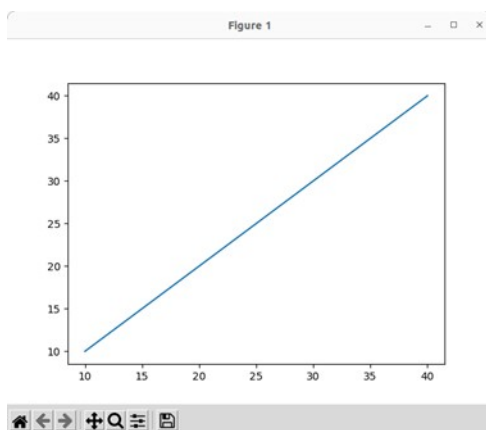
A line chart or line graph is a type of chart which displays information as a series of data points called 'markers' connected by a straight line segment. The `pyplot` interface offers a `plot()` function for creating a line graph.

Example 3.1: Create Line Graph with the help of two given List.

The python code is shown below with the list `a` and `b`. The output as a plotted graph is shown below the program.

```
File Edit Shell Debug Options Window Help
>>> # Create Line Graph with the help of two given List a & b
>>> import matplotlib.pyplot as pp
>>> a=[10,20,30,40]
>>> b=[10,20,30,40]
>>> pp.plot(a,b)
[<matplotlib.lines.Line2D object at 0x7f105d478400>]
>>> print('The line graph is plotted as')
The line graph is plotted as
>>> pp.show()
Ln: 26 Col: 0
```

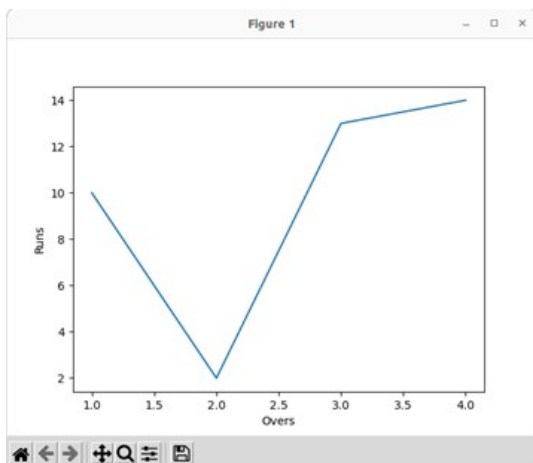
The output is shown with the line graph plotted as below.



Example 3.2: Write a program to plot a Line Graph of number of runs and over provided in two different lists.

The python program and output is given below.

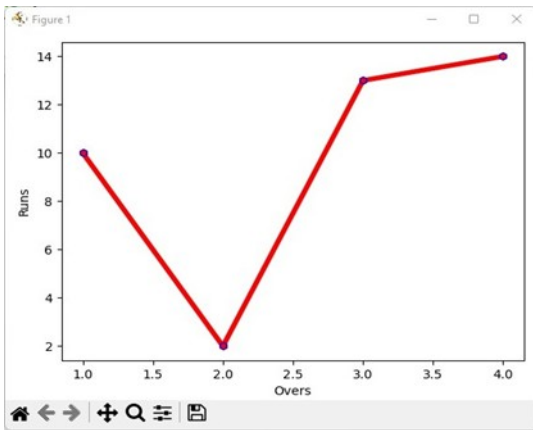
```
File Edit Format Run Options Window Help
# Plot a line graph of runs and over
import matplotlib.pyplot as pp
a=[1,2,3,4]
b=[10,2,13,14]
pp.xlabel("Overs")
pp.ylabel("Runs")
pp.plot(a,b)
pp.show()
Ln: 8 Col: 0
```



Example 3.3 : Let us modify previous examples for changing marker size, edge color and increase the line width using various parameters of plot() function.

```
File Edit Format Run Options Window Help
# Changing marker size, edge color, line width
import matplotlib.pyplot as pp
a=[1,2,3,4]
b=[10,2,13,14]
pp.xlabel("Overs")
pp.ylabel("Runs")
pp.plot(a,b, "r",marker="h",markersize=6,
        markeredgecolor="b",linewidth=4)
pp.show()
Ln: 9 Col: 9
```

Here in this example, the parameters like marker, markersize, markeredgecolor and linewidth are used to give specification in line plot.



The various codes for marker parameters are given below.

Character	Description
'p'	pentagon marker
'*'	star marker
'h'	hexagon1 marker
'H'	hexagon2 marker
'+'	plus marker
'x'	x marker
'D'	diamond marker
'd'	thin_diamond marker
' '	vline marker
'_'	hline marker

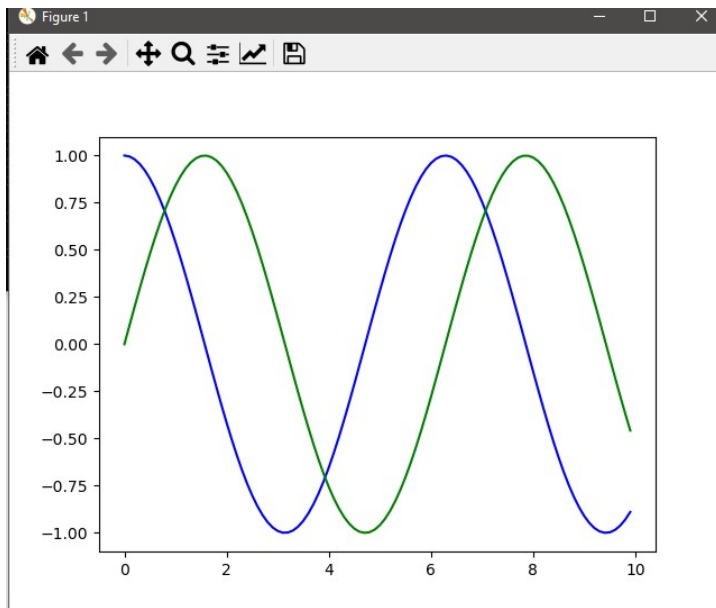
Example 3.4. Write a python code for creating a line chart with different line colour.

In this example we have used the `arange()` function. Variable `z` is initialized with multiple instances of values from 0 to 10 with the interval of 0.1 using this function. These multiple values of `z` will be passed to `sin` and `cos` functions respectively and the result will be stored in variable `a`, `b`.

Now using `pp` object values of `z` will be plotted in a line chart along with `a` and `b` within blue and green colour respectively.

```
File Edit Format Run Options Window Help
# Plot a line graph with different line colour
import matplotlib.pyplot as pp
import numpy as np
z=np.arange(0,10,0.1)
a=np.cos(z)
b=np.sin(z)
pp.plot(z,a,"b")
pp.plot(z,b,"g")
pp.show()
```

Ln: 9 Col: 9



In the above program, the various colour codes as given below can be used while preparing the chart as below.

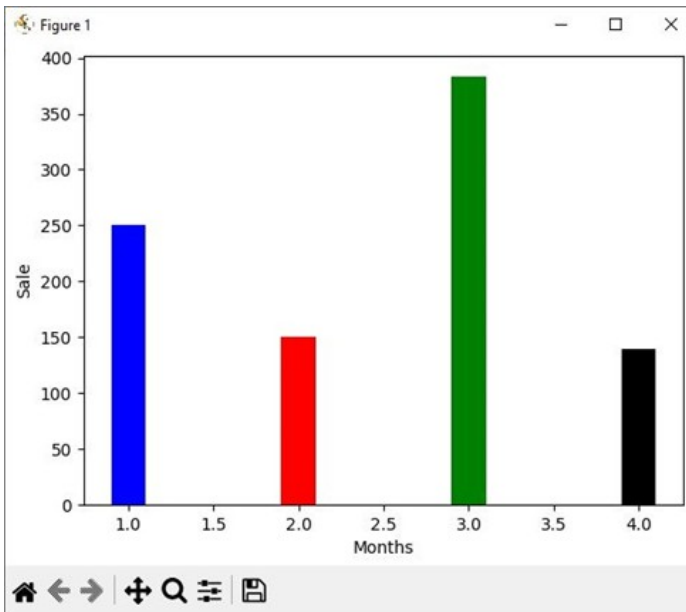
Code	Colour Name
"b"	Blue
"g"	Green
"r"	Red
"c"	Cyan
"m"	Magenta
"y"	Yellow
"k"	Black
"w"	White

CREATING BAR CHART

A Bar Graph/Chart a graphical display of data using bars of different heights. We can use bar() and barh() for this purpose. We can use width and color parameter of bar Graph

Example 3.5: Let us create a bar Graph for monthly sale of an electronic items shop using a list.

```
File Edit Format Run Options Window Help
# Creating bar graph for monthly sale of electronic item
import matplotlib.pyplot as pp
a=[1,2,3,4]
b=[250,150,383,140]
pp.xlabel("Months")
pp.ylabel("Sale")
pp.bar(a,b,width=[1/5],color=['b','r','g','k'])
pp.show()
Ln: 1 Col: 10
```

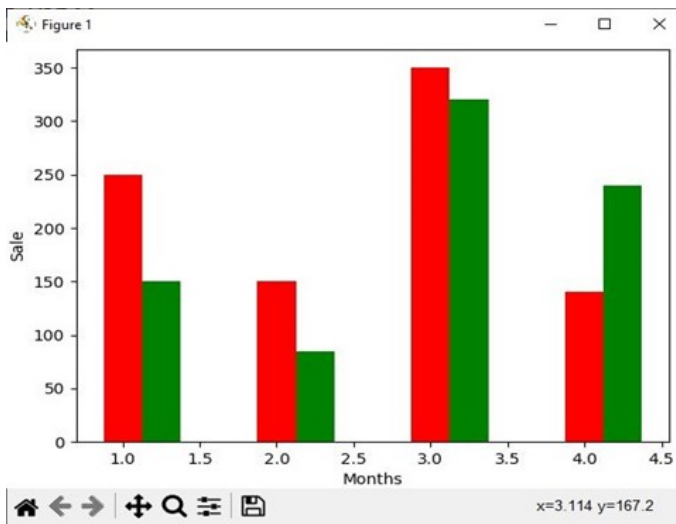



Creating Multiple Bar Chart

Grouped Bar chart is another name for a multiple bar chart. There are several ways to customize a bar plot or bar chart, including multiple bar plots, stacked bar plots, and horizontal bar charts. Typically, multiple bar charts are used to compare multiple items using the chart graphically.

Example 3.6: Let us create multiple Bar Graphs for monthly sale of Keyboard and mouse with different colors.

```
File Edit Format Run Options Window Help
# Creating multiple bar graph for monthly sale
import matplotlib.pyplot as pp
import numpy as np
month=np.arange(1.0,5.0,1.0)
mousesale=[250,150,350,140]
kbsale=[150,85,320,240]
pp.xlabel("Months")
pp.ylabel("Sale")
pp.bar(month,mousesale,color="r",width=0.25)
pp.bar(month+0.25,kbsale,color="g",width=0.25)
pp.show()
Ln: 3 Col: 0
```

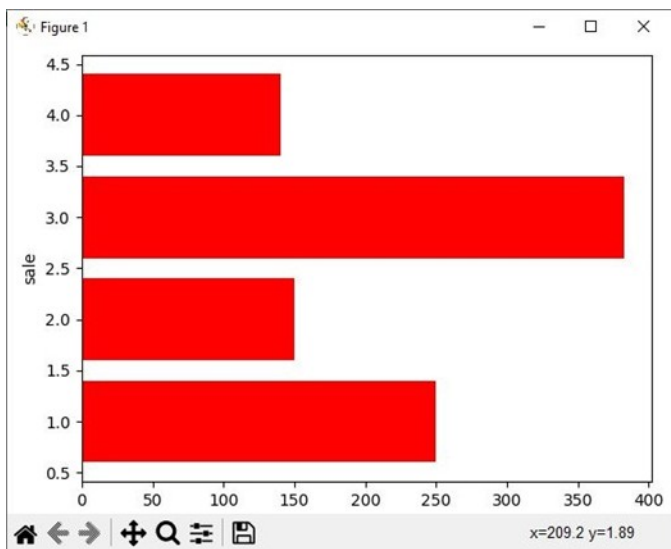


Creating Horizontal Bar Chart

As you have prepared a vertical bar graph in example 5, you can prepare a horizontal bar chart using another bar chart function i.e. `barh()`.

Example 3.7: Modify Example 5 to create Horizontal Bar Chart graph using `barh()` function.

```
File Edit Format Run Options Window Help
# Creating Horizontal bar graph for monthly sale
import matplotlib.pyplot as pp
import numpy as np
month=np.arange(1.0,5.0,1.0)
mousesale=[250,150,350,140]
pp.ylabel("Sale")
pp.barh(month,mousesale,color="r")
pp.show()
Ln: 7 Col: 7
```



CREATING PIE CHART

Pie chart is made up of different parts of a circle where each part shows a particular ratio of data. We can use pie() function to create this type of chart.

The following property we can use with pie Graph like:

Label: Name of Pie

Autopct: Percentage value of Pie in circle

Color: Colours of Pie

Explode: Detached from circle.

Example 3.8: Create a Pie Graph Using salary and name as a list and to use color, autopct, label, explode property.

Program to create a pie graph is given below.

```
File Edit Format Run Options Window Help
# Creating Pie Graph using salary and name
import matplotlib.pyplot as pp
import numpy as np
salary=[12000,15000,8000,7500]
empname=["Amit","Mohan","sagufata","John"]
c=['r','y','b','g']
e=[0,0,0,.3]
pp.pie(salary,colors=c,labels=empname,autopct="%1.1f%%",explode=e)
pp.show()
```

Ln: 9 Col: 9

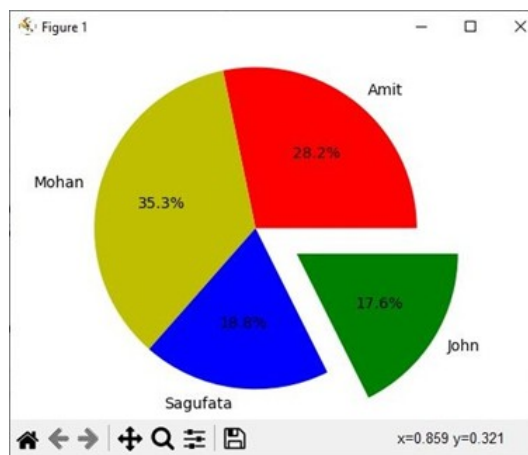


Chart Anatomy and Saving Graph

Every type of chart has a special structure with its contents/values. The values for each bar represent a specific category of data. The y-axis is the vertical axis that runs along either the left or right side of the bar graph. The x-axis is the horizontal axis located at the bottom of a bar graph. The value of the data is represented by the height or length of the bars. The common key points in any bar chart are as under.

Figure – Any chart will be made under this area only. This is the area of plot.

Axes – This is that area which has actual plotting.

Axis Label – This is made up of x-axis and y-axis.

Limits – This is the limit of values marked on x-axis and y-axis.

Tick Marks – This is the individual value on x-axis and y-axis.

Title – It is the text to be shown at the top of the plot.

Legends – This is the set of data of different color which is to be used during plotting.

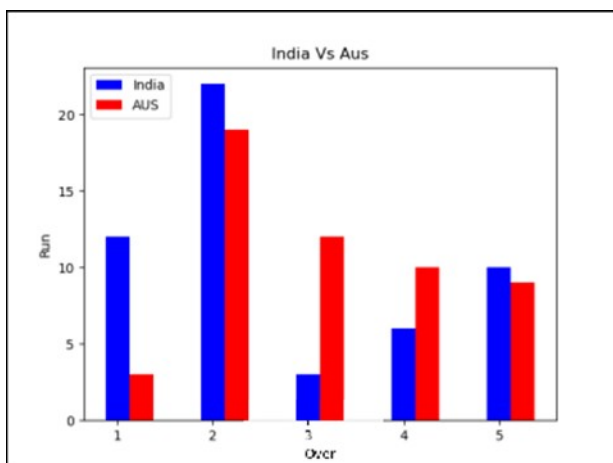
Example 3.9 : Create a graph to show the anatomy of Chart using legend, xlabel, ylabel, title of Chart and understand how to save the graph as image.

```
File Edit Format Run Options Window Help
# Creating a Graph to show anatomy of Chart
import matplotlib.pyplot as pp
import numpy as np
over=np.arange(1,6,1)
India=[12,22,3,6,10]
Aus=[3,19,12,10,9]
pp.title("India Vs Aus")

pp.bar(over, India,color="b",width=0.25,label="India")
pp.bar(over+0.25, Aus,color="r",width=0.25,label="AUS")
pp.legend(loc="upper left")
pp.xlabel("Over")
pp.ylabel("Run")
# Specify the location of computer to save the file
pp.savefig("/home/dds/WinIndia.png")
pp.show()
```

Ln: 14 Col: 51

In this code snippet, pp.savefig() function is used to save the chart as an image. It will save a chart at the given path/location in the function as a parameter.



Module 4. Neural Network

Session 1. Artificial Neural Network (ANN)

1.1 Neural Network

A Neural Network or Artificial Neural Network (ANN), constitutes a network of interconnected neurons. In technical terms, a Neural Network serves as a machine-learning algorithm inspired by the human brain's working. The brain processes information by transmitting signals from one neuron to another, forming neural pathways that dictate functions like memory retention, motor skills, and speech articulation.

Neural networks, or ANN replicates the brain's functionality by integrating data inputs, weights, and bias. These components collaborate to effectively identify, categorize, and describe objects within datasets.

The human brain comprises approximately 86 billion nerve cells, known as neurons, which communicate with thousands of other cells via axons. Dendrites receive stimuli from the external environment or sensory organs, generating electrical impulses that swiftly traverse the neural network. Subsequently, a neuron may transmit the message to another neuron to address an issue or withhold its propagation, as shown in Figure 1.1.

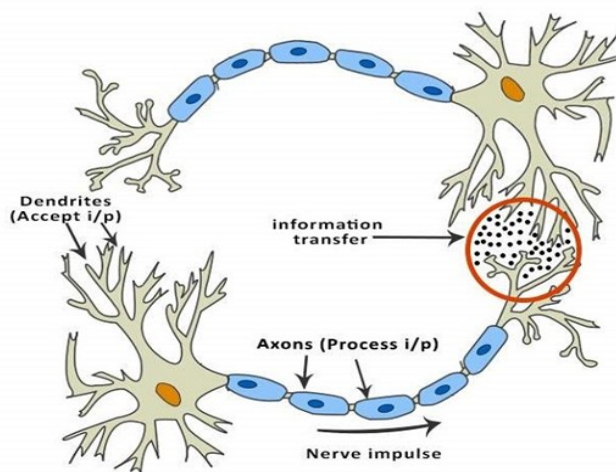
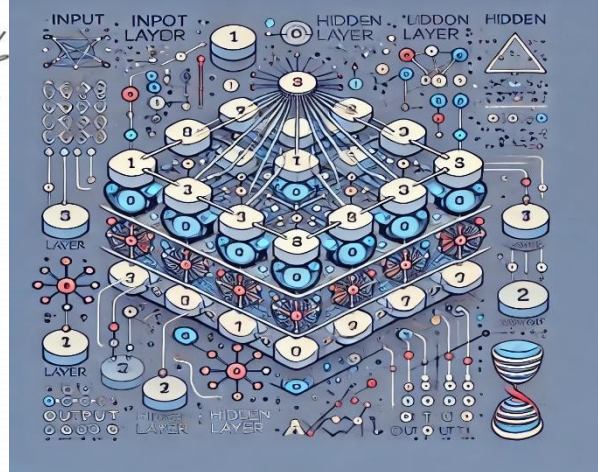


Fig. 1.1. (a) Biological Neural Network



(b) Artificial Neural Network

ANNs are composed of multiple nodes, which imitate biological neurons of the human brain. The neurons are connected by links and they interact with each other. The nodes can take input data and perform simple operations on the data. The result of these operations is passed to other neurons. The output at each node is called its activation or node value.

Each link is associated with weight. Artificial Neural Networks (ANN) are capable of learning, which takes place by altering weight values. As shown in Figure 1.2, a simple Artificial Neural Network (ANN):

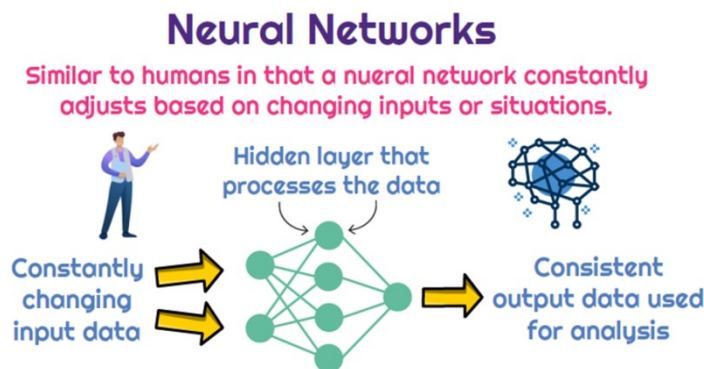


Fig. 1.2 Artificial Neural Network *(modify the figure and add weight to links)*

Similarly, a neural network is made up of a series of nodes that are connected. Each neuron/node takes in a number of inputs and produces an output. The outputs of the neurons are then combined to produce the final output of the neural network. Neural networks can be used to solve a wide variety of problems and are particularly well-suited for problems that involve pattern recognition.

How Neural Network works

Neural networks operate through data-driven learning. During training, a neural network receives a dataset along with corresponding desired outputs. Through iterative adjustments of its internal weights, the network endeavours to generate the expected outcomes for any given input within the dataset.

Once trained, the neural network becomes proficient at making predictions on novel data. For instance, if you train a neural network to distinguish between images of cats and dogs, it can subsequently predict whether a new image features a cat or a dog. As shown in Figure 1.3.

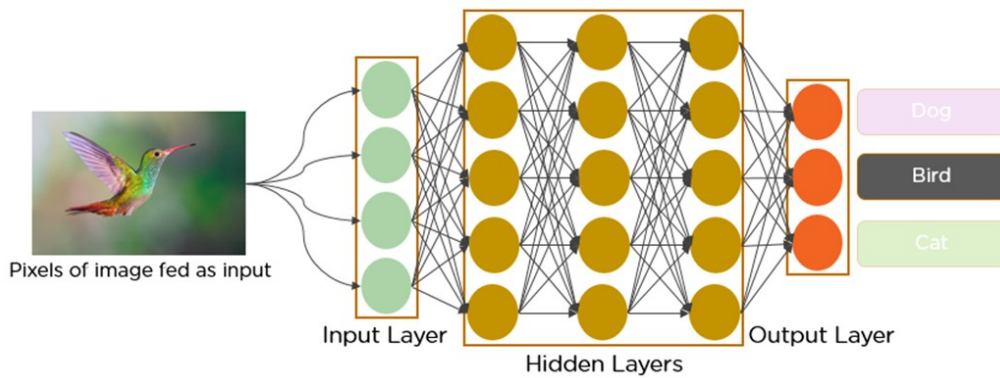


Fig. 1.3 : Working of Neural Network

Consider this simple analogy for understanding neural networks: Think of teaching a child how to recognize various animals. You begin by showing the child pictures of different animals and explaining what each one is. Gradually, the child learns to connect specific features with particular animals; for instance, they may associate four legs and fur with dogs.



Fig. 1.4 : Teaching a Child

With continued learning, the child forms a mental model of animal appearances, enabling them to recognize new animals they encounter.

Similarly, neural networks operate by training on data to associate specific features with desired outputs. Once trained, they can apply this knowledge to identify new data they haven't encountered previously.

Step-by-Step Working of a Neural Network

Step 1. Receiving Input: The input layer takes in numerical data. For example, if an image is fed into the network, it is converted into numbers (pixels).

Step 2. Forward Propagation: Each neuron processes inputs by applying a weight called an importance factor. It uses an activation function such as ReLU, or

Sigmoid to decide whether to pass the information forward. The data moves from the **input layer** → **hidden layers** → **output layer**.

Step 3. Calculating Error (Loss Function): The output is compared to the correct answer. A loss function calculates how far the prediction is from the actual value.

Step 4. Backpropagation and Learning: The network adjusts weights using backpropagation, a method that reduces errors. A technique called Gradient Descent helps improve accuracy.

This process is repeated multiple times, called training, until the network makes correct predictions.

Breakdown of a Neural Network

The fundamental components and mechanisms that make functionality of neural networks:

Neurons:

Artificial neurons, also referred to as nodes or units, lie at the heart of a neural network. Modelled after biological neurons in the brain, these units serve as the basic building blocks of the network. Neurons receive inputs, perform computations, and generate outputs. Neurons are structured into layers within the neural network.

Layers: There are three primary types of layers:

Input Layer: This marks the inception point of the network, receiving the initial data or input.

Hidden Layers: These are layers in between where the input data goes in and the final answer comes out. They do a lot of complicated math to pick out important things from the input.

Output Layer: This is the last layer that gives you the final answer or guess based on all the math done in the hidden layers.

Neural networks can have lots of these hidden layers, and each layer can have different amounts of math-doing units called neurons. When any neural network is designed, it has to decide how big these layers are and how many neurons they have. It depends on what specific problem is trying to solve with the neural network, as shown in Figure 1.5

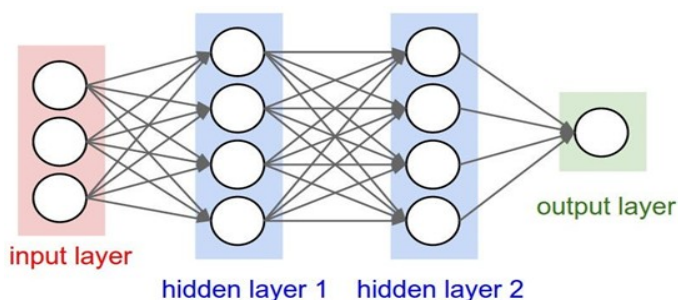


Fig. 1.5 Type of Layers

Connections and weights

Neurons in one layer are connected to neurons in the next layer through connections. Each connection is associated with a weight, which represents the strength or importance of the connection.

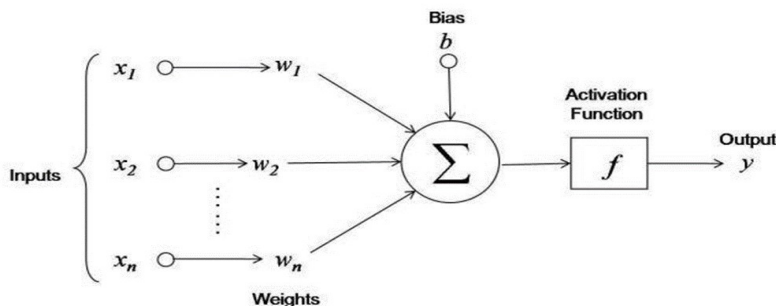


Fig. 1.6: Connections and weights

These weights determine how much influence a particular neuron has on the neurons in the next layer. Initially, these weights are assigned random values, but they get adjusted during the training process to optimize the network's performance.

Bias

As you can see in Figure 1.6, bias is a value that is added to the output of each neuron. The bias helps to prevent the neural network from becoming too sensitive to small changes in the input data.

Activation Function

In the diagram above, the activation function is clearly marked. An activation function is a math rule used to change the output of one neuron so it can be used by the next neuron in the neural network.

These functions are super important because they let the neural network understand complex patterns in the data. What's a nonlinear function? Well, it's a relationship between two things where if one thing changes, the other thing doesn't change by the same amount every time.

There are lots of activation functions to pick from for neural networks. Which one has to be chosen depends on what kind of problem is trying to solve with the network.

Feedforward Propagation

When data enters a neural network, it flows through the layers in a process called feedforward propagation. Each neuron receives inputs from the previous layer, multiplies them by their corresponding weights, sums them up, and

applies the activation function. This process continues layer by layer until the output layer produces the final result.

Backpropagation

Backpropagation is a method used in machine learning to teach neural networks. It starts from the output layer of the network and moves backward, correcting errors along the way. At each step, it calculates how far off the network's guess was and adjusts the weights and biases of the neurons accordingly. This process repeats until the difference between what the network guessed and what it should have guessed is as small as possible. By minimizing these errors, we make the neural network better at learning and improve its ability to tackle tasks, making it a valuable tool in machine learning.

Practical Activity 1.1. Demonstrate the deep learning method by building a hypothetical airplane ticket price estimation service.

Material needed

Computer, paper for writing

Procedure

Here a supervised learning method is used to train the system.

Step1. Airplane ticket price estimator predict the price using the following inputs (excluding return tickets for simplicity)

- Origin Airport
- Destination Airport
- Departure Date
- Airline

Step 2. Like animals, our estimator AI's brain has neurons. They are represented by circles. These neurons are inter-connected. The neurons are grouped into three different types of layers:

1. Input Layer
2. Hidden Layer(s)
3. Output Layer

The input layer receives input data. In our case, we have four neurons in the input layer: Origin Airport, Destination Airport, Departure Date, and Airline. The input layer passes the inputs to the first hidden layer.

Step 3. The hidden layers perform mathematical computations on our inputs. One of the challenges in creating neural networks is deciding the number of hidden layers, as well as the number of neurons for each layer. The “Deep” in

Deep Learning refers to having more than one hidden layer.

Step 4. The output layer returns the output data. In this case, it gives the price prediction.

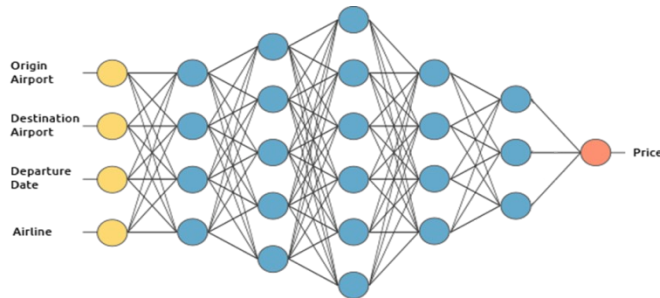


Fig. 1.7

Step 5. Each connection between neurons is associated with a weight. This weight dictates the importance of the input value. The initial weights are set randomly.

Step 6. When predicting the price of an airplane ticket, the departure date is one of the heavier factors. Hence, the departure date neuron connections will have a big weight.

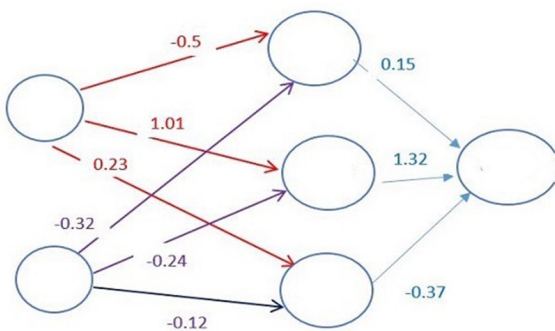


Fig. 1.8

Once a set of input data has passed through all the layers of the neural network, it returns the output data through the output layer.

Session 2 Applications of Neural Network

Artificial Neural Networks (ANNs) are a crucial part of Artificial Intelligence (AI) and Machine Learning (ML). Inspired by the human brain, ANNs consist of interconnected nodes (neurons) that process and analyze complex data. Due to their ability to learn patterns, make decisions, and improve accuracy over time, ANNs are widely used across various industries.

1. Image and Speech Recognition

a) Face Recognition: It is used in security systems, smartphones, and social media platforms like Facebook for automatic tagging. It can be used in applications such as, facial authentication for unlocking devices, surveillance systems, and biometric verification (Figure 2.1).



Fig. 2.1 : Biometric Verification

b) Speech-to-Text Conversion: Virtual assistants like Siri, Google Assistant, and Alexa use ANNs to convert human speech into text. It can be used in applications such as, automated customer service, real-time transcription, and voice-activated systems (Figure 2.2).



Fig. 2.2 : Voice Activated System

c) Handwriting Recognition: Converts handwritten text into digital format using pattern recognition techniques.

Applications: Postal address scanning, bank cheque processing, and digitization of historical manuscripts.

2. Healthcare and Medical Diagnosis

a) Disease Detection: ANNs help in diagnosing diseases like cancer, diabetes, and Alzheimer's by analyzing medical imaging such as, X-rays, MRIs, CT scans. AI-

driven tools assist radiologists in detecting tumors and anomalies with higher accuracy.



Fig. 2.3 : Medical Imaging Using ANN

b) Drug Discovery and Development: It can predict the effectiveness of new drugs by analyzing molecular structures and biological interactions. It speeds up pharmaceutical research and reduces costs.

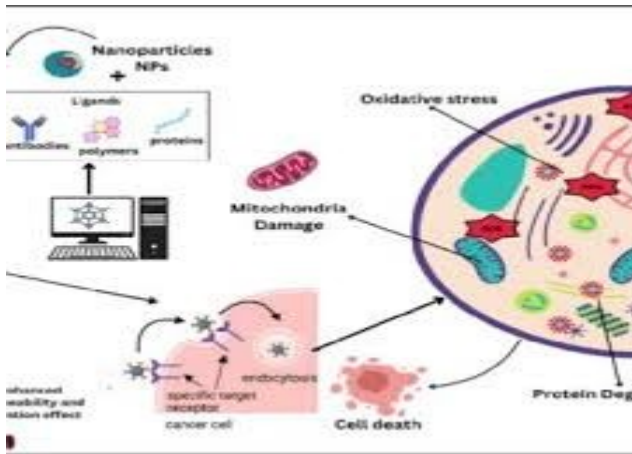


Fig. 2.4 : AI in Pharmacy Research

c) Patient Monitoring: Wearable devices such as smartwatches, fitness bands track health parameters like heart rate, blood pressure, and oxygen levels. Apple Watch detects irregular heartbeats and alerts users about potential heart conditions.

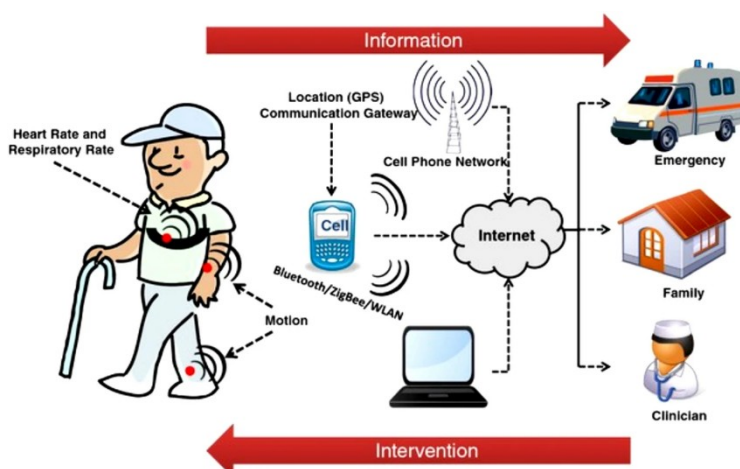


Fig. 2.5 : Patient Monitoring Using ANN

3. Autonomous Vehicles (Self-Driving Cars)

a) **Object Detection and Recognition:** ANNs help detect objects like pedestrians, cyclists, traffic signs, and other vehicles. It is used in Tesla Autopilot and Google's Waymo self-driving cars.

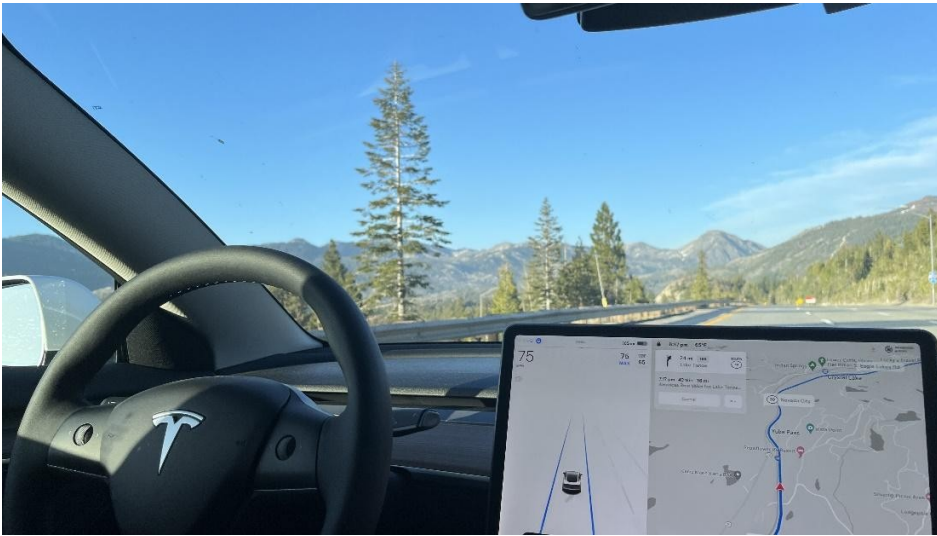


Fig. 2.6: Autopilot

b) **Lane Detection and Path Planning:** Neural networks analyze road images to ensure vehicles stay within lanes. It uses computer vision and deep learning to detect road boundaries.

c) **Decision-Making in Real-Time:** Self-driving cars process vast amounts of data from sensors and cameras to make quick decisions. For example, determining whether to stop at a pedestrian crossing or take an alternate route in case of traffic congestion.

4. Finance and Banking

a) **Fraud Detection:** ANN models analyze transaction patterns to detect suspicious activities. Banks use AI to flag unusual credit card transactions and prevent cyber fraud.

b) **Stock Market Prediction:** Predicts future stock prices based on historical trends and market news sentiment analysis. For example, hedge funds and investment firms use ANN models for algorithmic trading.

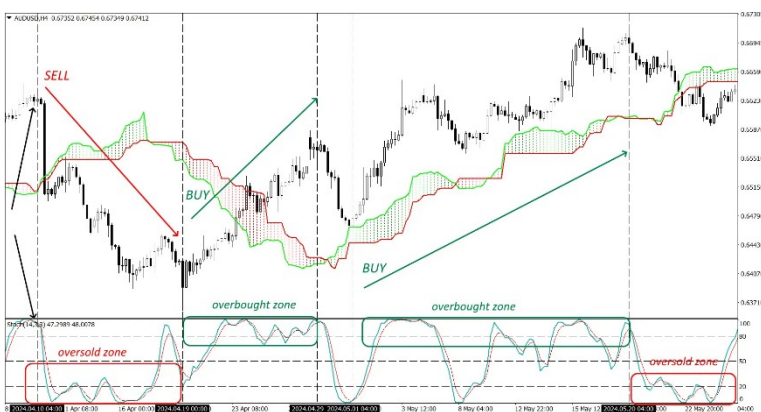


Fig. 2.7: Algorithmic Trading

c) **Chatbots and Customer Support:** AI-driven virtual assistants provide instant responses to customer queries. *Example:* Banking chatbots like HDFC EVA and SBI's AI assistant help customers with account details, transactions, and loan inquiries.



Fig. 2.8: AI Assistant

5. Robotics and Automation

a) **Industrial Automation:** Neural networks help in robotic assembly lines, defect detection, and predictive maintenance. *Example:* AI-powered robots used in automobile manufacturing and semiconductor fabrication.

b) **Humanoid Robots:** Robots powered by ANNs assist in healthcare, education, and customer service. For example, Sophia, developed by Hanson Robotics, interacts with humans using ANN-based NLP models.

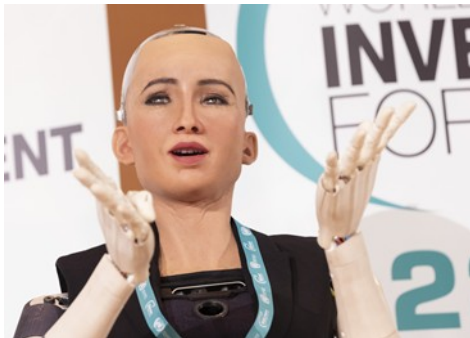


Fig. 2.9 : Sophia Robot

c) **Agriculture and Farming:** AI-powered drones and robots analyze soil health, detect pests, and optimize irrigation. For example, Blue River Technology uses AI to identify weeds and spray herbicides precisely.



Fig. 2.10: Agriculture Robot

6. Natural Language Processing (NLP) and Text Analysis

a) Machine Translation: Neural networks power translation services like Google Translate, DeepL, and Microsoft Translator. For example, Translating entire documents from English to Hindi in real-time.

b) Sentiment Analysis: ANN models analyze customer reviews, social media comments, and news articles to determine sentiment (positive, negative, or neutral). For example, Businesses use sentiment analysis for brand reputation management.

c) Chatbots and Virtual Assistants: AI chatbots understand human language and provide human-like responses. For example, ChatGPT, IBM Watson, and Google Bard process customer inquiries and automate responses.

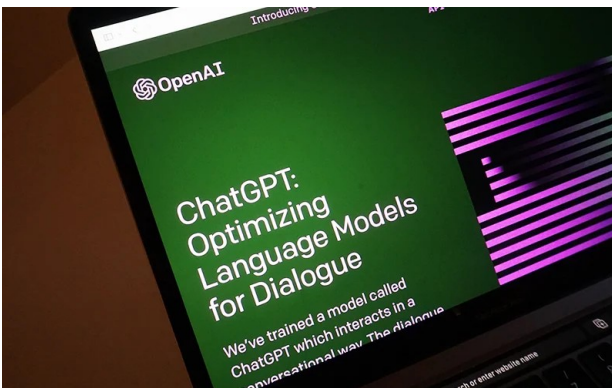


Fig. 2.11 : ChatGPT

7. Gaming and Entertainment

a) AI Opponents in Games: ANN-powered Non-Player Characters (NPCs) make games more interactive and realistic. For example, DeepMind's AlphaGo defeated world champions in the game of Go using ANN models.



Fig. 2.12 : Alphago

b) **Content Recommendation Systems:** Streaming platforms use ANN models to suggest movies, videos, and music based on user preferences. For example, Netflix, YouTube, and Spotify recommend content based on user history.

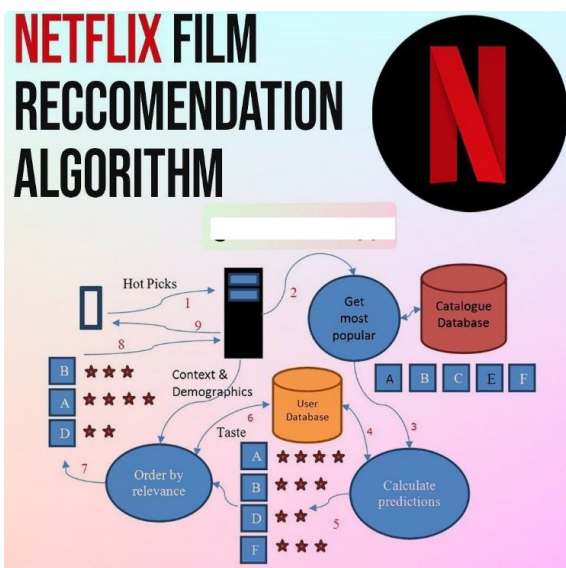


Fig. 2.13 : Netflix Content Recommendation System

c) **Deepfake Technology:** Neural networks generate hyper-realistic AI-generated images and videos. Used in the entertainment industry for special effects and voice synthesis.



Fig. 2.14 : Image Creation using Deepfake Technology

8. Cybersecurity and Network Security

a) **Intrusion Detection Systems:** Detects suspicious activities in networks and prevents cyber-attacks. For example, Firewalls and malware detection software use ANN to identify threats.

b) **Email Spam Filtering:** Identifies spam, phishing emails, and fraudulent activities. For example, Gmail's AI-powered spam filter prevents malicious emails from reaching users.

c) **Secure Authentication Systems:** Biometric security using fingerprint scanning, facial recognition, and voice authentication. For example, iPhone Face ID and fingerprint sensors in banking apps.

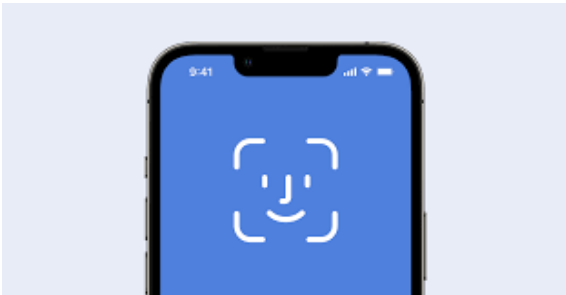


Fig. 2.15 : Face ID on phone

9. Education and E-Learning

a) **Personalized Learning Systems:** AI-powered platforms adjust the difficulty level of courses based on student performance. For example, Khan Academy and Duolingo use AI for adaptive learning.



Fig. 2.16: AI for Education

b) **Automatic Grading and Plagiarism Detection:** AI analyzes written content and provides instant feedback on assignments. *For example*, Turnitin and Grammarly use ANN models for grammar correction and plagiarism detection.

c) **Virtual Teaching Assistants:** AI tutors answer students' queries in real-time. For example, IBM Watson Tutor helps students with homework and complex concepts.



Fig. 2.17: IBM Watson

Artificial Neural Networks (ANNs) have revolutionized multiple industries by enabling machines to learn, adapt, and improve their decision-making abilities. From healthcare and finance to autonomous vehicles and gaming, ANNs are shaping the future of AI-driven innovations. With advancements in deep learning and computational power, the impact of ANNs will continue to grow, making technology more efficient, intelligent, and human-like.

Session 3. Machine Learning Tools

Machine learning (ML) tools are essential for developing, training, and deploying AI models efficiently. These tools provide frameworks, libraries, and platforms that help data scientists and developers create intelligent applications without having to build algorithms from scratch.

Categories of Machine Learning Tools

Machine learning tools can be classified into the following categories:

1. Frameworks and Libraries
2. Integrated Development Environments (IDEs)
3. Data Preparation and Visualization Tools
4. Model Deployment and Monitoring Platforms
5. AutoML Platforms

1. Frameworks and Libraries

These provide essential functions for building and training machine learning models.

- (i) **TensorFlow:** Developed by Google, TensorFlow is an open-source framework for deep learning and ML model development.

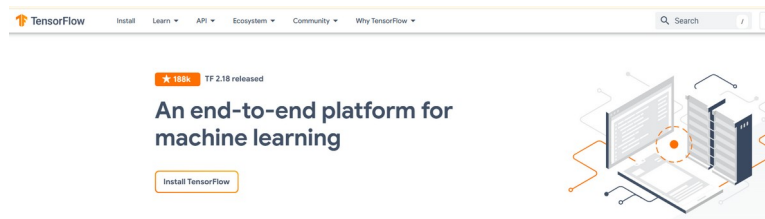


Fig. 3.1 : TensorFlow Web site

- (ii) **PyTorch:** Developed by Facebook, PyTorch is popular for its ease of use and dynamic computation graph, ideal for deep learning research.



Fig. 3.2 : PyTorch Web

- (iii) **Scikit-learn:** A simple and efficient tool for data mining and ML, providing a range of supervised and unsupervised learning algorithms.



Fig. 3.3 : Scikit-Learn web

- (iv) **Keras:** A high-level neural networks API, running on top of TensorFlow for fast experimentation.



- (v) **XGBoost:** An optimized gradient boosting library designed for speed and performance.



Fig. 3.4 : XGBoost

2. Integrated Development Environments (IDEs)

These environments support the development and execution of ML models efficiently.

- (i) **Jupyter Notebook:** An interactive computing environment that supports live code, equations, visualizations, and narrative text.



Fig. 3.5 : Jupyter Notebook

- (ii) **Google Colab:** A cloud-based platform that provides free GPU/TPU support for executing ML models.

Google Colaboratory
Colab is a hosted Jupyter Notebook service that requires no setup to use and provides free access to computing resources, including GPUs and TPUs. Colab is especially well suited to machine learning, data science, and education.

Fig. 3.6 : Google Colab

- (iii) **Spyder:** A Python-based IDE for data science and ML development.



Fig. 3.7 : Spyder

3. Data Preparation and Visualization Tools

Data is a crucial aspect of ML, and these tools help in data preprocessing and visualization.

- (i) **Pandas:** A powerful library for data manipulation and analysis.
- (ii) **NumPy:** A fundamental package for numerical computing in Python.
- (iii) **Matplotlib & Seaborn:** Libraries used for data visualization to understand patterns and distributions.
- (iv) **Tableau:** A visualization software that helps create interactive and shareable dashboards.

4. Model Deployment and Monitoring Platforms

Once an ML model is trained, it needs to be deployed and monitored for performance.

- (i) **TensorFlow Serving:** A flexible, high-performance ML model serving system.
- (ii) **AWS SageMaker:** A fully managed service for building, training, and deploying ML models at scale.
- (iii) **Google AI Platform:** A cloud-based service for training and deploying ML models.

- (iv) **MLflow:** An open-source platform to manage the ML lifecycle, including experimentation, deployment, and monitoring.

5. AutoML Platforms

AutoML tools automate ML processes, making them accessible to non-experts.

- (i) **Google AutoML:** Provides a suite of ML solutions with minimal expertise required.
- (ii) **H2O.ai:** Offers AutoML capabilities to simplify ML model creation.
- (iii) **Microsoft Azure AutoML:** Automates ML model selection, tuning, and deployment.
- (iv) **AutoKeras:** An open-source AutoML library built on top of Keras.

Machine learning tools are critical for accelerating AI development. Whether for data preprocessing, model training, or deployment, choosing the right tool depends on project requirements, scalability needs, and expertise levels. As ML continues to evolve, these tools will play a vital role in simplifying AI development and making it more accessible.

Module 5. AI Project

Session 1. Project Guidelines

1. Objectives of Project Work

- To train students to independently formulate and solve social, philosophical, commercial, or technological problems by using AI and present the results in both written and oral form.
- To expose students to real-life problems in the World of Work.
- To provide students with opportunities to interact with people and understand human relations.

2. About the Project Work

- The project carries --- Credit Points and is graded out of --- marks.
- Every project will have a guide from school.
- A person providing guidance from the industry or business world will serve as the External Guide.
- The Internal Guide is the counsellor responsible for guiding the student.
- Students must report their project progress to the internal guide three times during the course of the project work.
- At the end of the project, students must prepare a document of their work in the form of a Project Report.

3. Development Process

Students are supposed to complete their project work within a period of --- months. Development process contain following points.

1. Project Definition

- (i) Clearly define the problem statement and project objectives.
- (ii) Identify key stakeholders and end-users.
- (iii) Determine the feasibility and impact of the AI solution.
- (iv) Establish success criteria and key performance indicators (KPIs).

2. Data Collection and Preparation

- (i) Identify data sources required for the project.
- (ii) Ensure data quality by handling missing values, inconsistencies, and noise.
- (iii) Split data into training, validation, and test sets.
- (iv) Comply with ethical and legal standards, including data privacy regulations (GDPR, CCPA, etc.).

3. Model Selection and Development

- (i) Choose the appropriate machine learning or deep learning model.

- (ii) Use frameworks like TensorFlow, PyTorch, or Scikit-learn based on project needs.
- (iii) Optimize model performance through hyperparameter tuning.
- (iv) Implement explainability techniques to interpret model decisions.

4. Training and Evaluation

- (i) Train the model on a well-prepared dataset.
- (ii) Evaluate model performance using suitable metrics (accuracy, precision, recall, F1-score, etc.).
- (iii) Perform cross-validation to prevent overfitting.
- (iv) Compare model results with baseline methods.

5. Deployment Strategy

- (i) Select an appropriate deployment platform (cloud, edge devices, on-premises).
- (ii) Containerize the model using Docker for portability.
- (iii) Use APIs for integration with existing applications.
- (iv) Implement monitoring tools to track model performance post-deployment.

6. Performance Monitoring and Maintenance

- (i) Continuously monitor model accuracy and update as needed.
- (ii) Set up automated alerts for model drift detection.
- (iii) Regularly retrain the model with new data.
- (iv) Maintain logs and documentation for reproducibility.

7. Ethical Considerations and Compliance

- (i) Ensure fairness and eliminate bias in the AI model.
- (ii) Adhere to AI governance policies and ethical guidelines.
- (iii) Maintain transparency by documenting decision-making processes.
- (iv) Ensure AI-driven decisions align with societal and business ethics.

8. Project Documentation and Reporting

- (i) Maintain thorough documentation, including methodology, code, and results.
- (ii) Provide periodic reports to stakeholders on project progress.
- (iii) Create user guides and technical documentation for end-users.
- (iv) Store source code and datasets securely for future reference.

9. Collaboration and Team Coordination

- (i) Assign clear roles and responsibilities to team members.
- (ii) Use project management tools like Jira, Trello, or Asana.
- (iii) Maintain version control with Git and conduct regular code reviews.
- (iv) Foster cross-functional collaboration between data scientists, engineers, and business teams.

10. Scalability and Future Improvements

- (i) Design models with scalability in mind for future enhancements.
- (ii) Plan for system upgrades and performance optimization.
- (iii) Explore automation possibilities for reducing manual efforts.
- (iv) Keep track of emerging AI trends to incorporate innovations into the project.

By following these guidelines, AI projects can be executed efficiently, ensuring robustness, fairness, and long-term sustainability.

Session 2. Project Formats

AI Project Synopsis

Use the following format for preparation of synopsis at the beginning of project work.

1. Title of the Project

(Provide a concise and clear title for the project.)

2. Introduction

(A brief introduction to the project, its purpose, and its significance.)

3. Objectives/ Existing System and Need for System

(List the main objectives of the project, specifying what it aims to achieve.)

4. Scope of the Project

(Define the boundaries of the project, including functionalities and limitations.)

5. Technologies Used

(Outline the programming languages, frameworks, databases, and tools used.)

6. System Architecture

(A high-level description of the system design, including a block diagram if applicable.)

7. Modules and Functionalities

(Divide the project into key modules and briefly describe their roles and functionalities.)

■ Module 1: Description

■ Module 2: Description

■ Module 3: Description

8. Methodology

(Describe the software development model used, e.g., Agile, Waterfall, etc.)

9. Expected Outcome

(Explain the expected results and benefits of the project.)

10. Conclusion

(A summary of the project's importance and impact.)

11. References (if any)

(List of books, websites, or papers referred to in the project.)

AI Final Project Writing Format

1. Title Page

- Project Title
- Author(s) / Team Members
- Institution / Organization
- Date of Submission

2. Abstract

- A brief summary of the project, including objectives, methodology, and key findings.

3. Introduction

- Background information on the problem domain.
- Problem statement and significance of the project.
- Objectives and expected outcomes.

4. Literature Review

- Overview of existing research and technologies related to the project.
- Comparison of different approaches and their limitations.
- Justification for the chosen methodology.

5. Methodology

- Data collection sources and preprocessing techniques.
- Machine learning or deep learning models used.
- Algorithms, tools, and frameworks utilized.
- Training and evaluation process.

6. Implementation

- Step-by-step description of model development.
- Code snippets (if necessary) and explanations.
- System architecture and workflow diagrams.

7. Results and Analysis

- Model performance metrics (accuracy, precision, recall, etc.).
- Visualizations of data and results (graphs, tables, etc.).
- Comparison with baseline models or previous research.

8. Discussion

- Interpretation of results and insights gained.
- Challenges faced during the project.
- Potential improvements and future scope.

9. Conclusion

- Summary of key findings and achievements.
- Final remarks on the impact and usability of the AI solution.

10. References

- Citation of books, research papers, and online sources used.

11. Appendices (if applicable)

- Additional data, code documentation, or supplementary materials.

This structured format ensures clarity, consistency, and professionalism in AI project documentation.

Session 3. Project Review

Use following Project Review Chart for Evaluation of Project

Project Stage	Evaluation Criteria	Rating (1-5)	Comments
Project Definition	Clarity of problem statement and objectives		
	Feasibility and relevance of the project		
Research and Background	Adequacy of literature review		
	Comparison with existing solutions		
Data Collection	Data sources identified and validated		
	Data preprocessing and cleaning		
Methodology	Appropriateness of ML/AI models used		
	Justification for chosen algorithms		
Implementation	Accuracy of model execution and training		
	Efficiency and optimization of the solution		
Evaluation & Testing	Performance metrics assessment		
	Model validation and cross-validation		
Deployment	Deployment strategy (cloud, edge, local)		
	Scalability and usability of the solution		
Results & Analysis	Presentation and visualization of results		
	Interpretation and discussion of findings		

Ethical Considerations	Bias mitigation strategies		
	Compliance with regulations		
Documentation	Clarity and completeness of project documentation		
	References and citations included		
Overall Evaluation	Project innovation and uniqueness		

Final Score: _____/-----

This review chart provides a structured approach to assessing each phase of an AI project. Ratings (1-5) help gauge the project's quality and effectiveness.

Session 4. Sample Project

AI Final Project

1. Title Page

Project Title: Sentiment Analysis on Customer Reviews

Author(s) / Team Members: X, Y

Institution / Organization: XYZ University

Date of Submission: March 7, 2025

2. Abstract

This project aims to develop a sentiment analysis model to classify customer reviews as positive, negative, or neutral. Using Natural Language Processing (NLP) techniques and machine learning, we preprocess and analyze text data to extract meaningful insights. The project employs a logistic regression model and deep learning techniques for sentiment classification, achieving an accuracy of 85%. The results can help businesses understand customer opinions and improve their services.

3. Introduction

Background Information

Customer reviews are a valuable source of feedback for businesses. Analyzing these reviews manually is time-consuming and inefficient. Sentiment analysis, an NLP technique, automates this process by categorizing opinions into sentiment classes.

Problem Statement

Existing sentiment analysis systems often struggle with contextual understanding and sarcasm, leading to misclassification. Our project aims to improve sentiment classification accuracy using machine learning.

Objectives

- Develop a machine learning model for sentiment classification.
- Improve accuracy through preprocessing and model optimization.
- Provide a user-friendly interface for sentiment analysis.

4. Literature Review

Existing Research

Several approaches exist for sentiment analysis, including lexicon-based methods and machine learning models like Naive Bayes, Support Vector Machines, and deep learning models.

Comparison of Approaches

- **Lexicon-based methods:** Depend on predefined word lists but lack context understanding.
- **Machine learning models:** Require labeled data but generalize well.
- **Deep learning models:** Capture complex patterns but need large datasets.

Justification

We chose logistic regression for baseline performance and a deep learning model for improved accuracy due to its ability to understand complex linguistic patterns.

5. Methodology

Data Collection

- Dataset: IMDB and Amazon customer reviews.
- Preprocessing: Tokenization, stopwords removal, stemming, and vectorization using TF-IDF.

Machine Learning Models

- Logistic Regression for baseline.
- LSTM (Long Short-Term Memory) neural network for deep learning.

Tools and Frameworks

- Python, Scikit-learn, TensorFlow, NLTK, Pandas, Matplotlib.

Training and Evaluation

- Split dataset into training (80%) and testing (20%).
- Evaluation metrics: Accuracy, Precision, Recall, F1-score.

6. ImplementationSteps

1. *Data preprocessing*: Cleaning and transforming text data.
2. *Feature extraction*: Converting text into numerical representation.
3. *Model training*: Training both logistic regression and LSTM models.
4. *Model evaluation*: Measuring performance using metrics.

System Architecture

- *Input*: Customer reviews.
- *Processing*: NLP preprocessing and model prediction.
- *Output*: Sentiment classification (Positive/Negative/Neutral).

Code Implementation

```
import pandas as pd
import numpy as np
import re
import nltk
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, SpatialDropout1D
```

```

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Load dataset
df = pd.read_csv('customer_reviews.csv')

# Preprocessing
def clean_text(text):
    text = re.sub(r'^a-zA-Z', ' ', text)
    text = text.lower()
    text = text.split()
    text = [word for word in text if word not in
stopwords.words('english')]
    return ' '.join(text)
df['cleaned_reviews'] = df['review'].apply(clean_text)

# TF-IDF Vectorization
vectorizer = TfidfVectorizer(max_features=5000)
X = vectorizer.fit_transform(df['cleaned_reviews']).toarray()
y = df['sentiment']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Logistic Regression Model
lr_model = LogisticRegression()
lr_model.fit(X_train, y_train)

# Evaluate Logistic Regression
y_pred = lr_model.predict(X_test)
print("Logistic Regression Accuracy:", accuracy_score(y_test,
y_pred))
print(classification_report(y_test, y_pred))

# Deep Learning Model with LSTM
max_words = 5000
max_len = 200

```

```

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(df['cleaned_reviews'])
X_seq = tokenizer.texts_to_sequences(df['cleaned_reviews'])
X_pad = pad_sequences(X_seq, maxlen=max_len)

X_train_dl, X_test_dl, y_train_dl, y_test_dl =
train_test_split(X_pad, y, test_size=0.2, random_state=42)

model = Sequential()
model.add(Embedding(max_words, 128, input_length=max_len))
model.add(SpatialDropout1D(0.2))
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(3, activation='softmax'))
model.compile(loss='sparse_categorical_crossentropy',
optimizer='adam', metrics=['accuracy'])

model.fit(X_train_dl, y_train_dl, epochs=5, batch_size=64,
validation_data=(X_test_dl, y_test_dl))

```

7. Results and Analysis

Performance Metrics

Model	Accuracy	Precision	Recall	F1-score
Logistic Regression	78%	76%	74%	75%
LSTM	85%	83%	82%	83%

Visualizations

- Confusion matrices for both models.
- Accuracy and loss curves for LSTM model.

8. Discussion

Interpretation of Results

- The LSTM model performed better than logistic regression.
- Challenges included handling sarcasm and ambiguous sentiments.

Potential Improvements

- Incorporating transformer-based models like BERT for better contextual understanding.
- Expanding the dataset to include multiple domains.

9. Conclusion

This project successfully implemented sentiment analysis using machine learning and deep learning. The LSTM model achieved an 85% accuracy, demonstrating its potential for real-world applications in customer feedback analysis.

10. References

- Jurafsky, D., & Martin, J. H. (2021). Speech and Language Processing.
- Pang, B., & Lee, L. (2008). Opinion Mining and Sentiment Analysis.
- Research papers on sentiment analysis techniques.

11. Appendices

- Sample code for data preprocessing and model training.
- Additional graphs and visualizations.
- Hyperparameter tuning details for deep learning models.