PYTHON PROGRAMMING (20CS31P) – STUDY MATERIAL

WEEK-1

Fundamental Concepts:

Brief history; features; applications of python; python distributions; versions; python IDEs; Python interpreter; Execution of python programs, debugging python code; Indentation, Comments; best practices for python programming; Character set; tokens; keywords, variables, naming rules for variables, Assignment.

Brief History:

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in December 1989 by Guido Van Rossum at CWI in Netherland.
- In February 1991, Guido Van Rossum published the code (labeled version 0.9.0) to alt.sources.
- In 1994, Python 1.0 was released with new features like lambda, map, filter, and reduce.
- Python 2.0 added new features such as list comprehensions, garbage collection systems.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify the fundamental flaw of the language.

What is Python?

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.

Features of Python Programming Language:

- Python provides many useful features which make it popular and valuable from the other programming languages.
- It supports object-oriented programming; procedural programming approaches and provides dynamic memory allocation.
- Few essential features of python are given below
  - ✓ Easy to Learn and Use
  - ✓ Expressive Language
  - ✓ Interpreted Language
  - ✓ Cross-platform Language
  - ✓ Free and Open Source
  - ✓ Object-Oriented Language
  - ✓ Extensible

- ✓ Large Standard Library
- ✓ GUI Programming Support
- ✓ Integrated
- ✓ Embeddable
- ✓ Dynamic Memory Allocation

Python Applications:

- Python is known for its general-purpose nature that makes it applicable in almost every domain of software development.
- Python makes its presence in every emerging field. It is the fastest growing programming language and can develop any application.
- Some of the applications of python are given below
  - ✓ Web Applications
  - ✓ Desktop GUI Applications
  - ✓ Console-based Application
  - ✓ Software Development
  - ✓ Scientific and Numeric
  - ✓ Business Applications
  - ✓ Audio or Video-based Applications
  - ✓ 3D CAD Applications
  - ✓ Enterprise Applications
  - ✓ Image Processing Application

Python Distributions:

Here is a brief tour of Python distributions, from the standard implementation (CPython) to versions optimized for speed (PyPy), for special use cases (Anaconda, ActivePython), for different language runtimes (Jython, IronPython), and even for cutting-edge experimentation (PyCopy, MesaPy).

Python Versions:

Python programming language is being updated regularly with new features and supports. There are lots of update in Python versions, started from 1994 to current release.

A list of Python versions with its released date is given below.

Python Version	Released Date
Python 1.0	January 1994
Python 2.0	October 16, 2000
Python 3.0	December 3, 2008
Python 3.6	December 23, 2016
Python 3.7	June 27, 2018
Python 3.8	October 14, 2019
Python 3.9	Oct. 5, 2020
Python 3.10.0	Oct. 4, 2021

Python IDEs:

- An IDE (Integrated Development Environment) understand the code much better than a text editor.
- It usually provides features such as build automation, code lining, testing and debugging.
- This can significantly speed up the work.
- There are different IDEs which are as follows.
  1. PyCharm
  2. Spyder
  3. Eclipse PyDev
  4. IDLE
  5. Sublime Text 3
  6. Atom
  7. Thonny
  8. Visual Studio Code
  9. Vim

Python Interpreter:

- The word "interpreter" can be used in a variety of different ways when discussing Python.
- Sometimes interpreter refers to the Python REPL, the interactive prompt you get by typing python at the command line.

- Sometimes people use "the Python interpreter" more or less interchangeably with "Python" to talk about executing Python code from start to finish.
- Before the interpreter takes over, Python performs three other steps: lexing, parsing, and compiling.
- Together, these steps transform the programmer's source code from lines of text into structured code objects containing instructions that the interpreter can understand.
- The interpreter's job is to take these code objects and follow the instructions.
- We may be surprised to hear that compiling is a step in executing Python code at all.
- Python is often called an "interpreted" language like Ruby or Perl, as opposed to a "compiled" language like C or Rust.
- However, this terminology isn't as precise as it may seem. Most interpreted languages, including Python, do involve a compilation step.

Python Statement, Indentation and Comments:

Python Statement

- Instructions that a Python interpreter can execute are called statements.
- For example, a = 1 is an assignment statement. if statement, for statement, while statement, etc. are other kinds of statements which will be discussed later.

Multi-line statement

- In Python, the end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\).

Ex:

```
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8 + 9
```

This is an explicit line continuation. In Python, line continuation is implied inside parentheses (), brackets [ ], and braces { }. For instance, we can implement the above multi-line statement as

```
a = (1 + 2 + 3 +
     4 + 5 + 6 +
     7 + 8 + 9)
```

We can also put multiple statements in a single line using semicolons, as follows:

```
a = 1; b = 2; c = 3
```

Python Indentation:

- Most of the programming languages like C, C++, and Java use braces { } to define a block of code. Python, however, uses indentation.
- Indentation refers to the spaces at the beginning of a code line.
- A code block (body of a function, loop, etc.) starts with indentation and ends with the first unindented line.
- All the statements with the block will have the same indentation.
- Generally, four whitespaces are used for indentation and are preferred over tabs.
- Incorrect indentation will result in **IndentationError**.

Ex:

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

- The enforcement of indentation in Python makes the code look neat and clean. This results in Python programs that look similar and consistent.
- Indentation can be ignored in line continuation, but it's always a good idea to indent. It makes the code more readable.

Ex:

```
if True:
    print('Hello')
    a = 5
```

Python Comments:

- Comments are non executable statements.
- Comments can be used to explain the python code.
- Comments can be used to make the python code more readable.
- In Python, comments start with # and Python Interpreter will ignore them.
- Comments can be placed at the end of the line and python will ignore the rest of the line.

Ex:

```
#This is a comment
#print out Hello
print('Hello')
```

Multiline comments:

To add a multi line comment insert # for each line.

Ex:

```
#This is a long comment
#and it extends
#to multiple lines
```

Best Practices of Python Programming:

Python best practices that can be really useful for Python programmers to improve their coding experience.

1. Writing Well-Structured Code
2. Having Proper Comments and Documentation
3. Proper Naming of Variables, Classes, Functions and Modules
4. Writing Modular Code
5. Using Virtual Environments

Python Character Set:

- Character set is the set of valid characters that a language can recognize.
- A character represents any letter, digit or any other symbol.
- Python has the following character sets:
  - ✓ Letters – A to Z, a to z
  - ✓ Digits – 0 to 9
  - ✓ Special Symbols - + - * / @ # $ % & ( ) { } [ ] : ; ‘ ‘’ etc.
  - ✓ Whitespaces – Blank Space, tab, carriage return, newline, form feed

Tokens:

- The smallest individual unit in a program is called token.
- All statements are built using tokens.
- Python has the following tokens:
  1) Keyword

2) Identifiers
3) Literals
4) Operators
5) Punctuators

Keywords:

- Keywords are reserved words that have special meaning in a programming language.
- These are reserved for special purpose and must not be used as normal identifier names.
- There are 33 keywords in Python 3.7. This number can vary slightly over the course of time.
- Some of the keywords are False, True, if, elif, else, while, for, break, continue, and, in, not, print, import, etc.

Identifiers:

- Identifiers are the names given to different program elements like variables, objects, classes, function, list, dictionaries etc

Rules for Naming Identifier:

- ✓ Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore. Names like **myClass**, **var_1** and **print_this_to_screen**, all are valid example.
- ✓ An identifier cannot start with a digit. **1variable** is invalid, but **variable1** is a valid name.
- ✓ Keywords cannot be used as identifiers.
- ✓ Any other special character or white spaces are strictly prohibited in identifiers.
- ✓ Python is a case sensitive programming language. So koppal, KOPPAL and Koppal are 3 different identifiers.
- ✓ Identifiers can be of any length.

Variables:

- The quantity which changes during the execution of the program is known as variable.
- In Python, there is no need to declare a variable explicitly as in C, C++, Java by using int, float, etc.
- To define a variable in Python, simple assign a value to a name as given below.

Ex:

A= 20

B = “hello”

C= 35.5

Rules for Naming Variables:

- ✓ Variable name consists of any number of letters, underscores and digits.
- ✓ Variable name should not start with a digit.

- ✓ Keywords cannot be used as variable names.
- ✓ Any other special character or white spaces are strictly prohibited in identifiers.
- ✓ Variable names are case sensitive. So total, TOTAL and Total are 3 different variables in Python.

Assignments:

- ➢ The assignment operator = is used to assign values to variables.
- ➢ The general format for assigning values to variables is as follows.

variable_name = value

Ex:

```
a = 20
print(a)
name = “Krishna”
print(name)
country =”India”
print(country)
```

WEEK-2

Basics I/O operations Input- input (), raw_input() ; output – print (), formatting output. Datatypes Scalar type: Numeric (int, long, float, complex), Boolean, bytes, None; Type casting Operators Arithmetic, Comparison/Relational, Logical/Boolean, Bitwise; string operators; Expressions and operator precedence.

Basics I/O operations

Input:

Python provides us with two inbuilt functions to read the input from the keyboard.

- **input(prompt)**
- **raw_input(prompt)**

input( ) : In python, input() function is used to read or collect data from the user.

The syntax of input() function is

variable_name = input([prompt])

The string that is written inside the parenthesis will be printed on the screen.

Ex:

```
name = input("what is your name?")
print(name)
age = input("what is your age?")
print(age)
print(type(age))
age = int(input("what is your age?"))
print(age)
print(type(age))
```

raw_ input(): This function works in older versions of python and is similar to input(). (like Python 2.x).

Ex:

```
name = raw_input("Enter your name : ")
print(name)
```

Output: print()

In python, print() function is used to display message or text to the screen.

The syntax of print() function is

1. **print("message")** => **to display string message**
2. **print(variable)** => **to display values**

- This string literal **\n** is used to add a new blank line while printing a statement.
- An empty quote (**""**) is used to print an empty line.
- Arguments are separated from each other using a **","** separator.

Ex:

```
print("Hello World")
print("Hello \n World")
print("")
a=10
print(a)
b=20
print(b)
print(a,b)
print("Hello",a,b)
c="koppal"
print("gpt",c)
s1="hello"
s2="koppal"
print(s1,s2)
```

Formatted Strings (f-strings):

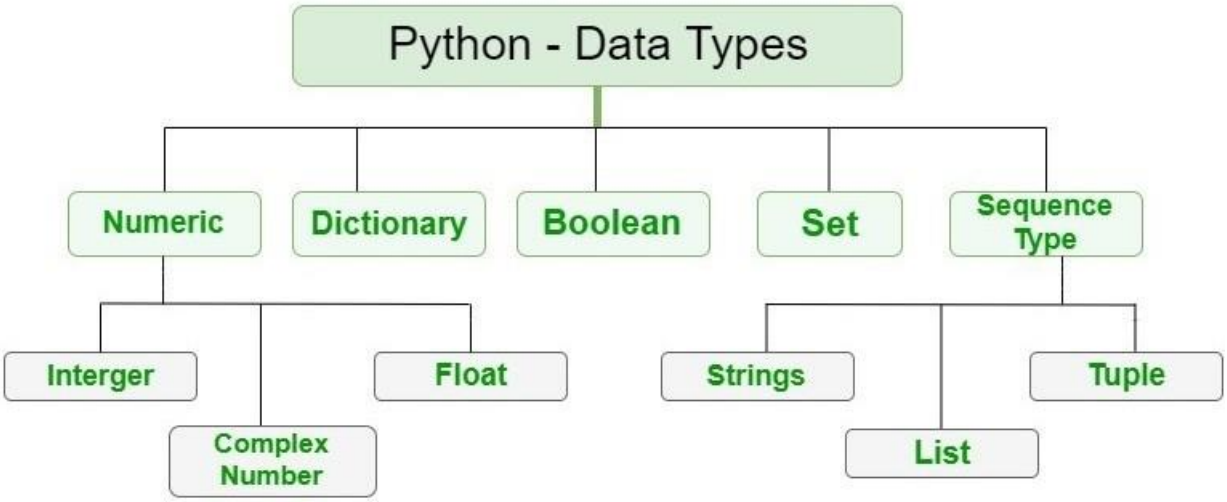
- F-strings were introduced in python 2.6
- A f-string is a string literal prefixed with "f"
- We need to specify the name of the variable or the expression inside the curly braces { } to display its value.

Ex:

```
country = "India"
print(f "I Live in { country }")
r = 3.142
print(f "Area= { 3.142*r*r }")
```

Data Types:

- Data types tell the type of data that the variable is holding.
- Data types are the classification or categorization of input values.
- Python variables can store data of different types.
- Python has the following built-in data types.
- Following are the standard or built-in data type of Python:
  - ✓ **Numeric**
  - ✓ **Sequence Type**
  - ✓ **Boolean**
  - ✓ **Set**
  - ✓ **Dictionary**



Numeric:

In Python, numeric data type represents the data which has numeric value. Numeric value can be integer, floating number or complex numbers.

- **Integers:** This value is represented by **int** class. It contains positive or negative **whole** numbers (without fraction or decimal). In Python there is no limit to how long an integer value can be.

  Ex:

  x= 1

  y=23534958345

  z=-2356

  print(x)

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

- **Float:** This value is represented by **float** class. It is a real number with floating point representation. It is specified by a decimal point.

  Ex:

```
x= 1.14560
y=1.1
z=-35.689
print(x)
print(y)
print(z)
print(type(x))
print(type(y))
print(type(z))
```

- **Complex Numbers**: Complex number is represented by **complex** class. It is specified as **(real part) + (imaginary part) j.**

```
x= 3+5j
y=6j
z=-7j
print(x)
print(y)
print(z)
print(type(x))
print(type(y))
print(type(z))
```

Boolean:

- Boolean represent one of the 2 values: True or False
- When we evaluate any expression in python, we get one of 2 values: True or False
- When we compare 2 values, the expression is evaluated and python returns the Boolean value.

Ex:

```
print(10>9)
print(10==9)
print(10<9)
```

- When we run a condition in an if statement, python returns True or False.

Ex:

```
a=10
b=20
if a>b:
    print("a is bigger")
else:
    print("a is bigger")
```

None:

- None keyword is used to define a null value.
- It is not the same as 0, False or an empty string.

Ex:

```
x=None
print(x)
```

Type Casting:

The process of converting one data type to another is known as type casting.

1. Integer conversion function: int()

Ex:

```
a= int(3.5)
print(a)
print(type(a))
b= int("5")
print(b)
print(type(b))
```

2. String conversion function: str()

Ex:

```
a= str(8)
print(a)
```

```
print(type(a))
b= str(3.5)
print(b)
print(type(b))
```

3. Float conversion function: float()

```
a= float(8)
print(a)
print(type(a))
b= float("3")
print(b)
print(type(b))
c= float(35.50)
print(c)
print(type(c))
```

4. complex() casting function

Ex:

```
a= complex(8)
print(a)
print(type(a))
b= complex(3,5)
print(b)
print(type(b))
```

Operators:

Python language supports a wide range of operators.

They are

1. Arithmetic Operators
2. Comparison Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators

Arithmetic Operators:

Arithmetic operators are used to execute arithmetic operations such as addition, subtraction, division, multiplication etc.

Operator	Description	Example
+	Addition	5+2 gives 7
-	Subtraction	5-2 gives 2
*	Multiplication	5*2 gives 10
/	Division	5/2 gives 2.5
%	Modulus: Gives the remainder	5%2 gives 1
//	Floor Division: Gives the integral part of the quotient	5//2 gives 2
**	Exponent	5**2 gives 25

```
print("Addition Operator")
print(10 + 35)
print("Subtraction  Operator")
print(-10 + 35)
print("Multiplication Operator")
print(4*2)
print("Exponent  Operator")
print(4**2)
print("Division  Operator")
print(45/10)
print("Modulus Operator --> Returns a reminder")
print(2025%10)
print("Floor division Operator -> Returns the integral part of the quotient")
print(45//10.0)
print(2025//10)
```

Comparison Operators:

- When the values of two operands are to be compared then comparison operators are used.
- The output of these comparison operators is always a Boolean value, either **True** or **False**.

Operator	Description	Example
>	Greater Than	5 > 7 gives False
<	Less Than	5 < 7 gives True
= =	Equal To	5 = =5 gives True
!=	Not Equal To	5! =5 gives False
<=	Less Than or Equal To	5 >= 7 gives False
>=	Greater Than or Equal To	5 <= 7 gives True

```
print("Equal Operator")
print(10 == 12)
print("Not Equal To Operator")
print(10 != 12)
print("Less Than Operator")
print(10 < 12)
print("Greater Than Operator")
print(10 > 12)
print("Lesser than or Equal to")
print(10 <= 12)
print("Greater than or equal to")
print(10 >= 12)
print("P" < "Q")
print("Ravi" > "Ramu")
print(True == True)
```

Logical Operators:

- The result of the logical operator is always a Boolean value, **True** or **False**.
- The logical operators are **and, or** and **not**.

Operator	Description	Example
and	Returns True if both statements are True	5 > 2 and 5 < 7 gives True
or	Returns True if at least one of the statement is True	5 > 2 or 5 < 3 gives True
not	Reverses the result.	not(5 > 2 or 5 < 3) gives False

```
print(True and False)
print(True or False)
print(not(True) and False)
print(not(True and False))
print((10 < 0) and (10 > 2))
print((10 < 0) or (10 > 2))
print(not(10 < 0) or (10 > 2))
print(not(10 < 0 or 10 > 2))
```

Bitwise operators:

- In Python, bitwise operators are used to performing bitwise calculations on integers.
- The integers are first converted into binary and then operations are performed on bit by bit, hence the name bitwise operators.

Operator	Name	Description
&	AND	Sets output bit to 1 if both input bits are 1
|	OR	Sets output bit to 1 if at least one of the two input bits is 1
^	XOR	Sets output bit to 1 if exactly one of the two input bits is 1
~	NOT	Inverts all bits
<<	Left Shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Right Shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

```
a=10 # binary 1010
b=7 # binary 0111
```

print(a&b)

print(a|b)

print(a^b)

print(~a)

print(a<<1)

print(a>>1)

Assignment Operators:

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

String Operators:

Operator	Description	Example
+	Concatenates the two strings	"ab"+"cd" gives "abcd"
*	Creates multiple copies of a string	"ab"*3 gives "ababab"
[]	Return a character from an index	"abcde"[2] gives "c"
[:]	Returns a substring between two indices	"abcde"[1:3] gives "bc"
in	Returns True if a given substring exists in the string	"ab" in "abcd" gives True
not in	Returns False if a given substring exists in the string	"ef" not in "abcd" gives True

Operator Precedence and Associativity

Operator precedence determines the way in which operators are parsed with respect to each other. Almost all the operators have left-to-right associativity.

Precedence	Operator	Name
1	()	Paranthesis
2	**	Exponentiation
3	+a, -a, ~a	Unary plus, minus, complement
4	/ * // %	Divide, Multiply, Floor Division, Modulo
5	+ -	Addition, Subtraction
6	>> <<	Shift Operators
7	&	Bitwise AND
8	^	Bitwise XOR
9	|	Bitwise OR
10	>= <= > <	Comparison Operators
11	!= ==	Equality Operators
12	+= -= /= *= =	Assignment Operators
13	is, is not, in, not in	Identity and Membership Operators
14	and, or, not	Logical Operators

WEEK-3

Control Flow: Conditional blocks If statement: general format; Multiway branching; Sufficient examples;

Control Flow: Conditional blocks

In Python, conditional block statements execute depending on whether a given condition is True or False. We can execute different blocks of codes depending on the outcome of a condition. Condition statements always evaluate to either True or False.

Python programming language assumes any non-zero and non-null values as TRUE, and any zero or null values as FALSE value.

Python programming language provides the following types of conditional-making statements. 1) **if statements:** An if statement consists of a Boolean expression followed by one or more statements.

2) **if...else statements:** An if statement can be followed by an optional else statement, which executes when the Boolean expression is FALSE.

3) **if…elif…else Statement:** The elif statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

4) **nested if statements:** One if or else if statement inside another if or else if statement(s).

if statement:

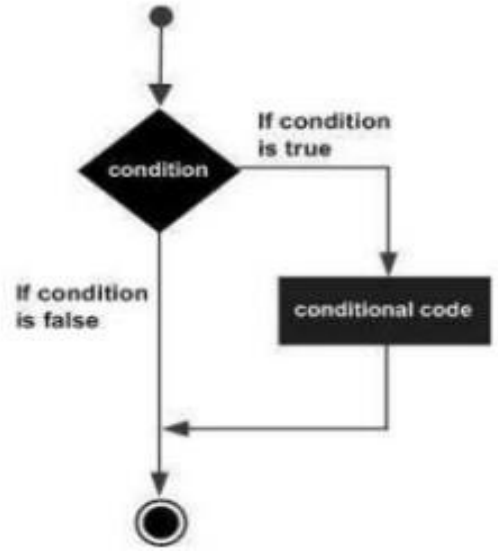
- It is a one way branching statement.
- The if statement checks the condition and executes the if block of code when the condition is True, and if the condition is False, it will not execute it.
- The syntax of if statement is given below

Syntax:

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

- If the condition is True, then statement(s) will be executed If the condition is False, statement(s) will not be executed.
- if is a keyword.
- Expression: valid Boolean expression results in either True or False
- Colon should be present at the end of if statement.
- If block statement should have proper indentation.

Flow Diagram



Ex1:

```
a = 10
b = 20
if b > a:
  print("b is greater than a")
```

Ex2:

```
a=int(input("enter a number"))
if a>0:
   print("positive number")
```

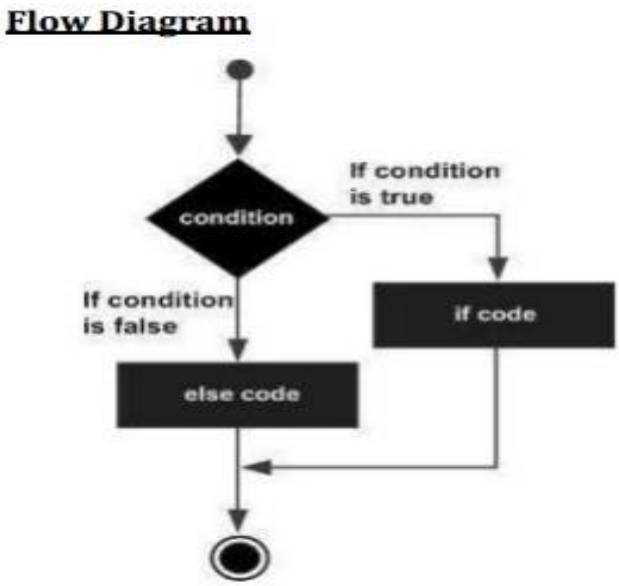
if...else statement:

- It is a 2 way branching statement.
- The if-else statement checks the condition and executes the if block of code when the condition is True, and executes the else block of code when the condition is False.
- The syntax of if….else statement is given below

Syntax:

```
if(expression):
   statement(s)-1
else:
   statement(s)-2
```

- If the condition is True, then statements-1 will be executed If the condition is False, statements-2 will be executed



Ex1: Python program to find the larger of 2 numbers.

```
a = 10
b = 20
if a > b:
   print("a is greater than b")
else:
   print("b is greater than a")
```

Ex2: Python program to validate the given password.

```
password=input("Enter the password")
if (password == "koppal"):
   print ("Correct Password")
else:
   print ("Wrong Password")
```

Ex3: Python program to check positive number or negative number.

```
a=int(input("enter a number"))
if a>=0:
   print("positive number")
else:
   print("negative number")
```

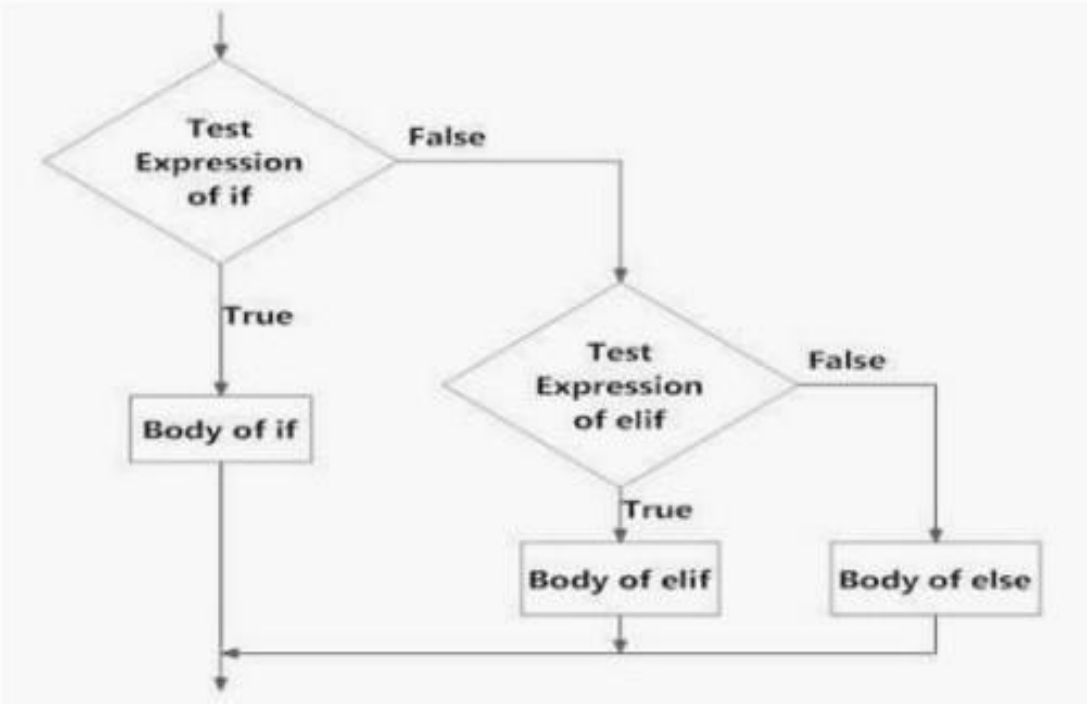
elif statement (Chain multiple if statement)

- It is a multi-way branching statement.
- The elif statement allows checking multiple expressions for TRUE and executing a block of code as soon as one of the conditions evaluates to TRUE.
- The if block can have multiple elif blocks but can have only one else block.
- Only one block among the several if…elif…else blocks is executed according to the condition.
- The syntax of elif statement is given below

Syntax:

```
if(expression1):
   statement(s)
elif(expression2):
   statement(s)
elif(expression3):
   statement(s)
.
.
else:
   statement(s)
```

Flowchart:



Ex1: To find positive, negative or zero

```
a= int(input("enter a number"))
if a>0:
   print("number is positive number")
elif a==0:
   print("number is zero")
else:
   print("number is negative number")
```

Ex2: To find largest of 3 numbers

```
a= int(input("enter a number"))
b= int(input("enter a number"))
c= int(input("enter a number"))
if (a>b and a>c):
   print(f"{a} is bigger")
elif (b>a and b>c):
   print(f"{b} is bigger")
else:
   print(f"{c} is bigger")
```

Nested-if Statements

- In Python, the nested if-else statement is an if statement inside another if-else statement.
- In Python it is allowed to put any number of if statements in another if statement.
- Indentation is the only way to differentiate the level of nesting.
- The nested if-else is useful when we want to make a series of decisions.
- In a nested if construct, can have an if...elif...else construct inside another if...elif...else construct.

Syntax:

```
if(expression1):
   if(expression2):
        statement(s)
   else:
        statements(s)
```

```
        else:
            statement(s)
```

Ex1: To find largest of 3 numbers

```
a= int(input("enter first number"))
b= int(input("enter second number"))
c= int(input("enter third number"))
if a>b:
    if a>c:
        print(f"{a} is bigger")
    else:
        print(f"{c} is bigger")
else:
    if b>c:
        print(f"{b} is bigger")
    else:
        print(f"{c} is bigger")
```

WEEK-4

Control Flow: Loops

While loop: general format; examples For loop: general format, examples. Range();nesting loops and conditional statements; Controlling loop execution: Break, continue, pass statements;

Control Flow: Loops

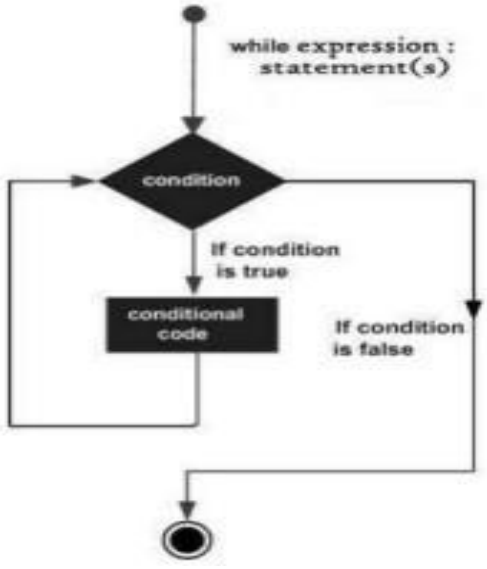
While loop:

- While Loop is used to execute a block of statements repeatedly until a given condition is **True** and when the condition becomes **False**, the line immediately after the while loop in the program is executed.
- The Syntax of while loop is given below

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

- **while** is a keyword in python.
- Here, statement(s) may be a single statement or a block of statements with uniform indent.
- The loop iterates while the condition is **True**. When the condition becomes **False**, program control passes to the line immediately following the loop.

Flowchart of While Loop :



Ex1:

```
count =0
while(count <3):
    print("Hello World")
```

```
        count =count +1
```

Output

Hello World

Hello World

Hello World

In the above example, the condition for while will be True as long as the counter variable (count) is less than 3.

Ex2:

```
i=1
while(i <= 5):
        print(i)
i=i+1
```

Output

1

2

3

4

5

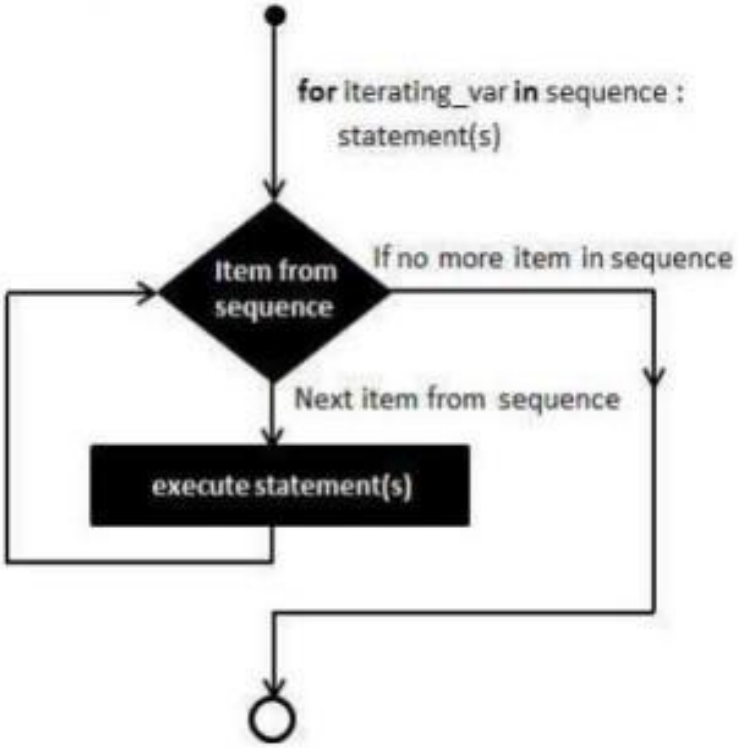
for Loop:

- A **for** loop is used for iterating over a sequence.
- A **for** loop is used for sequential traversal i.e. it is used for iterating over an iterable like string, tuple, list, dictionary etc.
- The Syntax of for loop is given below

```
        for var in sequence:
                statement(s)
```

- The first item in the sequence is assigned to the iterating variable var. Next, the statements block is executed.
- Each item in the list is assigned to var, and the statement(s) block is executed until the entire sequence is exhausted.

Flowchart of for loop



Ex1:

```
for x in "koppal":
    print(x)
```

output:

k

o

p

p

a

l

Ex2: Traversal of List sequence

```
fruits = ['banana', 'apple', 'mango']
for f in fruits:
        print (f)
```

output:

banana

apple

mango

Ex3: range() function

```
for i in range(10):
    print(i)
```

range() function:

- The range() function is used to generate a sequence of numbers.
- The range() function is a built-in function of Python.
- The range() function allows the user to generate a series of numbers within a given range.
- The range() works as it produces a list of numbers from zero to the value minus one.
- The syntax of range() is given below

  Syntax

  ```
  range([start,] stop [,step])
  ```

  where,

  - ✓ **start**: integer starting from which the sequence of integers is to be returned.
  - ✓ **stop**: integer before which the sequence of integers is to be returned.
  - ✓ **step**: integer value which determines the increment between each integer in the sequence.
- The 3 general forms of range are given below
  1. range(stop)
  2. range(start, stop)
  3. range(start, stop, step)
- range(5) produces five values: 0, 1, 2, 3, and 4.
- range(3,8) will produce the list 3, 4, 5, 6, 7

Statement	Values generated
range(10)	0,1,2,3,4,5,6,7,8,9
range(1,10)	1,2,3,4,5,6,7,8,9
range(3,7)	3,4,5,6
range(2,15,3)	2,5,8,11,14
range(9,2,-1)	9,8,7,6,5,4,3

Ex1: Create a sequence of numbers from 0 to 5, and print each item in the sequence:

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

Output

0
1
2
3
4
5

Ex2: Program that counts down from 5 and then prints a message.

```
for i in range(5,0,-1):
        print(i, end=' ')
print('End!!')
```

Output

5 4 3 2 1 End!!

The **end=' '** just keeps everything on the same line.

Using else Statement with Loops:

- Python supports having an else statement associated with a loop statement.
- If the else statement is used with a **for** loop, the else statement is executed when the loop has exhausted iterating the list.
- If the else statement is used with a **while** loop, the else statement is executed when the condition becomes false.

Ex1:

```
count = 0
while(count < 5):
        print (count, " is less than 5")
        count = count + 1
else:
         print (count, " is not less than 5")
```

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

0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5

Ex2:

```
num=[11,33,55,39,55,75,37,21,23,41,13]
for n in num:
```

```
            if n%2==0:
                print ("the list contains an even number")
                break
        else:
            print ("the list does not contain even number")
```

Output
the list does not contain even number

Nested loops:

- Python programming language allows the use of one loop inside another loop.
- Syntax for a nested for loop statement in Python programming language is as follows

```
for var in sequence:
    for var in sequence:
        statements(s)
    statements(s)
```

- Syntax for a nested while loop statement in Python programming language is as follows

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

Ex:

```
for i in range(1,10):
    for j in range(1,10):
        print (i*j)
```

Loop Control Statements:

The Loop control statements change the execution from its normal sequence.

Python supports the following control statements:

break statement :

- Terminates the loop statement and transfers the execution to the statement immediately following the loop.
- It is used in both while and for loops.
- It is usually associated with if statement.
- If you are using nested loops, the break statement stops the execution of the innermost loop and starts executing the next line of the code after the block.
- Syntax for a break statement in Python is as follows

break

Ex1:

```
for letter in 'python':
    if letter == 'h':
        break
    print (letter)
```

output:

p
y
t

Ex2:

```
for i in range(10):
    if  i==5:
        break
    print (i)
```

output:

0
1
2
3
4

continue statement:

- It is used to skip the current iteration and continue with the next iteration.
- It is present only inside a loop.
- It is usually associated with if statement.
- Syntax for a continue statement in Python is as follows

continue

Ex1:

```
for letter in 'python':
    if letter == 'h':
        continue
    print (letter)
```

output:

p
y
t
o

n

Ex2:

```
for i in range(5):
    if  i==2:
        continue
    print (i)
```

output:

0
1
3
4

pass statement:

- The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.
- It is a null operation, nothing happens when it is executed.
- It acts as a placeholder with empty body.
- It is used in both loops and functions.
- Syntax for a pass statement in Python is as follows

```
pass
```

Ex1:

```
for letter in 'python':
    pass
print (letter)
```

output:

n

WEEK-5

Data Collections: Concept of mutability, Set – features, declaration, initialization, operations, comprehension;

Tuple-features; declaration, initialization, basic operations; indexing; slicing; built in functions; Nested tuples;

Set:

- Set is a collection which is unordered, unchangeable and unindexed.
- Sets are used to store multiple items in a single variable.
- Sets are written with curly braces { }.
- It is one of the 4 built-in data types in python to store collection of data. The other 3 are list, tuple and dictionary.

Set Features:

- It is represented using curly braces { } with comma separation.
- Set items can be of any data type. (heterogeneous collection)
- Order of elements is not preserved.
- Each element must be unique. (no duplication)
- We cannot change its items. But we can add new item or remove items from set.
- Indexing and slicing operations are not applicable.
- We can perform union, intersection and difference operations on set elements.

Declaration and Initialization:

- Set can be created by enclosing the comma separated items with the curly braces.

```
s1={10,20,30,40}
fruits={"apple","banana","orange"}
s2={10,36.7,"banana",True}
print(s1)
print(fruits)
print(s2)
```

output:

```
{40, 10, 20, 30}
{'banana', 'apple', 'orange'}
{'banana', True, 10, 36.7}
```

- A set () function is used to create an empty set and to convert other data collections to set.

```
s1=set()
print(s1)
```

output:

```
set()
```

String to Set:

```
s2=set("koppal")
print(s2)
```

output:

```
{'a', 'o', 'k', 'l', 'p'}
```

List to Set:

```
l1=[20,30,40,50]
s3=set(l1)
print(s3)
```

output:

```
{40, 50, 20, 30}
```

Tuple to set:

```
t1=(5,7,9,11)
s4=set(t1)
print(s4)
```

output:

```
{9, 11, 5, 7}
```

Range to Set:

```
s5=set(range(1,20,4))
print(s5)
```

output:

```
{1, 5, 9, 13, 17}
```

Basic Operations:

Adding items to the set:

add(): The add() function is used to add a single element to the list.

Ex:

```
s1={10,20,30,40}
```

```
print(s1)
s1.add(50)
print(s1)
s1.add(60)
print(s1)
```

output:

```
{40, 10, 20, 30}
{40, 10, 50, 20, 30}
{40, 10, 50, 20, 60, 30}
```

update(): The update() function is used to add multiple elements to the set.

Ex:

```
s1={10,20,30,40}
print(s1)
s1.update([5,7,9])
print(s1)
s1.update((4,8,12))
print(s1)
```

output:

```
{40, 10, 20, 30}
{5, 7, 40, 9, 10, 20, 30}
{4, 5, 7, 40, 9, 10, 8, 12, 20, 30}
```

Ex:

```
s1={10,20,30,40}
print(s1)
l1=[5,7,9]
s1.update(l1)
print(s1)
t1=(4,8,12)
s1.update(t1)
print(s1)
```

output:

```
{40, 10, 20, 30}
{5, 7, 40, 9, 10, 20, 30}
```

{4, 5, 7, 40, 9, 10, 8, 12, 20, 30}

Ex:

```
s1={10,20,30,40}
s2={50,60,70,80}
print(s1)
print(s2)
s1.update(s2)
print(s1)
print(s2)
```

output:

```
{40, 10, 20, 30}
{80, 50, 60, 70}
{70, 40, 10, 80, 50, 20, 60, 30}
{80, 50, 60, 70}
```

Removing items from the set:

- The remove() and discard() method is used to remove the items from the set.
- If the item to be removed does not exist, remove() method will raise an error.
- If the item to be removed does not exist, discard() method will not raise an error.

Ex:

```
s1={10,20,30,40,50,60}
print(s1)
s1.remove(10)
print(s1)
s1.discard(20)
print(s1)
```

output:

```
{40, 10, 50, 20, 60, 30}
{40, 50, 20, 60, 30}
{40, 50, 60, 30}
```

pop():

- We can use pop() method to remove an item from the set.
- This method will remove the last item from the set.
- The return value of the pop() method is the removed item.

```
s1={10,20,30,40,50,60}
print(s1)
s1.pop()
print(s1)
```

output:

```
{40, 10, 50, 20, 60, 30}
{10, 50, 20, 60, 30}
```

Mathematical operations such as union, intersection, difference and symmetric difference can be performed on sets.

Union of 2 sets:

- The **union**() method is used to join 2 or more sets.
- This method returns a new set containing all the items from both sets.
- It will remove the duplicates.

Ex:

```
s1={5,10,15,20}
s2={10,20,30,40}
s3=s1.union(s2)
print(s1)
print(s2)
print(s3)
```

output:

```
{10, 20, 5, 15}
{40, 10, 20, 30}
{5, 40, 10, 15, 20, 30}
```

Intersection of 2 sets:

- The **intersection()** method will return a new set that only contains the items that are present in both the sets.

Ex:

```
s1={5,10,15,20}
s2={10,20,30,40}
s3=s1.intersection(s2)
print(s1)
print(s2)
print(s3)
```

output:

{10, 20, 5, 15}

{40, 10, 20, 30}

{10, 20}

Difference of 2 sets:

- The **difference()** method will return a new set containing the difference between 2 or more sets.
- It is denoted by A-B, the resulting set will contains the items that are present only in A but not in B.

Ex:

```
s1={5,10,15,20}
s2={10,20,30,40}
s3=s1.difference(s2)
print(s1)
print(s2)
print(s3)
```

output:

{10, 20, 5, 15}

{40, 10, 20, 30}

{5, 15}

Symmetric Difference:

- The **symmetric_difference()** method removes the items that are present in both the sets.

Ex:

```
s1={5,10,15,20}
s2={10,20,30,40}
s3=s1.symmetric_difference(s2)
print(s1)
print(s2)
print(s3)
```

output:

{10, 20, 5, 15}

{40, 10, 20, 30}

{5, 40, 30, 15}

Tuple:

- Tuples are used to store multiple items in a single variable.
- A tuple is a collection which is ordered and **unchangeable**.
- Tuples are written with round brackets ( ).

Features of Tuple:

- It is represented using parenthesis ( ) with comma separation.
- Tuple items can be of any data type. (heterogeneous collection)
- Order of elements is preserved in tuple.
- We cannot change the content of tuple items.
- Each element need not be unique. (allows duplication)
- Indexing operation is supported.
- Slicing operation is supported.

Declaration and Initialization:

- Tuple can be created by enclosing the comma separated items with the parenthesis ( ).

```
t1 = (20,40,60,80,100)
t2 = ("apple","banana","orange")
t3 = (10,True,56.78,"apple")
print(t1)
print(t2)
print(t3)
```

output:

```
 (20, 40, 60, 80, 100)
('apple', 'banana', 'orange')
(10, True, 56.78, 'apple')
```

- It is also possible to use the **tuple**() constructor to create a tuple.

```
t1 = tuple(("apple", "banana", "cherry"))
print(t1)
```

output:

```
 ('apple', 'banana', 'cherry')
```

List to tuple:

```
l1=[10,20,30,40]
t1=tuple(l1)
```

```
print(t1)
```

output:

```
(10, 20, 30, 40)
```

Set to tuple:

```
s1={10,20,30,40}
t1=tuple(s1)
print(t1)
```

output:

```
(40, 10, 20, 30)
```

String to tuple:

```
t1=tuple("koppal")
print(t1)
```

output:

```
('k', 'o', 'p', 'p', 'a', 'l')
```

Range to tuple:

```
t1=tuple(range(1,20,3))
print(t1)
```

output:

```
(1, 4, 7, 10, 13, 16, 19)
```

Basic Operations:

Concatenation operator:

We can add the elements of 2 or more tuples using concatenation operator +.

```
t1 = (20,40,60)
t2 = (23,48,56)
t3 = t1 + t2
print(t3)
```

output:

```
(20, 40, 60, 23, 48, 56)
```

Repetition Operator:

To repeat the elements present inside the tuple for specified number of times, use the repetition operator *.

```
t1=(10,20,30)
print(t1*3)
```

output:

(10, 20, 30, 10, 20, 30, 10, 20, 30)

Membership operators (in, not):

```
t1=(10,20,30)
print(10 in t1)
print(50 in t1)
print(10 not in t1)
print(50 not in t1)
```

output:

```
True
False
False
True
```

Deleting a tuple:

The entire tuple elements can be deleted by using the **del** keyword.

```
t1=(10,20,30,40)
del(t1)
```

Length of a tuple:

The **len** is used to find the length of the tuple

```
t1 = (20,40,60,80,100)
t2 = ("apple","banana","orange")
print(len(t1))
print(len(t2))
```

output:

```
5
3
```

Indexing:

- ✓ Tuple items are ordered, so they can be accessed by their index.
- ✓ The items of the tuple can be accessed by using the index [ ] operator.
- ✓ The first item has an index 0, the second item has an index 1, and next items have an index 2, 3, 4 and up to length-1.
- ✓ The index value must be an integer.

- ✓ The tuple supports both positive and negative Indexing.
- ✓ The positive indexing means forward direction (left to right)
- ✓ The negative indexing means backward direction (right to left)
- ✓ Nested tuple can be accessed using nested indexing.
- ✓ The representation of indexing is given below.

Ex:

```
t1 = tuple(("apple", "banana", "cherry"))
print("Positive Indexing")
print(t1[0])
print(t1[1])
print(t1[2])
print("Negative Indexing")
print(t1[-1])
print(t1[-2])
print(t1[-3])
```

output:

```
Positive Indexing
apple
banana
cherry
Negative Indexing
cherry
banana
apple
```

Slicing:

- ✓ To print the specific part of items from the tuple we use the slice operation.
- ✓ The syntax of slice operator is given below

[begin:end:step]

- ✓ Begin: beginning of a object
- ✓ End: end of a object
- ✓ Step: increment for each slice operation.
- ✓ The step value can be positive or negative.

- ✓ If the step value is positive, slicing will happen in forward direction. (left to right).
- ✓ If the step value is negative, slicing will happen in backward direction. (right to left).
- ✓ For forward direction:
  - The default value for begin is 0.
  - The default value for end is the length of the list.
  - The default value for step is +1.

Ex:

```
t1 = (10,20,30,40,50,60,70,80)
print(t1[2:])
print(t1[:4])
print(t1[2:5])
print(t1[0:6:2])
print(t1[::2])
print(t1[::])
print(t1[-4:-2])
print(t1[::-1])
```

output:

```
 (30, 40, 50, 60, 70, 80)
(10, 20, 30, 40)
(30, 40, 50)
(10, 30, 50)
(10, 30, 50, 70)
(10, 20, 30, 40, 50, 60, 70, 80)
(50, 60)
(80, 70, 60, 50, 40, 30, 20, 10)
```

Ex2:

```
t1 = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(t1[:4])
print(t1[2:])
print(t1[-4:-1])
```

output:

```
 ('apple', 'banana', 'cherry', 'orange')
('cherry', 'orange', 'kiwi', 'melon', 'mango')
```

('orange', 'kiwi', 'melon')

Built in Functions of tuple:

Sl No	Function	Description
1	cmp(t1,t2)	It compares 2 tuples and returns True if t1 is greater than t2 otherwise False
2	len(tuple)	It returns the length of the tuple.
3	max(tuple)	It returns the maximum item of the tuple.
4	min(tuple)	It returns the minimum item of the tuple.
5	count()	It returns the number of occurrence of the specified item present in the tuple.
6	index()	It returns the first occurrence of the specified item present in the tuple.
7	reversed()	This function is used to print the tuple in reverse order.
8	tuple(sequence)	It converts the specified sequence to the tuple.

Ex:

```
t1=(43,26,33,41)
print(len(t1))
print(max(t1))
print(min(t1))
t2=(10,20,30,40,20,30,20)
print(t2.count(20))
print(t2.index(30))
```

output:

```
4
43
26
3
2
```

Nested tuple:

- ✓ A tuple inside another tuple is known as nested list.
- ✓ Nested indexing is used to access the nested elements.

Ex:

```
t=(4,5,6,(8,9),(10,20,30))
print(t)
print(t[2])
print(t[3])
print(t[4])
print(t[3][1])
print(t[4][2])
```

output:

```
(4, 5, 6, (8, 9), (10, 20, 30))
6
(8, 9)
(10, 20, 30)
9
30
```

WEEK-6

List features; declaration, initialization, basic operations; indexing; List iterations; Slicing; built in functions; Nested Lists; Comprehensions; Applications

List:

- List is a collection which is ordered, changeable and indexed.
- Lists are used to store multiple items in a single variable.
- Lists are written with square brackets [ ]

List Features:

- It is represented using square brackets [ ] with comma separation.
- List items can be of any data type. (heterogeneous collection)
- Lists are ordered i.e. order of elements is preserved.
- Lists are mutable. i.e. we can modify the contents of list items.
- List items are accessed by their index.
- List items need not be unique. (It allows duplication)
- Slicing operation is supported.
- Lists can be nested.

Declaration and Initialization:

- An empty list can be created by using the square brackets [ ] as follows.

```
l1=[]
print(l1)
```

output:

```
[ ]
```

- List with items can be created by enclosing the comma separated items within the square brackets [ ].

```
l1=[10,20,30,40]
fruits=["apple","banana","orange"]
l2=[10,36.7,"banana",True]
print(l1)
print(fruits)
print(l2)
```

output:

```
[10, 20, 30, 40]
```

['apple', 'banana', 'orange']

[10, 36.7, 'banana', True]

- A list () function is used to convert other data collections to list.
- Tuple to list:

```
t=(10,20,30,40)
l1=list(t)
print(l1)
```

output:

[10, 20, 30, 40]

- String to list:

```
l2=list("koppal")
print(l2)
```

output:

['k', 'o', 'p', 'p', 'a', 'l']

```
s1="welocome to python programming"
x=s1.split()
print(x)
```

output:

['welocome', 'to', 'python', 'programming']

- Set to list:

```
s1={20,30,40,50}
l3=list(s1)
print(l3)
```

output:

[40, 50, 20, 30]

- range to list:

```
l4=list(range(1,20,4))
print(l4)
```

output:

[1, 5, 9, 13, 17]

Basic operations:

1. Access list items in Lists:

Index is used to access the list items.

```
x=[10,20,30,40,50]
print(x[2])
print(x[4])
```

output:

```
30
50
```

2. Update list items:

We can update single value or multiple values of a list.

```
x=[10,20,30,40,50]
x[0]="gpt"
x[2]="koppal"
print(x)
```

output:

```
['gpt', 20, 'koppal', 40, 50]
```

3. Delete list items:

The del() can be used to delete a single item or whole list.

```
x=[10,20,30,40,50]
del(x[2])
print(x)
del(x)
```

output:

```
[10, 20, 40, 50]
```

4. Concatenation operator (+):

Concatenate operator + is used to join or merge 2 or more lists.

```
l1=[10,20,30]
l2=[50,60,70]
l3=l1+l2
print(l3)
```

output:

```
 [10, 20, 30, 50, 60, 70]
```

5. Repetition Operator (*):

```
l1=[10,20,30]
```

```
l2= l1*3
print(l2)
```

output:

```
[10, 20, 30, 10, 20, 30, 10, 20, 30]
```

6. Membership Operators (in, not)

```
l1=[10,20,30]
print(10 in l1)
print(50 in l1)
print(10 not in l1)
print(50 not in l1)
```

output:

```
True
False
False
True
```

7. Traversing List items

```
l1=[10,20,30,40,50]
for x in l1:
   print(x)
```

output:

```
10
20
30
40
50
```

Indexing:

- ✓ List items are ordered, so they can be accessed by their index.
- ✓ The items of the list can be accessed by using the index [ ] operator.
- ✓ The first item has an index 0, the second item has an index 1, and next items have an index 2, 3, 4 and up to length-1.
- ✓ The index value must be an integer.
- ✓ The List supports both positive and negative Indexing.

- ✓ The positive indexing means forward direction (left to right)
- ✓ The negative indexing means backward direction (right to left)
- ✓ Nested list can be accessed using nested indexing.
- ✓ The representation of indexing is given below.

Ex:

```
x = [10,20,30,40,50]
print("Positive Indexing")
print(x[0])
print(x[3])
print(x[2])
print("Negative Indexing")
print(x[-1])
print(x[-3])
print(x[-5])
```

Output:

```
Positive Indexing
10
40
30
Negative Indexing
50
30
10
```

Slicing:

- ✓ To print the specific part of items from the list we use the slice operation.
- ✓ The syntax of slice operator is given below

[begin:end:step]

- ✓ Begin: beginning of a object
- ✓ End: end of a object
- ✓ Step: increment for each slice operation.
- ✓ The step value can be positive or negative.
- ✓ If the step value is positive, slicing will happen in forward direction. (left to right).

- ✓ If the step value is negative, slicing will happen in backward direction. (right to left).
- ✓ For forward direction:
  - The default value for begin is 0.
  - The default value for end is the length of the list.
  - The default value for step is +1.

Ex:

```
x = [1,2,3,4,5,6,7,8]
print(x[2:])
print(x[4:])
print(x[:4])
print(x[1:5])
print(x[0:6:2])
print(x[::-1])
print(x[:-6])
print(x[-6:-1])
```

Output:

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

Ex2:

```
x = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(x[:4])
print(x[2:])
print(x[-4:-1])
```

Output:

```
['apple', 'banana', 'cherry', 'orange']
['cherry', 'orange', 'kiwi', 'melon', 'mango']
['orange', 'kiwi', 'melon']
```

List Iterations:

✓ The for loop is used to iterate the contents of list items.

```
l1=[10,20,30,40,50]
for x in l1:
   print(x)
```

output:

```
10
20
30
40
50
```

✓ The while loop can also be used to iterate the list items.

```
l1=[10,20,30,40,50]
i=0
while i<len(l1):
   print(l1[i])
   i=i+1
```

output:

```
10
20
30
40
50
```

✓ Using list comprehension also we can iterate the list items.

```
l1=[10,20,30,40,50]
[print(i) for i in l1]
```

output:

```
10
20
30
40
50
```

Built-in Functions:

1. **append():** The append()function is used to add a item at the end of the list.

Ex:

```
x=[10,20,30,40]
print(x)
x.append(50)
print(x)
x.append((5,6))
print(x)
x.append([7,8])
print(x)
```

output:

```
[10, 20, 30, 40]
[10, 20, 30, 40, 50]
[10, 20, 30, 40, 50, (5, 6)]
[10, 20, 30, 40, 50, (5, 6), [7, 8]]
```

2. **insert()**

- ✓ The insert()function is used to add an item at the desired position.
- ✓ This method takes 2 arguments.
- ✓ The syntax of insert() function is given below

insert(position,value)

Ex:

```
x=[10,20,30,40]
x.insert(3,25)
print(x)
x.insert(2,[2,3])
print(x)
```

output:

```
[10, 20, 30, 25, 40]
[10, 20, [2, 3], 30, 25, 40]
```

3. **extend():** The extend() function is used to add multiple elements at the same time at the end of the list.

Ex:

```
x=[10,20,30,40]
x.extend([3,4,5])
print(x)
x.extend((2,3,6))
print(x)
```

output:

```
[10, 20, 30, 40, 3, 4, 5]
[10, 20, 30, 40, 3, 4, 5, 2, 3, 6]
```

4. remove()

- ✓ The remove() method is used to remove elements from the list.
- ✓ It removes only one element at a time.
- ✓ It removes the specified item.
- ✓ It gives an error if element does not exist in list.

Ex:

```
x=[10,20,30,40]
print(x)
x.remove(20)
print(x)
```

output:

```
[10, 20, 30, 40]
[10, 30, 40]
```

5. pop()

- ✓ By default, the pop() method will remove the last element from the list.
- ✓ The pop() with specified position will remove the specified element from the list.

Ex1:

```
x=[10,20,30,40]
print(x)
x.pop()
print(x)
```

output:

```
[10, 20, 30, 40]
[10, 20, 30]
```

Ex2:

```
x=[10,20,30,40]
print(x)
x.pop(1)
print(x)
```

output:

```
[10, 20, 30, 40]
[10, 30, 40]
```

6. **clear() :** This function is used to erase all the elements from the list.

Ex:

```
x=[10,20,30,40]
print(x)
x.clear()
print(x)
```

output:

```
[10, 20, 30, 40]
[ ]
```

7. **count():** This function returns the number of occurrences of the specified element from the list.

Ex:

```
x=[10,20,30,40,20,30,40,20,30]
print(x.count(20))
```

output:

```
3
```

8. **index():** This function returns the index of the first occurrence of the specified value.

```
x=[10,30,30,40,20,40,20]
print(x.index(20))
```

output:

```
4
```

9. **sort()**

- ✓ Sort means arranging items either in ascending order or in descending order.
- ✓ The sort() function sorts the items in ascending order.

Ex:

```
x=[10,30,40,20,50]
x.sort()
```

```
print(x)
y=["cherri","kiwi","banana","orange"]
y.sort()
print(y)
```

output:

```
[10, 20, 30, 40, 50]
['banana', 'cherri', 'kiwi', 'orange']
```

10. reverse() : This function reverses the elements of the given list.

Ex:

```
x=[10,20,30,40,50]
print(x)
x.reverse()
print(x)
```

output:

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

11. copy(): This function returns the copy of the list.

Ex:

```
x=[10,20,30,40,50]
print(x)
y=x.copy()
print(y)
```

output:

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

Nested List:

- ✓ A List inside another list is known as nested list.
- ✓ Duplicate elements are also allowed.
- ✓ Nested indexing is used to access the nested elements.

Ex:

```
l=[4,5,6,[8,9],[10,20,30]]
```

```
print(l)
print(l[2])
print(l[3])
print(l[4])
print(l[3][1])
print(l[4][2])
```

output:

```
[4, 5, 6, [8, 9], [10, 20, 30]]
6
[8, 9]
[10, 20, 30]
9
30
```

List Comprehension:

List comprehension offers a shorter syntax when we want to create a new list based on the values of an existing list.

The Syntax

newlist = [expression for item in iterable if condition == True]

The return value is a new list, leaving the old list unchanged.

Condition: The condition is like a filter that only accepts the items that valuate to True.

Iterable: The iterable can be any iterable object, like a list, tuple, set etc.

Expression: The expression is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list.

Ex:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
```

WEEK-7

Dictionary: Features, declaration, initialization, basic operations; indexing; adding and removing keys, iterating through dictionaries; built in functions; Comprehensions; Applications.

Dictionary:

- Dictionaries are used to store data values in key-value pairs.
- A dictionary is a collection which is ordered, changeable and do not allow duplicates.
- Dictionaries are written with flower brackets { }.

Features:

- It is an ordered collection that contains key - value pairs separated by comma inside curly braces {}.
- Each key is separated from its value by a colon (:)
- Dictionaries are ordered i.e. order of elements is preserved.
- Dictionaries are changeable, means we can change, add or remove items from the dictionary.
- Duplicate keys are not allowed i.e. dictionaries cannot have 2 items with the same key.
- Dictionary keys are case sensitive.
- Keys must be single element.
- Values in dictionary can be of any data type and can be duplicated. Ex: list, tuple ,string, integer, etc
- Dictionaries are indexed by keys.
- Slicing concept is not supported.

Declaration and Initialization:

- An empty dictionary can be created by using the curly braces { } as follows.

```
d1={}
print(d1)
```

output:

```
{ }
```

- A dictionary is created by placing a sequence of elements within curly braces {}, separated by comma and each key is separated from its value by the colon (:)

```
d1={1:'cs',2:'ec',3:'mech'}
d2={'name':'ravi','age':50,'marks':[45,56,78]}
print(d1)
print(d2)
```

output:

```
{1: 'cs', 2: 'ec', 3: 'mech'}
```

{'name': 'ravi', 'age': 50, 'marks': [45, 56, 78]}

- A dictionary can also be created by using the dict() function.

Ex:

```
d1=dict([(1,'cs'),(2,'ec'),(3,'mech')])
d2=dict([(1,'sun'),(2,'mon'),(3,'tues')])
print(d1)
print(d2)
```

output:

```
{1: 'cs', 2: 'ec', 3: 'mech'}
{1: 'sun', 2: 'mon', 3: 'tues'}
```

Basic Operations:

1. Accessing elements from a dictionary:

A dictionary holds data in key-value pair. The value can be accessed by specifying the key.

```
car={"brand":"Ford","year":2002}
print(car['brand'])
print(car['year'])
d1={1:'cs',2:'ec',3:'mech'}
print(d1[1])
print(d1[2])
print(d1[3])
```

output:

```
Ford
2002
cs
ec
mech
```

2. Changing Dictionary Items:

The value of a key-value pair can be changed by referring to the key.

```
car={"brand":"Ford","year":2002}
print(car)
car['year']=2018
print(car)
```

output:

```
{'brand': 'Ford', 'year': 2002}
{'brand': 'Ford', 'year': 2018}
```

3. Membership operators:(in, not)

```
d = {37:'koppal', 35:'hospet'}
print(37 in d)
print(38 in d)
print(37 not in d)
print(38 not in d)
```

output:

```
True
False
False
True
```

4. Concatenation Operator (+):

```
d1={1:'cs',2:'ec'}
d2={3:'mech',4:'civil'}
d3=d1+d2
print(d3)
```

5. Repetition operator(*):

```
d1={1:'cs',2:'ec',3:'mech'}
d2=d1*3
print(d2)
```

Indexing:

In dictionary, key is used to access the content of value.

```
car={"brand":"Ford","year":2002,"price":50000}
print(car['brand'])
print(car['year'])
print(car['price'])
d1={1:'cs','name':'ravi','age':48,'marks':[45,67,56]}
print(d1['name'])
print(d1['age'])
```

```
print(d1['marks'])
print(d1[1])
```

output:

```
Ford
2002
50000
ravi
48
[45, 67, 56]
cs
```

Adding and Removing Keys:

The dictionary is a mutable data type, and its value can be updated by using the specific keys.

The new key value pair can be added or removed easily from the dictionary.

Adding Keys (Elements):

A new key-value pair can be added as follows.

```
d={1:'cs',2:'ec'}
d[3]='mech'
d[4]='civil'
print(d)
```

output:

```
{1: 'cs', 2: 'ec', 3: 'mech', 4: 'civil'}
```

The **update**() method can also be used to add new key-value pairs to the dictionary.

```
car={"brand":"Ford","year":2002}
car.update({'electric':False})
print(car)
```

output:

```
{'brand': 'Ford', 'year': 2002, 'electric': False}
```

Removing Keys:

A specific key-value pair can be removed from the dictionary using **del** keyword or by using **pop**() method.

Ex1:

```
d={1: 'cs', 2: 'ec', 3: 'mech', 4: 'civil'}
del d[2]
```

```
print(d)
```

output:

```
{1: 'cs', 3: 'mech', 4: 'civil'}
```

Ex2:

```
d={1: 'cs', 2: 'ec', 3: 'mech', 4: 'civil'}
d.pop(2)
print(d)
```

output:

```
{1: 'cs', 3: 'mech', 4: 'civil'}
```

The **popitem**() method is used to remove the last item from the dictionary.

```
d={1: 'cs', 2: 'ec', 3: 'mech', 4: 'civil'}
d.popitem()
print(d)
```

output:

```
{1: 'cs', 2: 'ec', 3: 'mech'}
```

The **clear**() method is used to remove all the elements from the dictionary

```
d={1: 'cs', 2: 'ec', 3: 'mech', 4: 'civil'}
d.clear()
print(d)
```

output:

```
{ }
```

Iterating Through Dictionaries:

For loop is used to iterate all the key-value pairs of a dictionary.

1) To iterate keys:

```
emp={'name':'ravi','age':30,'salary':50000}
for x in emp:
    print(x)
```

output:

```
name
age
salary
```

2) To iterate values:

```
emp={'name':'ravi','age':30,'salary':50000}
for x in emp:
   print(emp[x])
```

output:

```
ravi
30
50000
```

3) To iterate values using values() method:

```
emp={'name':'ravi','age':30,'salary':50000}
for x in emp.values():
   print(x)
```

output:

```
ravi
30
50000
```

4) To iterate keys using keys() method:

```
emp={'name':'ravi','age':30,'salary':50000}
for x in emp.keys():
   print(x)
```

output:

```
name
age
salary
```

5) To iterate both key-values using items() method:

```
emp={'name':'ravi','age':30,'salary':50000}
for x in emp.items():
   print(x)
```

output:

```
 ('name', 'ravi')
('age', 30)
('salary', 50000)
```

Dictionary Built-in Functions:

1. keys():

The keys() method will return a list of all the keys in the dictionary.

```
emp={'name':'ravi','age':30,'salary':50000}
x = emp.keys()
print(x)
```

output:

```
dict_keys(['name', 'age', 'salary'])
```

2. values():

The values() method will return a list of all the values in the dictionary.

```
emp={'name':'ravi','age':30,'salary':50000}
x = emp.values()
print(x)
```

output:

```
dict_values(['ravi', 30, 50000])
```

3. items():

The items() method will return each item in a dictionary as tuples in list.

```
emp={'name':'ravi','age':30,'salary':50000}
x = emp.items()
print(x)
```

output:

```
dict_items([('name', 'ravi'), ('age', 30), ('salary', 50000)])
```

4. copy():

The copy() method returns the shallow copy of the dictionary.

```
emp1={'name':'ravi','age':30,'salary':50000}
print(emp1)
emp2=emp1.copy()
print(emp2)
```

output:

```
{'name': 'ravi', 'age': 30, 'salary': 50000}
{'name': 'ravi', 'age': 30, 'salary': 50000}
```

5. get():

The get() method returns the value of the item with the specified key.

```
emp={'name':'ravi','age':30,'salary':50000}
x=emp.get('age')
print(x)
x=emp.get('name')
print(x)
```

output:

```
30
ravi
```

6. pop():

The pop() method removes and returns the specified item from the dictionary.

```
d={1: 'cs', 2: 'ec', 3: 'mech', 4: 'civil'}
x = d.pop(2)
print("The deleted item =",x)
print(d)
```

output:

```
The deleted item = ec
{1: 'cs', 3: 'mech', 4: 'civil'}
```

7. popitem():

The popitem() method removes the last inserted key-value pair from the dictionary.

```
d={1: 'cs', 2: 'ec', 3: 'mech', 4: 'civil'}
x = d.popitem()
print("The deleted item =",x)
print(d)
```

output:

```
The deleted item = (4, 'civil')
{1: 'cs', 2: 'ec', 3: 'mech'}
```

8. clear():

The clear() method removes all items from the dictionary.

```
d={1: 'cs', 2: 'ec', 3: 'mech', 4: 'civil'}
```

```
d.clear()
print(d)
```

output:

```
{ }
```

9. update():

The update() method updates the dictionary with the elements from another dictionary or from an iterable of key-value pairs.

```
d1={1: 'cs', 2: 'ec'}
d2={3: 'mech', 4: 'civil'}
d1.update(d2)
print(d1)
```

output:

```
{1: 'cs', 2: 'ec', 3: 'mech', 4: 'civil'}
```

Dictionary comprehension:

Dictionary comprehension is an elegant and concise way to create dictionaries.

The minimal syntax for dictionary comprehension is:

dictionary = {key: value for vars in iterable}

```
square_dict = {}
for num in range(1, 5):
   square_dict[num] = num * num
print(square_dict)
```

output:

```
{1: 1, 2: 4, 3: 9, 4: 16}
```

Dictionary comprehension:

```
square_dict = {num: num * num for num in range(1, 5)}
print(square_dict)
```

output:

```
{1: 1, 2: 4, 3: 9, 4: 16}
```

WEEK-8

Arrays and Strings:

Arrays: features; create, initialize, indexing, traversal, manipulation;

Strings: create, assign, indexing, built in functions;

Arrays:

- Arrays are used to store multiple values in one single variable.
- The main difference between arrays and lists. While python lists can contain values corresponding to different data types, arrays in python can only contain values corresponding to same data type.
- To use arrays in python language, we need to import the standard array module. This is because array is not a fundamental data type like strings, integer etc.

Features:

- Arrays are ordered collection of items.
- Arrays are homogenous collection of items. i.e. All the elements of array must be same numeric type.
- We can store only numeric values.
- Arrays are mutable. i.e. we can modify the contents of array elements.
- It supports both indexing and slicing.

Creating and initializing an array:

- Array is created in Python **by importing array module** to the python program.
- If we create arrays using the array module, elements of the array must be of the same numeric type.
- The syntax for creating an array is given below

```
import array as arr
arrayName = arr.array(typecode[,arrayitems])
                    OR
from array import *
arrayName = array(typecode, [arrayitems])
```

Ex:

```
import array as arr
a = arr.array('i', [10, 23, 35])
print(a)
```

output:

```
array('i', [10, 23, 35])
```

OR

Commonly used type codes are listed as follows:

Code	C Type	Python Type	Min bytes
b	signed char	int	1
B	unsigned char	int	1
u	Py_UNICODE	Unicode	2
h	signed short	int	2
H	unsigned short	int	2
i	signed int	int	2
I	unsigned int	int	2
l	signed long	int	4
L	unsigned long	int	4
f	float	float	4
d	double	float	8

Indexing:

- We can access each element of an array using the index of the element.
- An array supports both positive and negative indexing.
- The syntax for indexing array elements is

  arrayName[indexNum]

Ex:

```
import array as arr
a1 = arr.array('i', [10,20,30,40,50])
print (a1[0])
print (a1[2])
print (a1[4])
print (a1[-1])
print (a1[-2])
print (a1[-5])
```

output:

```
10
30
50
50
40
10
```

Slicing:

We can access a range of items in an array by using the slicing operator (**:**).

Ex:

```
import array as arr
numbers_list = [2, 5, 62, 5, 42, 52, 48, 5]
numbers_array = arr.array('i', numbers_list)
print(numbers_array[2:5]) # 3rd to 5th
print(numbers_array[:-5]) # beginning to 4th
print(numbers_array[5:]) # 6th to end
print(numbers_array[:]) # beginning to end
```

output:

```
array('i', [62, 5, 42])
array('i', [2, 5, 62])
array('i', [52, 48, 5])
array('i', [2, 5, 62, 5, 42, 52, 48, 5])
```

Array Traversal:

For loop is used to traverse the contents of array as shown below.

Ex:

```
import array as arr
a1 = arr.array('i', [10,20,30,40,50])
for x in a1:
   print(x)
a2 = arr.array('d', [22.5, 36.2, 53.3])
for i in range (0, len(a2)):
   print (a2[i], end =" ")
```

output:

```
10
20
30
40
50
22.5 36.2 53.3
```

Array manipulations:

Changing Array Elements:

Arrays are mutable; their elements can also be changed in a similar way as lists.

Ex:

```
import array as arr
numbers = arr.array('i', [1, 2, 3, 5, 7, 10])
numbers[0] = 50
print(numbers)
numbers[2:5] = arr.array('i', [40, 60, 80])
print(numbers)
```

output:

```
array('i', [50, 2, 3, 5, 7, 10])
array('i', [50, 2, 40, 60, 80, 10])
```

Adding Array Elements: insert(), append() and extend():

- We can use **insert**() method to insert an item into an array at any given index of the array.
- This method takes two arguments index and value and the syntax is given below.

arrayName.insert(index, value)

Ex:

```
import array as arr
numbers = arr.array('i', [300,200,100])
print(numbers)
numbers.insert(2, 150)
print(numbers)
```

output:

```
array('i', [300, 200, 100])
array('i', [300, 200, 150, 100])
```

- We can add one item at the end of the array using the **append**() method.
- We can add several items at the end of the array using the **extend**() method.

Ex:

```
import array as arr
numbers = arr.array('i', [1, 2, 3])
print(numbers)
numbers.append(4)
print(numbers)
```

```
numbers.extend([5, 6, 7])
print(numbers)
```

output:

```
array('i', [1, 2, 3])
array('i', [1, 2, 3, 4])
array('i', [1, 2, 3, 4, 5, 6, 7])
```

Concatenation Operator (+):

We can also concatenate two arrays using + operator.

Ex:

```
import array as arr
odd = arr.array('i', [1, 3, 5])
even = arr.array('i', [2, 4, 6])
numbers = odd + even
print(odd)
print(even)
print(numbers)
```

output:

```
array('i', [1, 3, 5])
array('i', [2, 4, 6])
array('i', [1, 3, 5, 2, 4, 6])
```

Removing Array Elements: (del, remove(), pop())

- We can delete one or more items from an array using Python's **del** keyword.

Ex:

```
import array as arr
numbers = arr.array('i', [15, 26, 37, 32, 43])
del numbers[2]
print(numbers)
del numbers
#print(numbers)  # Error: array numbers is not defined
```

output:

```
array('i', [15, 26, 32, 43])
```

- The **remove**() method is used to remove the given item.
- The **pop**(i) method with index will remove an item at the given index.

- The **pop**() method without index will remove last element.

Ex:

```
import array as arr
numbers = arr.array('i', [10, 11, 12, 13, 14])
numbers.remove(12)
print(numbers)
print(numbers.pop(2))
print(numbers)
print(numbers.pop())
print(numbers)
```

output:

```
array('i', [10, 11, 13, 14])
13
array('i', [10, 11, 14])
array('i', [10, 11])
```

index():

This function returns the **index of the first occurrence** of value mentioned in arguments.

Ex:

```
import array as arr
numbers = arr.array('i', [20, 30, 40, 50, 60])
print(numbers.index(40))
```

output:

```
2
```

reverse():

This function **reverses** the given array.

Ex:

```
import array as arr
numbers = arr.array('i', [20, 30, 40, 50, 60])
print(numbers)
numbers.reverse()
print(numbers)
```

output:

```
array('i', [20, 30, 40, 50, 60])
array('i', [60, 50, 40, 30, 20])
```

count():

We can count the occurrence of elements in the array using the **count**() method

Ex:

```
import array as arr
number = arr.array('i', [22, 32, 52, 42, 32, 30, 32])
print(number.count(32))
```

output:

```
3
```

STRINGS:

- A string is a sequence of characters.
- Computers do not deal with characters; they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0s and 1s. This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encodings used.
- In Python, a string is a sequence of Unicode characters. Python does not have a character data type,a single character is simply a string with a length of 1.

Creating and Assigning a String:

- Strings in Python can be created by enclosing characters inside a single quotes or double quotes or even triple quotes.
- Triple quotes can be used in Python but generally used to represent multiline strings.

Ex:

```
s1 = 'Hello'
print(s1)
s2 = "World"
print(s2)
s3 = '''Hello World'''
print(s3)
# triple quotes string can extend multiple lines
s4 = """Hello, welcome to
 the world of Python"""
```

```
print(s4)
```

output:

```
Hello
World
"Hello World"
Hello, welcome to
the world of Python
```

Indexing: Accessing characters in Python:

- In Python, individual characters of a String can be accessed by using the method of Indexing.
- The index is used to access a specific character of a string.
- Strings support both positive and negative indexing.
- In positive Indexing, the index value of the first character is 0.
- Negative Indexing means to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on.

Ex:

```
str = 'program'
print(str[0])
print(str[1])
print(str[6])
print(str[-1])
print(str[-2])
print(str[-6])
```

output:

```
p
r
m
m
a
r
```

String Slicing:

- To access a range of characters in the String, the method of slicing is used.
- Slicing in a String is done by using a Slicing operator (colon)

- The syntax of slice operator is **s[begin:end:step]**

Ex:

```
str = 'program'
print(str[:5])
print(str[3:])
print(str[1:4])
print(str[::])
print(str[::2])
print(str[-3:])
print(str[-3:-1])
print(str[-5:])
print(str[::-1])
```

output:

```
progr
gram
rog
program
porm
ram
ra
ogram
margorp
```

Iterating through a string:

For loop is used to iterate through a string.

Ex:

```
str = "gptkoppal"
for x in str:
    print(x)
```

output:

```
g
p
t
```

```
k
o
p
p
a
l
```

Ex2: Count the number of l's in the given string.

```
str = "hello world"
count = 0
for x in str:
   if x == 'l':
      count = count+1
print("The numbers of l's in sting is:",count)
```

output:

```
The numbers of l's in sting is: 3
```

Deleting/Updating from a String:

- Strings are immutable. This means that elements of a string cannot be changed once they have been assigned. But can simply reassign different strings to the same name.
- Cannot delete or remove characters from a string. But deleting the string entirely is possible using the del keyword.

Ex:

```
s1 = "Hello"
print("Initial String:")
print(s1)
s1 = "Welcome to the Python World"
print("Updated String:")
print(s1)
del s1
print("String s1 deleted")
#print(s1) #NameError: name 's1' is not defined
```

output:

```
Initial String:
```

Hello

Updated String:

Welcome to the Python World

String s1 deleted

String Methods / String built-in functions:

1. **upper():** This method converts all the characters of a string to upper case.

Ex:

```
s = "koppal"
print(s.upper())
```

output:

KOPPAL

2. **lower():** This method converts all the characters of a string to lower case.

Ex:

```
s = "KOPPAL"
print(s.lower())
```

output:

koppal

3. **strip():** This method removes all the white spaces before and after a string.

Ex:

```
s = "    koppal      "
print(s)
print(s.strip())
```

output:

```
    koppal
koppal
```

4. **replace():** This method replaces a substring with another string.

Ex:

```
s = "welcome to gpt koppal"
print(s)
print(s.replace('gpt','wpt'))
```

output:

welcome to gpt koppal

welcome to wpt koppal

5. **split():** This method returns a list of strings by splitting the string according to a delimiter string.

Syntax: string.split(delimeter)

Ex:

```
s1 = "welcome to gpt koppal"
x = s1.split(' ')
print(x)
s2 = "python,is,a,easy,language"
x = s2.split(',')
print(x)
```

output:

```
['welcome', 'to', 'gpt', 'koppal']
['python', 'is', 'a', 'easy', 'language']
```

6. **count():** This method counts the number of times a specified character is appeared in a given string.

Ex:

```
s = "koppal"
print(s.count('p'))
```

output:

```
2
```

7. **index():** This method returns the index of the first occurrence of a specified character of a string.

Ex:

```
s = "koppal"
print(s.index('p'))
```

output:

```
2
```

8. **len():** This method returns the length of a string.

Ex:

```
s = "koppal"
print(len(s))
```

output:

```
6
```

Week -9

Functions:

Need of functions, Types, Define Function, Calling function, Function arguments; return and yield, None keyword; scope of variables; Recursion; Anonymous functions; Sufficient examples.

Function:

- A function is a block of code which only runs when it is called.
- We can pass data known as arguments to a function.
- A function can also return data as a result.
- There are 2 types of functions.
  1. **User defined function**: These functions are defined by the user to perform some task.
  2. **Built-in function**: These functions are predefined in python.

Defining a Function:

- In python, a function is defined using the **def** keyword.
- The indented block of code will be executed when the function is called.
- The syntax of defining a function is given below.

```
def  function_name(parameters):
     statement(s)
```

Ex:

```
def display():
   print("hello world")
```

Calling a Function:

- A function can be called using its name.
- It can be called any number of times.
- When a function is called, then only the statements associated with that function will be executed.
- The syntax of calling a function is given below.

```
function_name()
        OR
function_name(parameters)
```

Ex:

```
def display():
   print("hello world")
display()
```

output:

hello world

- The function can also take some arguments that can be used in the function body.

Ex:

```
def area(radius):
    return 3.14 * radius**2
print(area(2))
```

output:

12.56

- The function can also return result (values) to the calling function using return keyword.
- The return keyword is followed by an expression.
- The expression will be evaluated first and result will be returned to the function.

Ex:

```
def add(a,b):
    return(a+b)
c=add(35,15)
print(c)
print(add(25,15))
```

output:

50

40

Function Arguments:

- The arguments are information or data which are passed to the function.
- The arguments are specified in the parenthesis.
- We can pass any number of arguments, but they must be separated by a comma.

Types of Function Arguments:

1. Fixed Arguments
2. Arbitrary Arguments
3. Keyword Arguments
4. Arbitrary Keyword Arguments
5. Default argument value
6. Passing List/tuple/string as an argument

Fixed Arguments:

In this type number of arguments of the function is fixed.

Ex:

```
def add(a,b):
   print(a+b)
add(5,10)
#add(10) => Error
#add(10,20,30)  => Error
```

output:

```
15
```

Arbitrary Arguments:

- It is possible to define a function so that any number of arguments can be passed to it.
- This is done by adding * before the argument name in function definition.
- The function will receive the input in the form of **tuple**.

Ex:

```
def disp(*args):
   for x in args:
      print(x)
disp(5,10)
disp('ravi','ramu')
```

output:

```
5
10
ravi
ramu
```

Keyword Arguments:

- Arguments can also be sent to the function with the key=value format.
- The arguments sent in this way are called keyword arguments.
- Order of arguments does not matter in this type.

Ex:

```
def disp(a,b):
   print(a)
   print(b)
```

```
disp(a='ravi',b='ramu')
```

output:

```
ravi
ramu
```

Arbitrary Keyword Arguments:

- It is possible to define a function so that any number of keyword arguments can be passed to it.
- This is done by adding ** before the argument name in function definition.
- The function will receive the input in the form of **dictionary**.

Ex:

```
def disp(**args):
    print(args['brand'])
    print(args['year'])
disp(brand='ford',year=2000,cost=50000)
```

output:

```
ford
2000
```

Default argument value:

A function with an argument can be defined so that it automatically give some default value to the argument if none is passed to it.

Ex:

```
def disp(country="India"):
    print(country)
disp("France")
disp("America")
disp()
disp("Japan")
disp()
```

output:

```
France
America
India
Japan
India
```

Passing a List/tuple/dictionary as an argument:

We can pass list or tuple or dictionary as an argument to the function.

Ex:

```
def disp(p):
   for x in p:
      print(x)
t1 = (25,30,35,40)
disp(t1)
l1 = [6,23,"hello"]
disp(l1)
```

output:

```
25
30
35
40
6
23
hello
```

Return and Yield:

- Return statement is used to return the result to the calling function.
- Return statement is used only inside a function.
- Return statement includes the return keyword and the value (expression) that will be returned after that.
- The syntax is given below

```
def  function_name(parameters):
      statement(s)
      return (expression)
```

```
def add(a,b):
   return(a+b)
print(add(25,15))
```

output:

```
40
```

Return with multiple values:

A function can return multiple values in the form of list or tuple or dictionary.

Ex1:

```
def disp():
    name="ravi"
    age=50
    return (name,age)
t1=disp()
print(t1)
```

output:

```
('ravi', 50)
```

Ex2:

```
def disp():
    name="ravi"
    age=50
    return [name,age]
l1=disp()
print(l1)
```

output:

```
['ravi', 50]
```

Yield:

- Python yield returns a generator object.
- The syntax is given below

```
def function_name(parameters):
    yield (expression)
```

Ex:

```
def disp():
    yield "computer science and engineering"
t=disp()
print(t)
for x in t:
    print(x)
```

output:

```
<generator object disp at 0x037688F0>
```

computer science and engineering

Ex:

```
def disp():
  yield 4
  yield 6
  yield 8
for x in disp():
   print(x)
```

output:

```
4
6
8
```

Ex:

```
def disp(n):
  yield n
  yield n*n
  yield n*n*n
for x in disp(5):
   print(x)
```

output:

```
5
25
125
```

Scope of Variables:

- The variable declared in one part of the program may not be accessible to other parts.
- The scope of a variable is the region of code in which a defined variable is accessible.
- There are 2 types of variables.
  1. Global Variable
  2. Local Variable

Global Variable:

- The variable which are declared outside the function definition are called global variable.
- The global variables can be accessed from anywhere and in any function.

Ex:

```
a=10
def disp():
   print(a)
disp()
print(a)
```

output:

```
10
10
```

Local Variable:

- The variable which are declared inside the function are called local variable.
- The local variables can be accessed only inside the function where it is defined.

Ex:

```
def disp():
   a=10
   print(a)
disp()
#print(a) #NameError: name 'a' is not defined
```

output:

```
10
```

Ex2:

```
a=10
def disp():
   a=20
   print(a)
disp()
print(a)
```

output:

```
20
10
```

Recursion:

- A function calls itself is called function recursion.

- A defined function can call itself.
- A proper termination condition should be present in recursive functions otherwise it will into infinite loop.
- Advantages of Recursion:
  1. Can quickly solve complex functions.
  2. Length of the code will be reduced.
  3. Readability of the program will be increased.

Ex:

```
def fact(n):
   return n*fact(n-1)  if n>1 else 1
print(fact(5))
```

output:

```
120
```

Ex:

```
def fib(n):
   return fib(n-1)+fib(n-2)  if n>1 else n
print(fib(10))
```

output:

```
55
```

Anonymous Functions:

- A python lambda function is known as anonymous function that is defined without a name.
- A lambda function is a short hand way of defining a function.
- A lambda function can take any number of arguments but it can have only one expression.
- The syntax of defining a lambda function is given below

lambda arguments : expression

Ex1:

```
x=lambda a:a+10
print(x(20))
```

output:

```
30
```

Ex2:

```
x=lambda a,b:(a+b)
print(x(5,10))
```

output:

15

Ex3:

```
def cube(x):
    return(x*x*x)
lc=lambda x:x*x*x
print(cube(3))
print(lc(4))
```

output:

27

64

Week -10

Modules and Packages:

Why modules? Module creation;Importing modules; Module Namespace;

Packages: basics; path setting;Package__init__.py Files;

Commonly used modules: Math, random; Emoji;

Module:

- Modules in Python are reusable libraries of code having .py extension, which implements a group of functions and statements.
- Python comes with many built-in modules as part of the standard library.
- A module is like a code library that can be imported and used in a program.
- The module can contain functions, but also variables of all types (arrays, dictionaries, lists etc)

Why modules?

- We use Modules to break down larger programs into smaller manageable programs.
- Modules provide reusability of code.
- Grouping related code into a module makes the code easier to understand and use.
- Maintainability of the function will be increased.

Module creation:

To create a module, we just need to save the required code in a file with the file extension **.py**

Ex: Add the below two functions in file and save it as **mymodule.py**

```
def add(a,b):
    print("the sum=",a+b)
def product(a,b):
    print("the product=",a*b)
```

Ex:

Importing modules:

- To use a module in your program, import the module using import statement.
- All the import statements are placed at the beginning of the program.
- The syntax for import statement is

  import module_name

Ex:

```
import mymodule
mymodule.add(5,10)
mymodule.product(4,5)
```

output:

```
the sum= 15
the product= 20
```

Ex2:

people.py

```
person1 = { 'name' : 'Ravi', 'age' : 25, 'state' : 'Tamil Nadu' }
person2 = { 'name' : 'Ganesh', 'age' : 29, 'state' : 'Karnataka' }
```

peopletest.py

```
import people
print(people.person1['name'])
print(people.person2['state'])
```

output:

```
Ravi
Karnataka
```

- Modules can also be imported with a different name as shown below.

Ex:

```
import people as ps
print(ps.person1['age'])
```

output:

```
25
```

- We can import only specific items from a module as follows.
- Do not use the module name while using **from for import** statement.

```
from mymodule import add
add(10,20)
```

output:

```
the sum= 30
```

Packages:

A python package is a collection of several modules. Physically, a package is a folder containing modules and maybe other folders that themselves may contain more folders and modules.

Conceptually, it's a namespace. This simply means that a package's modules are bound together by a package name, by which they may be referenced. A package folder usually contains one file named __init_.py that basically tells python that the folder is a package. The init file may be empty, or it may contain code to be executed upon package initialization.

Commonly used modules:

Math:

This module contains commonly used mathematical functions like:

- The math.sqrt() method returns the square root of a number:
- The math.ceil() method rounds a number upwards to its nearest integer.
- The math.floor() method rounds a number downwards to its nearest integer.
- The math.fabs() method returns the absolute value of a number.
- The math.pow() method returns the value of x to the power of y.
- The math.exp() method returns e raised to the power of x.
- The math.factorial() method returns the factorial of a number
- The math.gcd() method returns the greatest common divisor of two integers
- Logarithm functions (math.log(),math.log2(),math.log10(), etc.)
- Trigonometric functions (math.sin(), math.cos(), math.tan(), etc.)
- The math.pi constant, returns the value of PI (3.14...)
- And more (math.exp(), math.log(), math.factorial(), etc.)

Ex:

```
import math
print("The value of 3**4 is:",math.pow(2,4))
print("The value of e power -3 is:",math.exp(-3))
print("The value of log 2 with base 3:",math.log(2,3))
print("The value of log 2 of 16 is:",math.log2(16))
print("The value of log 10 of 10000:",math.log10(10000))
print("The square root of 25 is :",math.sqrt(25))
print("The absolute value of -9 is :",math.fabs(-9))
```

```
print("The floor value of 3.65 is :",math.floor(3.65))
print("The ceil value of 3.65 is :",math.ceil(3.65))
print("The factorial of 5 is :",math.factorial(5))
print("The gcd of 5 and 15 is :",math.gcd(5,15))
print("The value of pi is :",math.pi)
print("The value of 2 in degrees :",math.degrees(2))
print("The value of 60 in radians :",math.radians(60))
print("The sine value of 2 is:",math.sin(2))
print("The cosine value of 3 is:",math.cos(3))
print("The largest number is:",max(54, 145, 67))
print("The smallest number is:",min(357, 190, 25))
```

output:

```
The value of 3**4 is: 16.0
The value of e power -3 is: 0.049787068367863944
The value of log 2 with base 3: 0.6309297535714574
The value of log 2 of 16 is: 4.0
The value of log 10 of 10000: 4.0
The square root of 25 is : 5.0
The absolute value of -9 is : 9.0
The floor value of 3.65 is : 3
The ceil value of 3.65 is : 4
The factorial of 5 is : 120
The gcd of 5 and 15 is : 5
The value of pi is : 3.141592653589793
The value of 2 in degrees : 114.59155902616465
The value of 60 in radians : 1.0471975511965976
The sine value of 2 is: 0.9092974268256817
The cosine value of 3 is: -0.9899924966004454
The largest number is: 145
The smallest number is: 25
```

Random:

- This module is used to generate pseudo random numbers.
- Some of its methods are:
- Seed(): Initialize the random number generator.
- random() - Generates random number in the interval [0, 1]
- randint(a, b) - Generates a random integer N such that a <= N <= b
- choice(seq) - Returns a random element from a non-empty sequence

Ex:

```
import random
random.seed(10)
print(random.random())
print("The random number between 1 and 10 is:",random.randint(1, 10))
print("The random number between 5 and 25 is:",random.randint(5, 25))
print("The random number between -10 and -2 is:",random.randint(-10, -2))
s1="koppal"
print("The random value from string is:",random.choice(s1))
t1 = (44,55,33,22,11)
print("The random value from tuple is:",random.choice(t1))
l1 = ["apple", "banana", "cherry"]
print("The random value from list is:",random.choice(l1))
random.shuffle(l1)
print("The list after first shuffle:",l1)
random.shuffle(l1)
print("The list after second shuffle:",l1)
```

output:

```
0.5714025946899135
The random number between 1 and 10 is: 7
The random number between 5 and 25 is: 20
The random number between -10 and -2 is: -10
The random value from string is: o
The random value from tuple is: 22
The random value from list is: banana
```

The list after first shuffle: ['cherry', 'apple', 'banana']

The list after second shuffle: ['banana', 'apple', 'cherry']

Emoji:

- Emojis are very small digital images used to express an idea or emotion of the sender.
- Emojis are very helpful in saving time and making the sender elaborate their mood easily.
- Emojis are also very helpful for receivers as, with the help of emojis, they can easily understand with what mood the sender has sent the message.

```
import emoji
print(emoji.emojize(":thumbs_up:"))
print(emoji.emojize(":grinning_face_with_big_eyes:"))
print(emoji.emojize(":winking_face_with_tongue:"))
print(emoji.emojize(":zipper-mouth_face:"))
print(emoji.emojize(":grinning_face:"))
print(emoji.emojize(":upside-down_face:"))
print(emoji.emojize(":zany_face:"))
print(emoji.emojize(":shushing_face:"))
```

output:

Week -11

NumPy Brief about NumPy module; NumPy arithmetic functions; NumPy array manipulation functions; NumPy statistical functions; Pandas Introduction, series, data frame; Create dataframes; formatting data; fundamental data frame operations;

NumPy:

- NumPy is a Python library used for working with arrays.
- It also has functions for working in domain of linear algebra, Fourier transform, and matrices.
- NumPy was created in 2005 by Travis Oliphant.
- It is open-source and can be used freely.
- NumPy stands for Numerical Python.

Why Use NumPy:

- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to **50x** faster than traditional Python lists.
- The array object in NumPy is called **ndarray**.
- It provides a lot of supporting functions that make working with ndarray very easy.
- Arrays are very frequently used in data science, where speed and resources are very important.

Installing NumPy:

- NumPy can be installed using the python package manger pip via command-line.
- The command for installing NumPy is:

```
pip install numpy
```

Numpy Arrays:

We can create a NumPy ndarray object by using the array() function.

Ex:

```
import numpy as np
arr = np.array([10, 23, 34, 46, 58])
print(arr)
print(type(arr))
```

output:

```
[10 23 34 46 58]
<class 'numpy.ndarray'>
```

NumPy arithmetic functions:

```
import numpy as np
a = np.array([44,78,23,45,67,12,34,56])
print(a)
print("Add 5 to array elements:",np.add(a,5))
print("Minimum element:",np.min(a))
print("Maximum element:",np.max(a))
print("Average:",np.average(a))
print("Mean:",np.mean(a))
print("Median:",np.median(a))
print("Sorted array:",np.sort(a))
print("Sum of array elements:",np.sum(a))
print("Product of array elements:",np.prod(a))
print("Difference of array elements:",np.diff(a))
print("Ceil:",np.ceil(56.234))
print("Floor:",np.floor(56.234))
```

output:

```
 [44 78 23 45 67 12 34 56]
Add 5 to array elements: [49 83 28 50 72 17 39 61]
Minimum element: 12
Maximum element: 78
Average: 44.875
Mean: 44.875
Median: 44.5
Sorted array: [12 23 34 44 45 56 67 78]
Sum of array elements: 359
Product of array elements: 213533184
Difference of array elements: [ 34 -55  22  22 -55  22  22]
Ceil: 57.0
Floor: 56.0
```

NumPy Array Manipulation Functions:

add(): This function is used to output the addition of two arrays.

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.add(a,b)
print(c)
print(a + b)
```

output:

```
[5 7 9]
[5 7 9]
```

subtract(): This function is used to output the subtraction of two arrays.

```
import numpy as np
a = np.array([10, 20, 30])
b = np.array([4, 5, 6])
c = np.subtract(a,b)
print(c)
print(a - b)
```

output:

```
[ 6 15 24]
[ 6 15 24]
```

multiply(): This function is used to output the product of two arrays.

```
import numpy as np
a = np.array([12, 6, 6])
b = np.array([4, 5, 6])
c = np.multiply(a,b)
print(c)
print(a * b)
```

output:

```
[48 30 36]
[48 30 36]
```

divide(): This function is used to output the division of two arrays.

```
import numpy as np
a = np.array([63, 16, 26])
b = np.array([4, 5, 6])
c = np.divide(a,b)
print(c)
print(a / b)
```

output:

```
[15.75      3.2       4.33333333]
[15.75      3.2       4.33333333]
```

mod() and remainder(): These functions are used to output the remainder of two arrays.

```
import numpy as np
a = np.array([63, 16, 26])
b = np.array([4, 5, 6])
c = np.remainder(a,b)
d = np.mod(a,b)
print(c)
print(d)
```

output:

```
[3 1 2]
[3 1 2]
```

NumPy Statistical Functions:

- The **numpy.amin()** and **numpy.amax()** functions are used to find the minimum and maximum of the array elements along the specified axis respectively.
- The function **numpy.median()** is used to calculate the median of the multi-dimensional or one-dimensional arrays.
- The function **numpy.mean()** is used to calculate the mean of the multi-dimensional or one-dimensional arrays.
- The **numpy.average()** function is used to find the weighted average along the axis of the multi-dimensional arrays where their weights are given in another array.

```
import numpy as np
a = np.array([[2,10,20],[80,43,31],[22,43,10]])
print("The minimum element among the array:",np.amin(a))
```

```
print("The maximum element among the array:",np.amax(a))
print("Percentile along axis 0",np.percentile(a, 10, 0))
print("Percentile along axis 1",np.percentile(a, 10, 1))
print("Median of array:",np.median(a))
print("Mean of array:",np.mean(a))
print("Average of array :",np.average(a))
```

output:

```
The minimum element among the array: 2
The maximum element among the array: 80
Percentile along axis 0 [ 6.  16.6 12. ]
Percentile along axis 1 [ 3.6 33.4 12.4]
Median of array: 22.0
Mean of array: 29.0
Average of array : 29.0
```

Pandas:

- Pandas is a Python library used for working with data sets.
- It has functions for analyzing, cleaning, exploring, and manipulating data.
- The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Uses of Pandas:

- Pandas allow us to analyze big data and make conclusions based on statistical theories.
- Pandas can clean messy data sets, and make them readable and relevant.
- Relevant data is very important in data science.

Installing Pandas:

- Pandas can be installed using the python package manger pip via command-line.
- The command for installing Pandas is:

```
pip install pandas
```

Series:

- A Pandas Series is like a column in a table.
- It is a one-dimensional array holding data of any type.
- It can be created from a list as below

```
import pandas as pd
```

```
a = [1, 7, 2]
srs = pd.Series(a)
print(srs)
```

output:

```
0  1
1  7
2  2
dtype: int64
```

DataFrames:

- A Pandas DataFrame is a two-dimensional data structure, like a two-dimensional array, or a table with rows and columns.
- It can be created from a dictionary as below

```
import pandas as pd
data = {
 "calories": [420, 380, 390],
 "duration": [50, 40, 45]
}
df = pd.DataFrame(data)
print(df)
```

output:

```
   calories  duration
0       420        50
1       380        40
2       390        45
```

- DataFrames can also be created from files (and usually is) as shown below.

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df)
```

The above code will load data from a Comma Separated file (CSV file) into a DataFrame and display it.

Week -12

Files Concept; features; file operations; Opening Files; Closing Files; Writing to Files; Reading to Files; File methods; Working with files using data frame.

Files:

- Files are named locations on disk to store related information.
- These are used to permanently store data in a non-volatile memory (e.g. hard disk).
- File handling in simple it means handling of files such as opening the file, reading, writing, and many other operations.
- A file operation takes place in the following order:
  - ✓ Open a file
  - ✓ Read or write
  - ✓ Close the file

Features of Files:

- A file always has a name.
- A file always takes up storage space.
- A file is always saved in a certain format (.txt, .pdf, .ppt, .doc, .jpeg, etc).
- A file contains information on when it was created and when it was last modified.
- Files usually have access rights.

File Operations:

Opening files:

- A file can be opened using the inbuilt function **open**() which returns a file object.
- The open() function takes two parameters; *filename*, and *mode*.
- The syntax is given below

fileobject = open(*filename, mode)*

Ex:

```
f = open(("demofile.txt", "r")
```

- It is necessary that demofile.txt is in the same directory as the python program.
- The mode in which the file is opened is specified as an additional argument while opening the file.
- If no additional argument is present, then the file is opened in read mode.
- A file can be opened in various modes:
  - ✓ '**r**' —> Read - Default value. Opens a file for reading, error if the file does not exist.
  - ✓ '**a**' —> Append - Opens a file for appending, creates the file if it does not exist.

- ✓ '**w**' —> Write - Opens a file for writing, creates the file if it does not exist.
- ✓ '**x**' —> Create - Creates the specified file, returns an error if the file exists.

➢ We can specify if the file should be handled in text or binary mode:

- ✓ '**t**' —> Text - Default value. Text mode.
- ✓ '**b**' —> Binary - Binary mode (e.g. for images).

➢ In below example where a file is opened in write mode.

```
f = open('demofile.txt', 'w')
```

➢ In below example where a file is opened in write mode and handled as binary:

```
f = open('demofile.txt', 'wb')
```

Reading Files:

➢ The **read**() method is used for reading the content of the file.

Ex:

```
f = open("demofile.txt", "r")
print(f.read())
```

➢ By default the **read**() method returns the whole text, but we can also specify how many characters we want to read.

Ex:

```
f = open("demofile.txt", "r")
print(f.read(10))
```

Returns the 10 first characters of the file:

readline():

We can also read files line by line by using the built in method **readline**().

Ex:

```
f = open("demofile.txt", "r")
print(f.readline())
```

The above code will return the first line in demofile.txt

readlines():

We can also create a list of all the lines in an opened file by calling **readlines**().

Ex:

```
f = open("demofile.txt", "r")
l = f.readlines()
print(type(l))
```

```
print(l[1])
```

Writing to Files:

- Data can be written to files by using the argument 'a' or 'w' in the **open**() function.
  - ✓ 'a' will append at the end of a file if it already exists.
  - ✓ 'w' will overwrite the file if it already exists.
- Both 'a' and 'w' will create a new file if it does not exist.
- After a file is opened in one of the above write modes, the built-in method **write**() is used to write data to the file.

Ex1:

```
f = open("demofile.txt", "a")
f.write(" Hello India")
```

The above code will add the text Hello India at the end of the file demofile.txt

Ex2:

```
f = open("demofile.txt", "w")
f.write("Hello India")
```

The above code will overwrite demofile.txt so that the text Hello India is the only data on it.

Closing Files:

- After working with a file, it is a good programming habit to close it.
- In some cases, the changes made to the file may not show until the file is closed.
- Files are closed using the built-in method **close**().

Ex:

```
f = open("demofile2.txt")
print(f.read())
f.close()
```

File Methods:

1. **read():**This method is used for reading the content of the file.
2. **readline():**This method reads one entire line from the file.
3. **readlines():** This method reads entire file and return a list containing the lines.
4. **write():**This method writes a string to the file.
5. **seek**(): We can change our current file cursor (position) using the **seek**() method.
6. **tell**(): This method returns the current file cursor position (in number of bytes).
7. **readable**(): This method returns **True** if the file is readable otherwise it will return **False**.

8. **writable**(): This method returns **True** if the file is writable otherwise it will return **False**.

Working with files using data frames:

- A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

Ex:

```
import pandas as pd
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
```

output:

```
df = pd.DataFrame(data)
print(df)
    calories  duration
0      420       50
1      380       40
2      390       45
```

- A simple way to store big data sets is to use CSV files (comma separated files).
- A CSV file contains plain text and is a well know format that can be read by everyone including Pandas.
- The columns()method returns the column labels of the DataFrame.
- The head() method returns the headers and a specified number of rows, starting from the top.
- The tail() method returns the headers and a specified number of rows, starting from the bottom.
- The info() method gives more information about the DataFrame.

```
import pandas as pd
data = {
 'CHN': {'COUNTRY': 'China', 'POP': 1398.72, 'AREA': 9596.96,
 'GDP': 12234.78, 'CONT': 'Asia'},
 'IND': {'COUNTRY': 'India', 'POP': 1351.16, 'AREA': 3287.26,
 'GDP': 2575.67, 'CONT': 'Asia', 'IND_DAY': '1947-08-15'},
 'USA': {'COUNTRY': 'US', 'POP': 329.74, 'AREA': 9833.52,
 'GDP': 19485.39, 'CONT': 'N.America',
```

```
 'IND_DAY': '1776-07-04'}}
columns = ('COUNTRY', 'POP', 'AREA', 'GDP', 'CONT', 'IND_DAY')
df = pd.DataFrame(data=data, index=columns).T
df.to_csv('data.csv')
df = pd.read_csv('data.csv', index_col=0)
print(df)
```

output:

```
  COUNTRY      POP     AREA      GDP      CONT     IND_DAY
CHN  China  1398.72  9596.96  12234.78       Asia        NaN
IND   India  1351.16  3287.26   2575.67       Asia  1947-08-15
USA      US  329.74  9833.52  19485.39  N.America  1776-07-04
```

Week-13

Error and Exception Handling: Python errors; exceptions: built in, user defined. How to catch exceptions? Raising exceptions;

Errors:

- Errors are the problems in a program due to which the program will stop the execution.
- On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.
- Two types of Error occur in python.
  1. Syntax errors
  2. Logical errors (Exceptions)

Syntax errors:

- Syntax errors, also known as parsing errors and when the proper syntax of the language is not followed then a syntax error will occur.

Logical errors (Exceptions):

- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
- Errors detected during execution are called exceptions.

Error Handling

- Like many other programming languages, it possible to program how an error/exception that is raised during runtime must be handled.
- It is also possible to throw custom exceptions in a program.

HANDLING EXCEPTIONS

- Exceptions are handled using the try, except, finally keywords.
- The try block lets you test a block of code for errors.
- The except block lets you handle the error.
- The finally block lets you execute code regardless of the result of the try and except blocks.
- During runtime, when an exception occurs, python will normally stop execution and generate an error message. However if the line triggering the exception is in a try block, then instead of stopping and generating an error message as usual, the code in the except block is executed.

Ex:

```
try:
    print(x)
```

```
except:
    print('This is a custom error message')
```

- In the above code the variable x is not defined. Hence an exception will be triggered at runtime.
- But this exception will only cause the code in except block to be executed. Hence the output of the above code will be:

This is a custom error message

Common Exceptions

Exceptions triggered at runtime can be various types. A few common types are:

- **NameError**: It is raised when a local or global name is not found.

Ex:

```
try:
  a=x+20
except NameError:
   print("Name Error")
```

- **TypeError**: It is raised when an operation or function is applied to an object of inappropriate type.

Ex:

```
try:
  a = "a" + 5
except TypeError:
   print("Type Error")
```

- **IndexError**: It is raised when a sequence subscript is out of range.

Ex:

```
try:
   l=[10,20,30]
   print(l[5])
except IndexError:
   print("Index Error")
```

- **ZeroDivisionError** – It is raised when the second argument of a division or modulo operation is zero.

Ex:

```
try:
   x = 5/0
except ZeroDivisionError:
   print("Zero Division Error")
```

- **StopIteration** – It is raised by built-in function next() to signal that there are no further items produced by the iterator.

Ex:

```
try:
    x = (i for i in [1, 2])
    a = next(x) # a will be 1
    b = next(x) # b will be 2
    c = next(x)
except          StopIteration:
   print("Stop Iteration Error")
```

Catching Specific Exceptions

- Exception blocks (or except blocks) can be made to trigger only for particular types of exceptions.
- There can also be multiple exception blocks for a try block with exception block made to trigger for a different type of exception. Below is an example with two exception blocks.

Ex:

```
try:
   print(x)
except NameError:
   # This block is executed if the exception raised is of type NameError
   print('Variable not defined')
except:
   # This is the default except block
   # This block is executed if an exception is raised but not caught above
   # The default except block must always be last
   print('Some other error occurred')
```

Output:

Variable not defined

Else Block:

- An else block can added be after try … except to execute a block of code if no error was triggered in the try block.

Ex:

```
try:
```

```
    print('Hello')
except:
    print('Something went wrong')
else:
    print('Nothing went wrong')
```

Finally Block:

- This block is executed regardless of whether or not an exception is triggered in the try block.

Ex:

```
try:
    print(x)
except:
    print('Something went wrong')
finally:
    print('Finished error handling')
```

RAISING EXCEPTIONS:

- It is possible to raise custom errors for some particular conditions in a program. The raise keyword is used to do this.

Ex:

```
x = -1
if x < 0:
    raise Exception('Only positive numbers are allowed')
```

- We could also raise custom errors of a particular type.

Ex:

```
x = 'hello'
if type(x) != int:
    raise TypeError('Only integers allowed')
```