Question Bank and Answers on Introduction to Python Programming Course(22PLC15B/25B)

1. Explain the salient features of Python

Python is a high-level, general-purpose programming language that is widely used for a variety of tasks, such as web development, data analysis, machine learning, and artificial intelligence. Python is a dynamic, high-level, free open source, and interpreted programming language. It supports object-oriented programming as well as procedural-oriented programming. In Python, we don't need to declare the type of variable because it is a dynamically typed language. For example, x = 10 Here, x can be anything such as String, int, etc.

1. Free and Open Source

Python language is freely available at the official website and you can download it from the given download link below click on the Download Python keyword. Download Python Since it is open-source, this means that source code is also available to the public. So you can download it, use it as well as share it.

2. Easy to code

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

3. Object-Oriented Language

One of the key features of Python is Object-Oriented programming. Python supports object-oriented language and concepts of classes, object encapsulation, etc.

4. GUI Programming Support

Graphical User interfaces can be made using a module such as PyQt5, PyQt4, wxPython, or Tk in python. PyQt5 is the most popular option for creating graphical apps with Python.

5. Extensible feature

Python is an Extensible language. We can write some Python code into C or C++ language and also we can compile that code in C/C++ language.

6. Python is a Portable language

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

7. Python is an Integrated language

Python is also an Integrated language because we can easily integrate Python with other languages like C, C++, etc.

8. Interpreted Language:

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

9. Large Standard Library

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

10. Dynamically Typed Language

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

2. Explain different types of operators in Python with examples.

Operators are symbols, such as +, –, =, >, and <, that perform certain mathematical or logical operation to manipulate data values and produce a result based on some rules. An operator manipulates the data values called operands.

Python language supports a wide range of operators. They are

1. Arithmetic Operators
2. Assignment Operators
3. Comparison Operators
4. Logical Operators
5. Bitwise Operators
6. Identity Operators
7. Membership Operators

1. Arithmetic Operators

Arithmetic operators are used to execute arithmetic operations such as addition, subtraction, division, multiplication etc. The following TABLE 2.2 shows all the arithmetic operators.

List of Arithmetic Operators

Operator	Operator Name	Description	Example
+	Addition operator	Adds two operands, producing their sum.	p + q = 5
–	Subtraction operator	Subtracts the two operands, producing their difference.	p – q = –1
*	Multiplication operator	Produces the product of the operands.	p * q = 6
/	Division operator	Produces the quotient of its operands where the left operand is the dividend and the right operand is the divisor.	q / p = 1.5
%	Modulus operator	Divides left hand operand by right hand operand and returns a remainder.	q % p = 1
**	Exponent operator	Performs exponential (power) calculation on operators.	p**q = 8
//	Floor division operator	Returns the integral part of the quotient.	9//2 = 4 and 9.0//2.0 = 4.0

Note: The value of p is 2 and q is 3.

2. Assignment Operators

Assignment operators are used for assigning the values generated after evaluating the right operand to the left operand. Assignment operation always works from right to left. Assignment operators are either simple assignment operator or compound assignment operators. Simple assignment is done with the equal sign (=) and simply assigns the value of its right operand to the variable on the left.

For example,

```
x = 5
x = x + 1
```

Compound assignment operators support shorthand notation for avoiding the repetition of the left-side variable on the right side. Compound assignment operators combine assignment operator with another operator with = being placed at the end of the original operator.

For example, the statement

x = x + 1

can be written in a compactly form as shown below.

x += 1

List of Assignment Operators

Operator	Operator Name	Description	Example
=	Assignment	Assigns values from right side operands to left side operand.	z = p + q assigns value of p + q to z
+=	Addition Assignment	Adds the value of right operand to the left operand and assigns the result to left operand.	z += p is equivalent to z = z + p
−=	Subtraction Assignment	Subtracts the value of right operand from the left operand and assigns the result to left operand.	z −= p is equivalent to z = z − p
*=	Multiplication Assignment	Multiplies the value of right operand with the left operand and assigns the result to left operand.	z *= p is equivalent to z = z * p
/=	Division Assignment	Divides the value of right operand with the left operand and assigns the result to left operand.	z /= p is equivalent to z = z / p
**=	Exponentiation Assignment	Evaluates to the result of raising the first operand to the power of the second operand.	z**= p is equivalent to z = z ** p
//=	Floor Division Assignment	Produces the integral part of the quotient of its operands where the left operand is the dividend and the right operand is the divisor.	z //= p is equivalent to z = z // p
%=	Remainder Assignment	Computes the remainder after division and assigns the value to the left operand.	z %= p is equivalent to z = z % p

3. **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. The operands can be Numbers or Strings or Boolean values. Strings are compared letter by letter using their ASCII values, thus, “P” is less than “Q”, and “Aston” is greater than “Asher”.

List of Comparison Operators

Operator	Operator Name	Description	Example
==	Equal to	If the values of two operands are equal, then the condition becomes True.	(p == q) is not True.
!=	Not Equal to	If values of two operands are not equal, then the condition becomes True.	(p != q) is True
>	Greater than	If the value of left operand is greater than the value of right operand, then condition becomes True.	(p > q) is not True.
<	Lesser than	If the value of left operand is less than the value of right operand, then condition becomes True.	(p < q) is True.
>=	Greater than or equal to	If the value of left operand is greater than or equal to the value of right operand, then condition becomes True.	(p >= q) is not True.
<=	Lesser than or equal to	If the value of left operand is less than or equal to the value of right operand, then condition becomes True.	(p <= q) is True.

Note: The value of p is 10 and q is 20.

4. **Logical Operators**

The logical operators are used for comparing or negating the logical values of their operands and to return the resulting logical value. The values of the operands on which the logical operators operate evaluate to either True or False. The result of the logical operator is always a Boolean value, True or False.

List of Logical Operators

Operator	Operator Name	Description	Example
and	Logical AND	Performs AND operation and the result is True when both operands are True	p and q results in False
or	Logical OR	Performs OR operation and the result is True when any one of both operand is True	p or q results in True
not	Logical NOT	Reverses the operand state	not p results in False

Note: The Boolean value of p is True and q is False.

Boolean Logic Truth Table

P	Q	P and Q	P or Q	Not P
True	True	True	True	False
True	False	False	True	
False	True	False	True	True
False	False	False	False	

5. Bitwise Operators

Bitwise operators treat their operands as a sequence of bits (zeroes and ones) and perform bit by bit operation. For example, the decimal number ten has a binary representation of 1010. Bitwise operators perform their operations on such binary representations, but they return standard Python numerical values.

List of Bitwise Operators

Operator	Operator Name	Description	Example
&	Binary AND	Result is one in each bit position for which the corresponding bits of both operands are 1s.	p & q = 12 (means 0000 1100)
|	Binary OR	Result is one in each bit position for which the corresponding bits of either or both operands are 1s.	p | q = 61 (means 0011 1101)
^	Binary XOR	Result is one in each bit position for which the corresponding bits of either but not both operands are 1s.	(p ^ q) = 49 (means 0011 0001)
~	Binary Ones Complement	Inverts the bits of its operand.	(~p) = −61 (means 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	p << 2 = 240 (means 1111 0000)
>>	Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	p >> 2 = 15 (means 0000 1111)

Note: The value of p is 60 and q is 13.

Bitwise and (&)	Bitwise or (|)
a = 0011 1100 → (60) b = 0000 1101 → (13) a & b = 0000 1100 → (12)	a = 0011 1100 → (60) b = 0000 1101 → (13) a | b = 0011 1101 → (61)
Bitwise exclusive or (^)	**One's Complement (~)**
a = 0011 1100 → (60) b = 0000 1101 → (13) a ^ b = 0011 0001 → (49)	a = 0011 1100 → (60) ~ a = 1100 0011 → (-61)
Binary left shift (<<)	**Binary right shift (>>)**
a = 0011 1100 → (60) a << 2 = 1111 0000 → (240) left shift of 2 bits	a = 0011 1100 → (60) a >> 2 = 0000 1111 → (15) right shift of 2 bits

6. **Identity Operators**

is and is not are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

is **True if the operands are identical**

is not **True if the operands are not identical**

Example:

a = 10

b = 20

```
c = a
print(a is not b)
print(a is c)
```

Output:

True
True

7. Membership Operators

in and not in are the membership operators; used to test whether a value or variable is in a sequence.

in True if value is found in the sequence
not in True if value is not found in the sequence

Example:

```
x = 24
y = 20
list = [10, 20, 30, 40, 50]
if (x not in list):
    print("x is NOT present in given list")
else:
    print("x is present in given list")
if (y in list):
    print("y is present in given list")
else:
    print("y is NOT present in given list")
Output
x is NOT present in given list
y is present in given list
```

3. Explain rules for defining a variable. How to assign different values to the variables

Variable is a named placeholder to hold any type of data which the program can use to assign and modify during the course of execution. Variables should be valid identifiers. An identifier is a name given to a variable, function, class or module. Identifiers may be one or more characters in the following format:

- *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 myCountry, other_1 and good_morning, all are valid examples. A Python identifier can begin with an alphabet (A – Z and a – z and _).*
- *An identifier cannot start with a digit but is allowed everywhere else. 1plus is invalid, but plus1 is perfectly fine.*
- *Keywords cannot be used as identifiers. Keywords are a list of reserved words that have predefined meaning. Keywords are special vocabulary and cannot be used by programmers as identifiers for variables, functions, constants or with any identifier name.*
- *Spaces and special symbols like !, @, #, $, % etc. as identifiers.*
- *Identifier can be of any length.*

4. Briefly explain binary left shift and binary right shift operators with examples.

Shift Operators

These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively. They can be used when we have to multiply or divide a number by two.

Bitwise right shift: *Shifts the bits of the number to the right and fills 0 on voids left(fills 1 in the case of a negative number) as a result. Similar effect as of dividing the number with some power of two.*

Example 1:

a = 10 = 0000 1010 (Binary)

a >> 1 = 0000 0101 = 5

Example 2:

a = -10 = 1111 0110 (Binary)

a >> 1 = 1111 1011 = -5

Bitwise left shift: *Shifts the bits of the number to the left and fills 0 on voids right as a result. Similar effect as of multiplying the number with some power of two.*

Example 1:

a = 5 = 0000 0101 (Binary)

a << 1 = 0000 1010 = 10

a << 2 = 0001 0100 = 20

Example 2:

b = -10 = 1111 0110 (Binary)

b << 1 = 1110 1100 = -20

b << 2 = 1101 1000 = -40

5. Explain precedence and associatively of operators with examples.

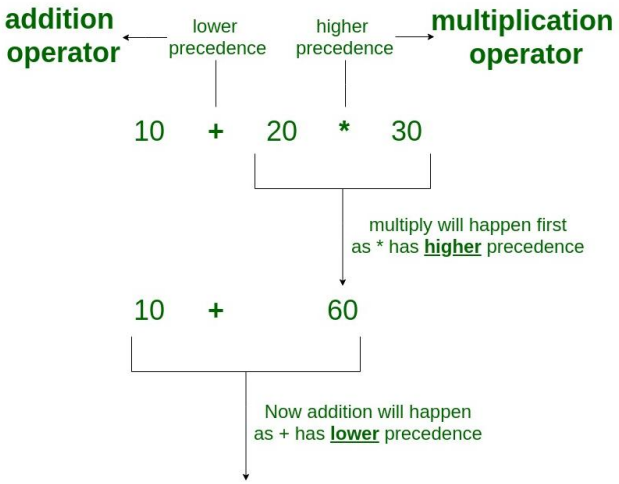
Operator precedence determines the way in which operators are parsed with respect to each other. Operators with higher precedence become the operands of operators with lower precedence. Associativity determines the way in which operators of the same precedence are parsed.

Operator Precedence in Python

Operators	Meaning
()	Parentheses
**	Exponent
+x, −x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, −	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

Example:

In the expression: 10 + 20 * 30



Operator Associativity: *If an expression contains two or more operators with the same precedence then Operator Associativity is used to determine. It can either be Left to Right or from Right to Left.*

Example: '*' and '/' have the same precedence and their associativity is Left to Right, so the expression "100 / 10 * 10" is treated as "(100 / 10) * 10".

Operator	Description	Associativity
( )	Parentheses	left-to-right
**	Exponent	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ –	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
is, is not in, not in	Identity Membership operators	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
not	Logical NOT	right-to-left
and	Logical AND	left-to-right
or	Logical OR	left-to-right
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left

6. Outline different assignment operators with examples.

Assignment operators are used for assigning the values generated after evaluating the right operand to the left operand. Assignment operation always works from right to left. Assignment operators are either simple assignment operator or compound assignment operators. Simple assignment is done with the equal sign (=) and simply assigns the value of its right operand to the variable on the left. For example,

```
1. >>> x = 5
2. >>> x = x + 1
3. >>> x
6
```

Compound assignment operators support shorthand notation for avoiding the repetition of the left-side variable on the right side. Compound assignment operators combine assignment operator with another operator with = being placed at the end of the original operator.

For example, the statement

```
>>> x = x + 1
```

can be written in a compactly form as shown below.

```
>>> x += 1
```

Operator	Description	Syntax
=	Assign value of right side of expression to left side operand	x = y + z
+=	Add AND: Add right-side operand with left side operand and then assign to left operand	a+=b a=a+b
-=	Subtract AND: Subtract right operand from left operand and then assign to left operand	a-=b a=a-b
*=	Multiply AND: Multiply right operand with left operand and then assign to left operand	a*=b a=a*b
/=	Divide AND: Divide left operand with right operand and then assign to left operand	a/=b a=a/b
%=	Modulus AND: Takes modulus using left and right operands and assign the result to left operand	a%=b a=a%b
//=	Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand	a//=b a=a//b
**=	Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand	a**=b a=a**b
&=	Performs Bitwise AND on operands and assign value to left operand	a&=b a=a&b
|=	Performs Bitwise OR on operands and assign value to left operand	a|=b a=a|b
^=	Performs Bitwise xOR on operands and assign value to left operand	a^=b a=a^b
>>=	Performs Bitwise right shift on operands and assign value to left	a>>=b

	operand	a=a>>b
<<=	Performs Bitwise left shift on operands and assign value to left operand	a <<= b a= a << b

Examples of Assignment Operators

```
a = 10
 # Assign value
b = a
print(b)
 # Add and assign value
b += a
print(b)
 # Subtract and assign value
b -= a
print(b)
 # multiply and assign
b *= a
print(b)
 # bitwise lishift operator
b <<= a
print(b)
```

Output

```
10
20
10
100
102400
```

7. Briefly explain how to read data from the keyboard.

Python provides us with in-built function input() to read the input from the keyboard.

input (): *This function first takes the input from the user and converts it into a string. The type of the returned object always will be <type 'str'>. It does not evaluate the expression it just returns the complete statement as String. For example, Python provides a built-in function called input which takes the input from the user. When the input function is called it stops the program and waits for the user's input. When the user presses enter, the program resumes and returns what the user typed.*

Example 1:

```
val = input("Enter your value: ")
print(val)
```

Example 2:

```
name = input('What is your name?\n')
print(name)
```

Output:

```
What is your name?
Ram
Ram
```

- *When input() function executes program flow will be stopped until the user has given input.*

- *The text or message displayed on the output screen to ask a user to enter an input value is optional i.e. the prompt, which will be printed on the screen is optional.*
- *Whatever the text entered as input, the input function converts it into a string. if integer value entered, still input() function converts it into a string. To read an integer, it should be explicitly converted into an integer in the code using typecasting.*

8. Explain Type conversion in Python with examples.

Python defines type conversion functions to directly convert one data type to another which is useful in day-to-day and competitive programming. This article is aimed at providing information about certain conversion functions.

There are two types of Type Conversion in Python:

- *Implicit Type Conversion*
- Explicit Type Conversion

Implicit Type Conversion

In Implicit type conversion of data types in Python, the Python interpreter automatically converts one data type to another without any user involvement.

Example:

```
x = 10
print("x is of type:",type(x))
y = 10.6
print("y is of type:",type(y))
z = x + y
print(z)
print("z is of type:",type(z))
```

Output:

```
x is of type: <class 'int'>
y is of type: <class 'float'>
20.6
z is of type: <class 'float'>
```

As we can see the data type of 'z' got automatically changed to the "float" type while one variable x is of integer type while the other variable y is of float type. The reason for the float value not being converted into an integer instead is due to type promotion that allows performing operations by converting data into a wider-sized data type without any loss of information. This is a simple case of Implicit type conversion in python.

Explicit Type Conversion

In Explicit Type Conversion in Python, the data type is manually changed by the user as per their requirement. With explicit type conversion, there is a risk of data loss since we are forcing an expression to be changed in some specific data type. Various forms of explicit type conversion are explained below:

1. int(a, base): This function converts any data type to integer. 'Base' specifies the base in which string is if the data type is a string.

2. float(): This function is used to convert any data type to a floating-point number.

3. ord() : This function is used to convert a character to integer.

4. hex() : This function is to convert integer to hexadecimal string.
5. oct() : This function is to convert integer to octal string.
6. tuple() : This function is used to convert to a tuple.
7. set() : This function returns the type after converting to set.
8. list() : This function is used to convert any data type to a list type.
9. dict() : This function is used to convert a tuple of order (key,value) into a dictionary.
10. str() : Used to convert integer into a string.
11. complex(real,imag) : This function converts real numbers to complex(real,imag) number.
12. chr(number): This function converts number to its corresponding ASCII character.
Example:

```
s = "10010"
# printing string converting to int base 2
c = int(s,2)
print ("After converting to integer base 2 : ", end="")
print (c)
# printing string converting to float
e = float(s)
print ("After converting to float : ", end="")
print (e)
```

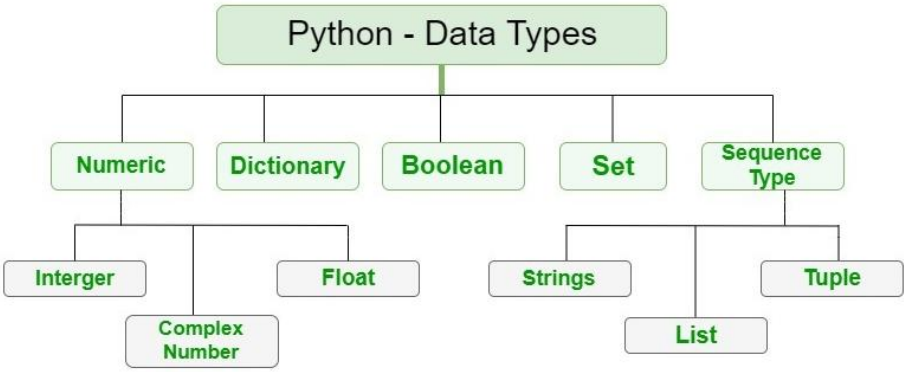
Output:

```
After converting to integer base 2 : 18
After converting to float : 10010.0
```

9. Write a short note on data types in Python.

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes. Following are the standard or built-in data type of Python:

- *Numeric*
- *Sequence Type*
- *Boolean*
- *Set*
- *Dictionary*



a. Numeric Data Type

In Python, numeric data type represent the data which has numeric value. Numeric value can be integer, floating number or even complex numbers. These values are defined as int, float and complex class in Python.

***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.*

***Float** – This value is represented by float class. It is a real number with floating point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.*

***Complex Numbers** – Complex number is represented by complex class. It is specified as (real part) + (imaginary part)j. For example – 2+3j*

b. In Python, sequence is the ordered collection of similar or different data types. Sequences allows to store multiple values in an organized and efficient fashion. There are several sequence types in Python –

- *String*
- *List*
- *Tuple*

c. Boolean data tupe

Data type with one of the two built-in values, True or False. Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false). But non-Boolean objects can be evaluated in Boolean context as well and determined to be true or false. It is denoted by the class bool.

d. Set

In Python, Set is an unordered collection of data type that is iterable, mutable and has no duplicate elements. The order of elements in a set is undefined though it may consist of various elements.

e. Dictionary

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds key:value pair. Key-value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon :, whereas each key is separated by a 'comma'.

10. Explain precedence and associatively of operators. Solve the expression 3/2*4+3+(10/4)3-3**

In any programming language, the order in which operations are performed can be important. The order in which operators are applied is determined by their precedence and associativity.

*Precedence refers to the order in which operations are performed when multiple operations are present in an expression. For example, multiplication is performed before addition. So, in the expression 2 + 3 * 4, the multiplication (3 * 4) is done first, and then the addition (2 + 12) is done.*

Associativity refers to the order in which operations with the same precedence are performed. For example, in the expression 2 - 3 - 4, the subtraction (3 - 4) is done first, and then the subtraction (2 - (-1)) is done.

In Python, the precedence of operators is as follows (highest to lowest):

Parentheses ()
*Exponentiation ***
Unary operator (e.g. +, -)
Multiplication, Division, and Modulus (, /, %)*
Addition and Subtraction (+, -)
Comparison (>, <, >=, <=, ==, !=)
Logical (and, or, not)
Assignment (=)

The associativity of operators is either left-to-right or right-to-left. For example, the assignment operator (=) is right-to-left associative, so in the statement a = b = c, the value of c is assigned to b, and then b is assigned to a.

In general, it's a good practice to use parentheses to make the order of operations clear in complex expressions.

It's also important to note that in python, the = operator is used for assignment and not for comparison, so in the expression a == b it compares the values of a and b and returns a boolean, whereas in the expression a = b it assigns the value of b to a.

Operator precedence determines the way in which operators are parsed with respect to each other. Operators with higher precedence become the operands of operators with lower precedence. Associativity determines the way in which operators of the same precedence are parsed. Almost all the operators have left-to-right associativity.

Evaluation of the expresson:

*3/2*4+3+(10/4)**3-3*

*= 3/2*4+3+(2.5)**3-3*

*= 3/2*4+3+2.5**3-3*

*= 3/2*4+3+ 15.625-3*

*= 1.5*4+3+ 15.625-3*

= 6.0+3+ 15.625-3

= 9.0+ 15.625-3

= 24.625-3

= 21.625

11. Write a program to read two integers and perform arithmetic operations on them (addition, subtraction, multiplication and division).

Program Code:

```
a= int(input("Enter First Number: "))
b= int(input("Enter Second Number: "))

op =input("Enter the operator: ")

if op=='+':
   result=a+b
elif op=='-':
   result = a-b
elif op=='*':
   result = a*b
elif op=='//':
   result = a//b
```

```
elif op=='/':
   result=a/b
elif op=='%':
   result = a %b
else:
   print("Invalid Operator")
   sys.exit()

print("{} {} {} = {}".format(a,op,b,result))
```

12. Write a program to read the marks of three subjects and find the average of them.

Program Code

```
m1, m2, m3= map(int, input("Enter Marks of 3 Subjects separared by space on a single line: ").split())

average = (m1+m2+m3)/3

print("Average Marks = {:.2f}".format(average))
```

13. Write a program to convert kilogram into pound.

Program Code

```
kg= int(input("Enter Weight in Kilograms : "))
pounds= 2.20462*kg

print("{} Kg = {:.2f} Pounds".format(kg, pounds))
```

14. Surface area of a prism can be calculated if the lengths of the three sides are known. Write a program that takes the sides as input (read it as integer) and prints the surface area of the prism (Surface Area = 2ab + 2bc + 2ca).

Program Code:

```
ab,bc, ca= map(int, input("Enter length of 3 sides on a single line : ").split())
surface_Area= 2*ab + 2*bc + 2*ca
print("Surface Area= {:.2f}".format(surface_Area))
```

15. A plane travels 395,000 meters in 9000 seconds. Write a program to find the speed of the plane (Speed = Distance / Time).

Program Code:

```
distance=39500
time = 9000
```

```
speed = distance/time

print("Speed of Plane = {:.2f} m/s".format(speed))
```

16. You need to empty out the rectangular swimming pool which is 12 meters long, 7 meters wide and 2 meter depth. You have a pump which can move 17 cubic meters of water in an hour. Write a program to find how long it will take to empty your pool? (Volume = l * w * h, and flow = volume/time).

Program Code

```
l=12
w = 7
d = 2
volume  = l*w*d
flowrate= 17
time = volume/flowrate
print("Time taken to empty the tank : {:.2f} hours".format(time))
```

17. Write a program to convert temperature from centigrade (read it as float value) to Fahrenheit.

Program Code

```
c = float(input("Enter temperature in Centigrade :"))

f = 9/5 * c + 32

print("{:.2f} degree Celcius  = {:.2f} Fahrenheit".format(c,f))
```

18. Write a program that calculates the number of seconds in a day.

Program Code

```
#program to compute the number of seconds in a day
hours = 24
minutes = 60
second = 60

seconds = hours*minutes *second

print("Number of Seconds in a day= {}".format(seconds))
```

19. A car starts from a stoplight and is traveling with a velocity of 10 m/sec east in 20 seconds. Write a program to find the acceleration of the car. (acc = (vfinal−vinitial)/ time).

Program Code

```
#program to acceleration of the car
```

```
velocity = 10
time = 20

acceleration = velocity / time

print("Acceleration of the car = {:.2f} metres per \
second square".format(acceleration))
```

20. Briefly explain the conditional statements available in Python.

Decision-making is as important in any programming language as it is in life. Decision-making in a programming language is automated using conditional statements, in which Python evaluates the code to see if it meets the specified conditions. The conditions are evaluated and processed as true or false. If this is found to be true, the program is run as needed. If the condition is found to be false, the statement following the If condition is executed. Conditional Statement in Python performs different computations or actions depending on whether a specific Boolean constraint evaluates to true or false. Conditional statements are handled by IF statements in Python.

Python has 3 key Conditional Statements:

a. *If statement*
b. *If else statement*
c. *If elif statement*

The if statement is a conditional statement in python, that is used to determine whether a block of code will be executed or not. Meaning if the program finds the condition defined in the if statement to be true, it will go ahead and execute the code block inside the if statement.

Syntax:

```
if condition:
    # execute code block
```

Example:

```
If i % 2 ==0:
        print("Number is Even")
If i%2==1:
        print("Number is Odd")
```

if-else Statement:

The if statement executes the code block when the condition is true. Similarly, the else statement works in conjuncture with the if statement to execute a code block when the defined if condition is false.

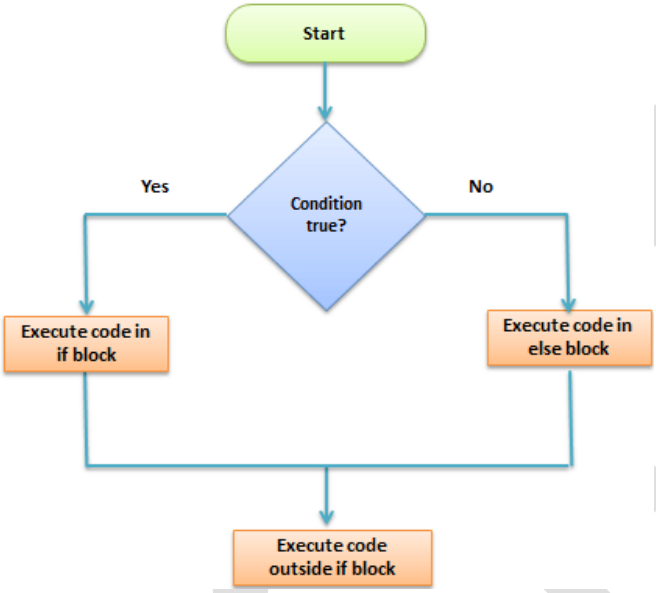
Syntax:

```
if condition:
    # execute code if condition is true
else:
    # execute code if condition if False
```

Example:

```
If a>b:
        print("A is big")
else:
        print("B is big")
```

Flow chart of if else statement



If elif statement:

The elif statement is used to check for multiple conditions and execute the code block within if any of the conditions evaluate to be true.

The elif statement is similar to the else statement in the context that it is optional but unlike the else statement, there can be multiple elif statements in a code block following an if statement.

Syntax:

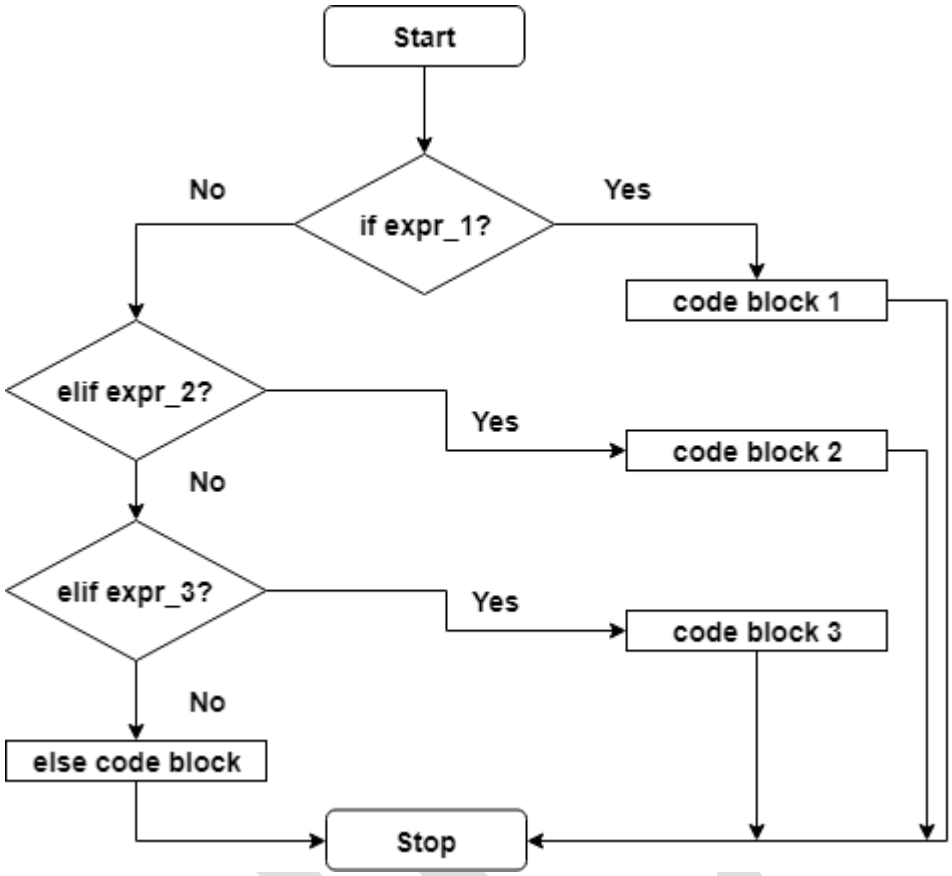
```
if condition1:
   # execute this statement
elif condition2:
   # execute this statement
.
.
else:
   # if non of the above conditions
   # evaluate to True
   # execute this statement
```

Example:

```
num = 7
if num > 0:
        print("Positive number")
elif num == 0:
        print("Zero")
else:
```

print("Negative number")

Flow Chart:



21. Explain the syntax of for loop with an example.

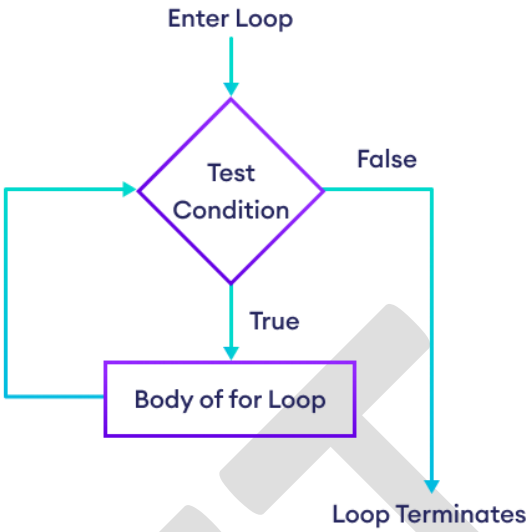
The for loop is used to run a block of code for a certain number of times. It is used to iterate over any sequences such as list, tuple, string, etc.

The syntax of the for loop is:

for val in sequence:

 # statement(s)

Here, val accesses each item of sequence on each iteration. Loop continues until we reach the last item in the sequence.



Example:

```
digits = [0, 1, 5,6]
for i in digits:
   print(i)
else:
   print("No items left.")
```

22. What is the purpose of using break and continue? Explain with suitable examples.

In Python, break and continue are loop control statements executed inside a loop. These statements either skip according to the conditions inside the loop or terminate the loop execution at some point.

Break Statement

The break statement is used to terminate the loop immediately when it is encountered.

The syntax of the break statement is:

```
break
```

A break statement is used inside both the while and for loops. It terminates the loop immediately and transfers execution to the new statement after the loop. For example, have a look at the code and its output below:

```
count = 0
while count <= 100:
                print (count)
                count += 1
                if count == 3:
                break
Program output:
0
1
2
```

In the above example loop, we want to print the values between 0 and 100, but there is a condition here that the loop will terminate when the variable count becomes equal to 3.

```
for val in sequence:
    # code
    if condition:
        break

    # code
```

```
while condition:
    # code
    if condition:
        break

    # code
```

Continue Statement

The continue statement causes the loop to skip its current execution at some point and move on to the next iteration. Instead of terminating the loop like a break statement, it moves on to the subsequent execution. The continue statement is used to skip the current iteration of the loop and the control flow of the program goes to the next iteration.

```
for i in range(0, 5):
    if i == 3:
        continue
    print(i)
```

Program output:

0
1
2
4

In the above example loop, we want to print the values between 0 and 5, but there is a condition that the loop execution skips when the variable count becomes equal to 3.

```
for val in sequence:
    # code
    if condition:
        continue

    # code
```

```
while condition:
    # code
    if condition:
        continue

    # code
```

Difference between break and continue in python

The main Difference between break and continue in python is loop terminate. In this tutorial, we will explain the use of break and the continue statements in the python language. The break statement will exist in python to get exit or break for and while conditional loop. The break and continue can alter flow of normal loops and iterate over the block of code until test expression is false.

Basis for comparison	break	continue
Task	It eliminates the execution of remaining iteration of loop	It will terminate only the current iteration of loop.
Control after break/continue	'break' will resume control of program to the end of loop enclosing that 'break'.	The 'continue' will resume the control of the program to next iteration of that loop enclosing 'continue'
causes	It early terminates the loop.	It causes the early execution of the next iteration.
continuation	The 'break 'stop the continuation of the loop.	The 'continue' does not stop the continuation of loop and it stops the current.

23. Differentiate the syntax of if...else and if...elif...else with an example.

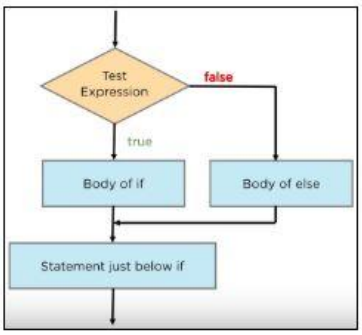
Decision making is an essential concept in any programming language and is required when you want to execute code when a specific condition is satisfied. The if else and if elif else statements are such constructs language which helps in decision making.

If-else Statement

The if-else statement is used to execute both the true part and the false part of a given condition. If the condition is true, the if block code is executed and if the condition is false, the else block code is executed.

Syntax:

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



The general way to write an if else condition	
1 <some code before>	<some code here> gets executed before checking the conditional. Typically, there's a variable that's assigned to an input.
2 if <conditional statement>:	The keyword "if" starts the conditional line, followed by a conditional statement. The if statement ends with a colon.
3 <do something>	Indented to represent code that's executed only if the condition is True.
4 else:	The keyword "else" followed by a colon
5 <do something>	Indented to represent code that's executed only if the condition is False.
6 <some code after>	Optional: This code runs regardless of the conditional

Example:

```
i=23
if i%2==0:
    print("This is the if block")
    print("i is an even number")
else:
    print("This is the else block")
    print("i is an odd number")
```

```
This is the else block
i is an odd number
```

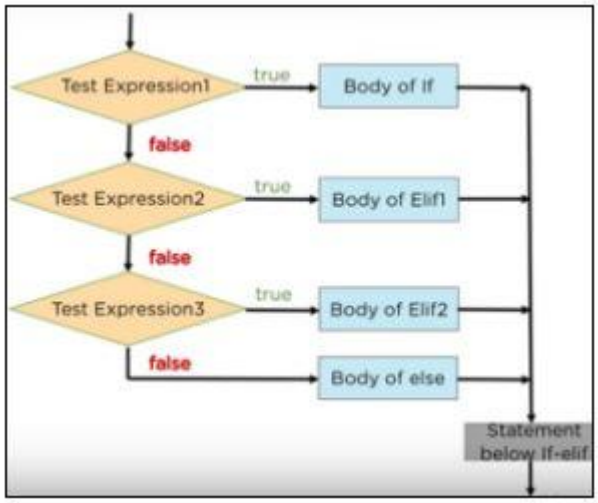
If-Elif-Else Statement

It checks the if statement condition. If that is false, the elif statement is evaluated. In case the elif condition is false, the else statement is evaluated.

Syntax:
if (condition):
statements
elif (condition):
    statements
elif condition:
    statements
.

.

else:
    statements



The general way to write an if elif else condition	
1 <some code before>	<some code here> gets executed before checking the conditional. Typically, there's an input statement here.
2 if <conditional statement>:	The keyword "if" starts the conditional line, followed by a conditional statement. The if statement ends with a colon.
3 <do something>	Indented to represent code that's executed only if the condition is True.
4 elif <conditional statement>:	The keyword "elif" followed by a colon
5 <do something>	Indented to represent code that's executed only if the condition is False.
6 else:	The keyword "else" followed by a colon
7 <do something>	
8 <some code after>	Optional: This code runs regardless of the conditional

Example Program

```
a=10
b=15
c=20
if (a>b) and (a>c):
    print("the greatest number is  a")
elif (b>a) and (b>c):
    print("the greatest is b")
else:
    print("the greatest number is c")
```

```
the greatest number is c
```

Difference between if and if elif else

- Python will evaluate all three if statements to determine if they are true.
- Once a condition in the if elif else statement is true, Python stop evaluating the other conditions.
- Because of this, if elif else is faster than three if statements.

24. Explain the use of range() function with an example.

The Python range() function returns a sequence of numbers, in a given range. The most common use of it is to iterate sequence on a sequence of numbers using Python loops.

Syntax: range(start, stop, step)

Parameters:

start: [optional] start value of the sequence

stop: next value after the end value of the sequence

step: [optional] integer value, denoting the difference between any two numbers in the sequence.

Return: Returns a range type object.

In simple terms, range() allows the user to generate a series of numbers within a given range. Depending on how many arguments the user is passing to the function, the user can decide where that series of numbers will begin and end, as well as how big the difference will be between one number and the next. Python range() function takes can be initialized in 3 ways.

range (stop) takes one argument.

range (start, stop) takes two arguments.

range (start, stop, step) takes three arguments.

When the user call range() with one argument, the user will get a series of numbers that starts at 0 and includes every whole number up to, but not including, the number that the user has provided as the stop.

When the user call range() with two arguments, the user gets to decide not only where the series of numbers stops but also where it starts, so the user don't have to start at 0 all the time. Users can use range() to generate a series of numbers from X to Y using range(X, Y).

When the user call range() with three arguments, the user can choose not only where the series of numbers will start and stop, but also how big the difference will be between one number and the next. If the user doesn't provide a step, then range() will automatically behave as if the step is 1. In this example, we are printing even numbers between 0 and 10, so we choose our starting point from 0(start = 0) and stop the series at 10(stop = 10). For printing an even number the difference between one number and the next must be 2 (step = 2) after providing a step we get the following output (0, 2, 4, 8).

Some Important points to remember about the Python range() function:

- range() function only works with the integers, i.e. whole numbers.
- All arguments must be integers. Users can not pass a string or float number or any other type in a start, stop and step argument of a range().
- All three arguments can be positive or negative.
- The step value must not be zero. If a step is zero, python raises a ValueError exception.
- range() is a type in Python
- Users can access items in a range() by index, just as users do with a list:

Example of Python range() function

```
# print first 5 integers
# using python range() function
for i in range(5):
   print(i, end=" ")
print()
```

Output:

0 1 2 3 4

25. Why would you use a try-except statement in a program?

Exception handling is one of the most important feature of Python programming language that allows us to handle the errors caused by 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. An exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs in the program, execution gets terminated. In such cases, we get a system-generated error message. However, these exceptions can be handled in Python. By handling the exceptions, we can provide a meaningful message to the user about the issue rather than a system-generated message, which may not be understandable to the user.

Handling of exception ensures that the flow of the program does not get interrupted when an exception occurs which is done by trapping run-time errors. Handling of exceptions results in the execution of all the statements in the program.

Try and Except statement is used to handle these errors within our code in Python. The try block is used to check some code for errors i.e the code inside the try block will execute when there is no error in the program. Whereas the code inside the except block will execute whenever the program encounters some error in the preceding try block.

Syntax:

```
try:
   # Some Code
except:
   # Executed if error in the
   # try block
```

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

Example:

```
Program to Check for ValueError Exception
while True:
    try:
        number = int(input("Please enter a number: "))
```

```
            print(f"The number you have entered is {number}")
        break
    except ValueError:
            print("Oops! That was no valid number. Try again…")
```

Output

Please enter a number: g
Oops! That was no valid number. Try again…
Please enter a number: 4
The number you have entered is 4

26. Explain the syntax of while loop with an example.

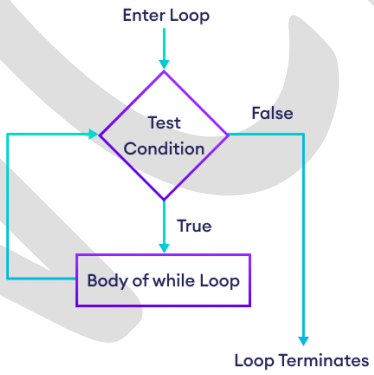
The Python while loop iteration of a code block is executed as long as the given condition, i.e., conditional_expression, is true.

Syntax of Python While Loop

```
while conditional_expression:
        Code block of while
```

The given condition, i.e., conditional_expression, is evaluated initially in the Python while loop. Then, if the conditional expression gives a boolean value True, the while loop statements are executed. The conditional expression is verified again when the complete code block is executed. This procedure repeatedly occurs until the conditional expression returns the boolean value False.

The statements of the Python while loop are dictated by indentation.
The code block begins when a statement is indented & ends with the very first unindented statement.
Any non-zero number in Python is interpreted as boolean True. False is interpreted as None and 0.



Python While Loop Example

Here we will sum of squares of the first 15 natural numbers using a while loop.

Code

```
# Python program example to show the use of while loop
num = 15
 # initializing summation and a counter for iteration
summation = 0
c = 1
while c <= num: # specifying the condition of the loop
   # begining the code block
```

```
    summation = c**2 + summation
    c = c + 1    # incrementing the counter
# print the final sum
print("The sum of squares is", summation)
```

Output:

The sum of squares is 1240

27. Differentiate between syntax error and an exception.

In any computer programming language there are rules and regulations in a combination of alphanumeric and symbols defined in a proper structured format document or fragment in that language.

A syntax error means there's an error in syntax such as misspelled keywords, a missing punctuation character, a missing bracket, a missing closing parenthesis or any errors in basic framing sequence of characters or tokens which is necessarily to be written in a particular programming language.
In compiled languages, syntax errors are detected at compile-time. A program will not compile until all syntax errors are corrected. A program will cannot be executed until all the syntax errors are rectified.

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 occurs in python.

- Syntax errors
- Logical errors (Exceptions)

Syntax Errors

Syntax errors are perhaps the most common kind of complaint you get while you are still learning Python.

Example:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
               ^
SyntaxError: invalid syntax
```

Python returns the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected.

The error is caused by (or at least detected at) the function, command, or token preceding the arrow.

In the example above, the error is detected at the function print(), since a colon (':') is missing before theprint()function and after the while Truestatement. File name and line number are also printed so you know where to look in case the input came from a script.

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 code execution are called exceptions and are not unconditionally fatal. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>

TypeError: Can't convert 'int' object to str implicitly
```

The last line of the error message indicates what happened. Here you see three different exceptions and the type is printed as part of the message. The types in the example are ZeroDivisionError, NameError and TypeError.

The string that denotes the exception type is built into the language.This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). The rest of the line provides detail based on the type of exception and what caused it.

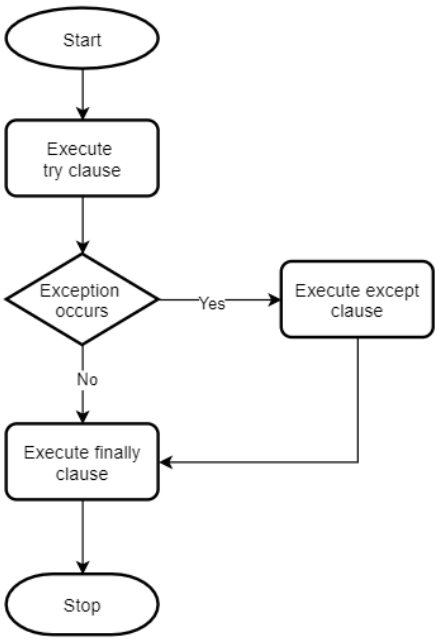
The preceding part of the error message shows the context where the exception happened in the form of a “traceback”.

28. Explain the syntax of the try-except-finally block.

Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. The Exemption Handling within Python is one of the capable component to handle the runtime errors so that the normal flow of the application can be maintained.

Syntax For try..except..finally

```
try:
   # code that may raise exception
except:
   # code that handle exceptions
else:
   # execute if no exception
finally:
   # code that clean up
```



First try block is executed
If error occurs then except block is executed.
If no error occurs then else block will be executed.
The finally block will always exceute if error occurred or not.

Example

```
a = 10
b = 0
try:
   c = a / b
   print(c)
except ZeroDivisionError as error:
   print(error)
finally:
   print('Finishing up.')
```

29. Write a program to calculate and print the Electricity bill of a given customer. units consumed by the user should be taken from the keyboard and display the total amount to be pay by the customer. The charges are as follows:

Unit	Charge/unit
upto 199	@1.20
200 and above but less than 400	@1.50
400 and above but less than 600	@1.80
600 and above	@2.00

If the bill exceeds Rs. 400 then a surcharge of 15% will be charged. If the bill is less than Rs. 400 then a minimum surcharge amount should be Rs. 100/-.

Program Code

```
units = int(input("Enter number of units consumed: "))
if units<=199:
```

```
    bill=units*1.20
elif units<400:
    bill=units*1.50
elif units<600:
    bill = units*1.80
elif units>=600:
    bill = units*2.00

surcharge=0
if bill>400:
    surcharge=bill*0.15
if bill<400:
    surcharge=100

bill=bill+surcharge

print("Bill Amount : {:.2f}".format(bill))
```

30. Write a program that uses a while loop to add up all the even numbers between 100 and 200.

Program code

```
#program to add even numbers between 100 and 200

sum =0
for i in range(100, 201):
    if i%2==0:
        sum=sum+i

print("Sum of all even numbers between 100 and 200 is : ", sum)
```

31. Write a program to print the sum of the following series.

a. 1 + ½ + 1/3 +. …. + 1/n

b. $1/1 + 2^2/2 + 3^3/3 + ……. + n^n/n$

a. **Program code for 1 + ½ + 1/3 +. …. + 1/n**

```
n=int(input("Enter the value of n : "))
sum =0.0
for i in range(1,n+1):
    sum+=1/i

print("Sum of the series : {:.2f}".format(sum))
```

b. **Program code for $1/1 + 2^2/2 + 3^3/3 + ……. + n^n/n$**

```
n=int(input("Enter the value of n : "))
sum =0.0
for i in range(1,n+1):
    sum+=i**i/i
```

```
print("Sum of the series : {:.2f}".format(sum))
```

32. Write a program to display Pascal's triangle.

```
# Print Pascal's Triangle in Python
n = int(input())
for i in range(1, n+1):
        for j in range(0, n-i+1):
                print(' ', end='')

        # first element is always 1
        C = 1
        for j in range(1, i+1):

                # first value in a line is always 1
                print(' ', C, sep='', end='')

                # using Binomial Coefficient
                C = C * (i - j) // j
        print()
```

33. **Write a program which repeatedly reads numbers until the user enters 'done'. Once 'done' is entered, print out the total, count, and average of the numbers. If the user enters anything other than a number, detect their mistake using try and except and print an error message and skip to the next number**.

```
total = 0
count = 0
while True:
        num = input("Enter a number: ")
        if num == 'done':
        print(f"Sum of all the entered numbers is {total}")
        print(f"Count of total numbers entered {count}")
        print(f"Average is {total / count}")
        break
else:
        try:
                total += float(num)
        except:
                print("Invalid input")
        continue
count += 1
```

34. Write a Program to Check Whether a Number Is Prime or Not

```
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, num):
        if num % i == 0:
            return False
    return True

print("Enter a number")
n = int(input())
if is_prime(n)"
        print("{} is a prime number".format(n))
else:
        print("{} is not a prime number".format(n))
```

35. Write a Program to Find the Factorial of a Number

```
n = int(input("Enter a number"))
fact=1
for i in range(n):
        fact = fact*i
print("Factorial of {} = {}".format(n, fact))
```

36. Write a Program to Find the Sum of All Odd and Even Numbers up to a Number Specified by the User

```
n = int(input("Enter the value of n : "))
sumodd=0
sumeven=0
for i in range(1,n+1):
        if i%2==0:
                sumeven+=i
        else:
                sumodd+=i
print("Sum of odd numbers = {}".format(sumodd))
print("Sum of even numbers = {}".format(sumeven))
```

37. Program to Repeatedly Check for the Largest Number Until the User Enters "done".

```
largest_number = int(input("Enter the largest number initially"))
check_number = input("Enter a number to check whether it is largest or not")
while check_number != "done":
        if largest_number > int(check_number):
                print(f"Largest Number is {largest_number}")
        else:
                largest_number = int(check_number)
        print(f"Largest Number is {largest_number}")
        check_number = input("Enter a number to check whether it is largest or not")
```

38. Write a Program to Display the Fibonacci Sequences up to nth Term, Where n is provided by the User.

```
def fibonacci(n):
    if n == 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]
    else:
        fib_list = [0, 1]
        for i in range(2,n):
            next_fib = fib_list[i-1] + fib_list[i-2]
            fib_list.append(next_fib)
        return fib_list

n = int(input("Enter the number of terms for the Fibonacci sequence: "))
print(fibonacci(n))
```

39. Write Python Program to Find the Sum of Digits in a Number

```
def sum_of_digits(num):
    sum = 0
    while num > 0:
        digit = num % 10
        sum += digit
        num = num // 10
    return sum

num = int(input("Enter a number: "))
print("Sum of digits:", sum_of_digits(num))
```

40. Program to Find the GCD of Two Positive Numbers

```
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
print("GCD of", num1, "and", num2, "is", gcd(num1, num2))
```

41. Write a Program to Find the Average of n Natural Numbers

```
n = int(input(“Enter the value of n : “))
sum=0
for i in range(1,n+1):
        sum+=i
```

```
average = sum/n
print("Average of given numbers = {:.2f}".format(average))
```

42. Program to Check If a Given Year Is a Leap Year

```
# Python program to check if year is a leap year or not
year = int(input("Enter a year in four digit format: "))
# divided by 100 means century year (ending with 00)
# century year divided by 400 is leap year
if (year % 400 == 0) and (year % 100 == 0):
   print("{0} is a leap year".format(year))

# not divided by 100 means not a century year
# year divided by 4 is a leap year
elif (year % 4 ==0) and (year % 100 != 0):
   print("{0} is a leap year".format(year))

# if not divided by both 400 (century year) and 4 (not century year)
# year is not leap year
else:
   print("{0} is not a leap year".format(year))
```

43. Calculate the Value of sin(x) up to n Terms Using the Series

```
# Import Module
import math
# Create sine function
def sin( x, n):
        sine = 0
        for i in range(n):
                sign = (-1)**i
                pi = 22/7
                y = x*(pi/180)
                sine += ((y**(2.0*i+1))/math.factorial(2*i+1))*sign
        return sine

# Enter value in degree in x
x = int(input("Enter value of x: "))
# Enter number of terms
n = int(input("Enter value of n : "))
# call sine function
print(round(sin(x,n),2))
```

44. Program to Check If a 3 Digit Number Is Armstrong Number or Not

In case of an Armstrong number of 3 digits, the sum of cubes of each digit is equal to the number itself. For example:

153 = 1*1*1 + 5*5*5 + 3*3*3 // 153 is an Armstrong number.

Python program to check if the number is an Armstrong number or not

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

```
# initialize sum
sum = 0

# find the sum of the cube of each digit
temp = num
while temp > 0:
   digit = temp % 10
   sum += digit ** 3
   temp //= 10

# display the result
if num == sum:
   print(num,"is an Armstrong number")
else:
   print(num,"is not an Armstrong number")
```

45. Define function. What are the advantages of using a function?

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide a way to organize code and make it more readable, reusable, and modular. Functions can take input in the form of parameters and can return an output.

In Python, a function is defined using the def keyword, followed by the function name, a pair of parentheses () containing the function's parameters, and a colon :. The code that makes up the function is indented under the definition.

Here's an example of a simple function in Python:

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

Advantages of using functions:

a. **Code Reusability:** Functions can be used multiple times in a program, reducing the need to write repetitive code.
b. **Modularity:** By dividing a program into smaller functions, it becomes easier to understand, test and modify the code.
c. **Abstraction:** Functions can hide the complexity of a task, allowing the user to focus on the desired outcome rather than the underlying implementation.
d. **Debugging:** Functions are isolated units of code, making it easier to identify and fix errors.
e. **Improved readability:** By dividing the code into functions, it becomes easier to read and understand the program.
f. **Reducing Complexity:** Functions can help to decompose a problem into smaller sub-problems and solve them separately which makes the overall solution less complex and easier to understand.
g. **Flexibility:** Functions can take different inputs and return different outputs based on the inputs which makes it more flexible.

Functions are an essential tool in any programming language, they enable you to write more efficient, readable, and maintainable code.

46. Differentiate between user-defined function and built-in functions.

A user-defined function is a function that is created and defined by the user, while a built-in function is a function that is already defined and available for use in the programming language.

User-defined functions are created to perform specific tasks that are unique to the program being written. They are defined by the user using the def keyword, and the code inside the function is written by the user. For example, in python, if you want to calculate the area of a circle, you can define a function area_of_circle(radius) that takes radius as an input and returns the area of the circle.

On the other hand, built-in functions are pre-defined functions that are part of the programming language and are available for use without the need to define them. They are already available in the interpreter and are optimized for performance. For example, in Python, the print() function is a built-in function that is used to output data to the screen.

In summary, user-defined functions are created by the user to perform specific tasks that are unique to the program being written, while built-in functions are pre-defined functions that are part of the programming language and are available for use without the need to define them.

47. Explain with syntax how to create a user-defined functions and how to call the user-defined function from the main function.

In Python, a user-defined function is defined using the def keyword, followed by the function name, a pair of parentheses () containing the function's parameters, and a colon :. The code that makes up the function is indented under the definition. Here's an example of a simple user-defined function in Python:

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

This function takes a single parameter name and prints a greeting message to the screen.

To call a user-defined function, you simply use the function name followed by a pair of parentheses () containing the function's arguments. For example, to call the greet() function defined above, you would write the following code:

```
greet("John")
```

You can also call a user-defined function from another user-defined function or from the main function. Here's an example of how to call a user-defined function from the main function:

```
def main():
    name = input("What's your name? ")
    greet(name)

def greet(name):
    print("Hello, " + name + "!")

if __name__ == "__main__":
    main()
```

In this example, the main() function is called when the script is executed and it takes an input from the user and calls the greet() function with the input as an argument.

It's worth noting that when the script is executed, if __name__ == "__main__": check is used to make sure that the code inside the main() function is executed only when the script is run and not when the script is imported as a module into another script.

48. Explain the built-in functions with examples in Python.

In Python, built-in functions are pre-defined functions that are part of the programming language and are available for use without the need to define them. They can be used to perform various tasks such as input/output, data type conversions, mathematical calculations, etc. Here are some examples of commonly used built-in functions in Python:

print(): *This function is used to output data to the screen. It takes one or more arguments, which can be strings, numbers, or variables, and prints them to the screen.*

```
print("Hello World!")
x = 5
y = 3
print("The value of x is", x, "and the value of y is", y)
```

input(): *This function is used to take input from the user. It takes a string as an argument, which is used as a prompt for the user, and returns the input as a string.*

```
name = input("What's your name? ")
print("Hello, " + name + "!")
```

len(): *This function is used to find the length of a string or the number of elements in a list or tuple.*

```
string = "Hello World!"
print(len(string))  # 12

numbers = [1, 2, 3, 4, 5]
print(len(numbers))  # 5
```

str(), int(), float(), bool(): *These functions are used to convert data from one data type to another.*

```
string = "123"
number = int(string)
print(number + 1)  # 124

number = 5
string = str(number)
print(string + " is a number")  # "5 is a number"

a = float(5)
b = bool(1)
```

min(), max(), sum(): *These functions are used to find the minimum, maximum, and sum of elements in a list or tuple.*

```
numbers = [1, 2, 3, 4, 5]
print(min(numbers))  # 1
print(max(numbers))  # 5
print(sum(numbers))  # 15
```

range(): *This function is used to generate a range of numbers. It takes up to three arguments: start, stop, and step.*

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

These are just a few examples of the many built-in functions that are available in Python. Some other commonly used built-in functions include abs(), pow(), round(), sorted(), type(), etc. With the help of documentation, you can explore more built-in functions and their usage.

49. What is a function? How to define a function in python? Write a program using function to find out the given string is palindrome or not.

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide a way to organize code and make it more readable, reusable, and modular. Functions can take input in the form of parameters and can return an output.

In Python, a function is defined using the def keyword, followed by the function name, a pair of parentheses () containing the function's parameters, and a colon :. The code that makes up the function is indented under the definition. Here's an example of a simple user-defined function in Python:

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

Here is an example of a function that checks whether a given string is a palindrome or not:

```
def check_palindrome(string):
    if string == string[::-1]:
        return True
    else:
        return False

string = input("Enter a string: ")
if check_palindrome(string):
    print("The string is a palindrome.")
else:
    print("The string is not a palindrome.")
```

In this example, the function check_palindrome() takes a string as an input, and then uses the slicing technique, where string[::-1] is used to reverse the string and compare it with the original string. If they are same then it returns True else False.
The input is taken from the user and passed to the function, then it checks the output of function whether it's True or False, based on that it prints the respective message.

50. Write a python program to create a function called collatz() which reads as parameter named number. If the number is even it should print and return number//2 and if the number is odd then it should print and return 3*number+1. The function should keep calling on that number until the function returns a value 1.

```
def collatz(number):
    while number != 1:
        if number % 2 == 0:
            print(number)
            number = number // 2
```

```
        else:
            print(number)
            number = 3 * number + 1
    print(number)

num = int(input("Enter a number: "))
collatz(num)
```

51. Write a user defined function to receive a number and display a message to indicate whether it is positive or negative number. Implement the program to check 5 numbers entered by the user.

```
def check_sign(num):
    if num > 0:
        return "Positive"
    elif num < 0:
        return "Negative"
    else:
        return "Zero"

for i in range(5):
    num = int(input("Enter a number: "))
    print(check_sign(num))
```

52. Differentiate between local and global variables with suitable examples.

A local variable is a variable that is defined within a function and can only be accessed within that function. The scope of a local variable is limited to the function in which it is defined, and it is not accessible outside of that function. Here's an example of a local variable:

```
def my_function():
    x = 5
    print(x)

my_function()
print(x) # This will result in an error, because x is a local variable and it's not accessible outside of the function
```

In this example, the variable x is defined within the my_function and can only be accessed within that function. If we try to access it outside of the function, it will result in an error because it's out of the scope of the variable.

On the other hand, a global variable is a variable that is defined outside of any function and can be accessed from anywhere in the code. The scope of a global variable is the entire program. Here's an example of a global variable:

```
x = 5

def my_function():
    print(x)

my_function()
print(x)
```

In this example, the variable x is defined outside of any function and can be accessed from anywhere in the code. The my_function can access the global variable x and the print statement outside of the function also can access it.

It's worth noting that, it's possible to access global variables inside a function, but if you want to modify their value inside a function, you need to use the global keyword to inform Python that you want to modify the global variable, not create a new local variable with the same name.

53. Demonstrate how functions return multiple values with an example.

In Python, a function can return multiple values by using the return statement followed by a comma-separated list of values. Here's an example of a function that returns multiple values:

```
def divide_and_remainder(x, y):
   quotient = x // y
   remainder = x % y
   return quotient, remainder

result = divide_and_remainder(10, 3)
print(result)
```

In this example, the function divide_and_remainder() takes two parameters, x and y, and calculates the quotient and remainder of x divided by y. The function returns the quotient and remainder as a tuple using the return statement. The returned values are then assigned to the variable result, which can be accessed as a tuple.

You can also unpack the returned tuple into separate variables:

```
quotient, remainder = divide_and_remainder(10, 3)
print(quotient)
print(remainder)
```

It's worth noting that returning multiple values is not the only way to return multiple results from a function, you can use data structures such as lists, tuples, or dictionaries to return multiple values as well.

In summary, you can return multiple values from a function in Python by separating the values with a comma in the return statement. The returned values can be accessed as a tuple and can be unpacked into separate variables, or you can use data structures like lists, tuple, or dictionaries to return multiple values.

54. Write a program using functions to perform the arithmetic operations.

```
def add(x, y):
   return x + y

def subtract(x, y):
   return x - y

def multiply(x, y):
   return x * y

def divide(x, y):
   return x / y

x = float(input("Enter the first number: "))
y = float(input("Enter the second number: "))
```

```
print("Addition: ", add(x, y))
print("Subtraction: ", subtract(x, y))
print("Multiplication: ", multiply(x, y))
print("Division: ", divide(x, y))
```

55. Write a program to find the largest of three numbers using functions.

```
def largest_number(x, y, z):
    if x > y and x > z:
        return x
    elif y > x and y > z:
        return y
    else:
        return z

x = float(input("Enter the first number: "))
y = float(input("Enter the second number: "))
z = float(input("Enter the third number: "))

print("The largest number is:", largest_number(x, y, z))
```

56. Write a Python program using functions to find the value of nPr and nCr.

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

def nPr(n, r):
    return factorial(n) / factorial(n-r)

def nCr(n, r):
    return factorial(n) / (factorial(r) * factorial(n-r))

n = int(input("Enter the value of n: "))
r = int(input("Enter the value of r: "))

print("nPr: ", nPr(n, r))
print("nCr: ", nCr(n, r))
```

57. Write a Python function named area that finds the area of a pentagon.

```
import math

def area(s):
    return (5 * (s ** 2)) / (4 * math.tan(math.pi / 5))

s = float(input("Enter the length of a side of the pentagon: "))
```

```
print("The area of the pentagon is:", area(s))
```

58. Write a program using functions to display Pascal's triangle.

```
def pascal_triangle(n):
    for i in range(n):
        for j in range(i+1):
            print(binomial_coeff(i, j), end=" ")
        print()

def binomial_coeff(n, k):
    res = 1
    if k > n - k:
        k = n - k
    for i in range(k):
        res = res * (n - i)
        res = res // (i + 1)
    return res

n = int(input("Enter the number of rows: "))
pascal_triangle(n)
```

59. Write a program using functions to print harmonic progression series and its sum till N terms.

```
def harmonic_progression(n):
    series = []
    for i in range(1,n+1):
        series.append(1/i)
    return series

def harmonic_sum(n):
    sum = 0
    for i in range(1,n+1):
        sum += 1/i
    return sum

n = int(input("Enter the number of terms: "))
series = harmonic_progression(n)
print("Harmonic progression series: ", series)
print("Sum of the series: ", harmonic_sum(n))
```

60. Write a code segment in python to find the largest value in a list.

```
#Assuming the list is named 'numbers'

largest_number = float("-inf")
for number in numbers:
    if number > largest_number:
        largest_number = number
```

print(largest_number)

This code will iterate through each number in the list, and compares it with the current largest_number, if the number is greater than the largest_number, it assigns the number to largest_number, at the end of the loop, largest_number will have the largest value in the list.

Alternatively, you can also use python's built-in max() function to directly find the maximum value in the list like this:

#Assuming the list is named 'numbers'

largest_number = max(numbers)
print(largest_number)

61. What is list? Explain the concept of slicing and indexing with proper examples.

A list in Python is a collection of ordered items, which can be of any data type, such as numbers, strings, or other objects. Lists are written in square brackets, with items separated by commas. For example:

```
numbers = [1, 2, 3, 4, 5]
words = ["apple", "banana", "cherry"]
```

Slicing and indexing are ways to access specific elements or sections of a list.

Indexing
In Python, lists are indexed starting from 0. This means that the first item in a list has an index of 0, the second item has an index of 1, and so on. To access an item in a list using its index, you can use square brackets:

```
numbers = [1, 2, 3, 4, 5]
print(numbers[0]) # Output: 1
print(numbers[2]) # Output: 3
```

Slicing
Slicing is used to extract a portion of a list, called a slice. You can use the slice operator (:) to extract a slice from a list. The slice operator takes two arguments: the start index and the end index. The start index is the first item you want to include in the slice, and the end index is the first item you want to exclude from the slice. For example:

```
numbers = [1, 2, 3, 4, 5]
print(numbers[1:3]) # Output: [2, 3]
print(numbers[:3]) # Output: [1, 2, 3]
print(numbers[2:]) # Output: [3, 4, 5]
```

In the first example above, we're getting the slice of numbers from index 1 to index 3 (exclusive), it returns the list [2,3].
In the second example, We're getting the slice of numbers from index 0 to index 3 (exclusive), it returns the list [1,2,3].
In the last example, we're getting the slice of numbers from index 2 to end of the list it returns the list [3,4,5].

In slicing we can also use negative indexes, in that case it starts counting from the end of the list like this:

```
numbers = [1, 2, 3, 4, 5]
print(numbers[-1]) # Output: 5
print(numbers[-3:]) # Output: [3, 4, 5]
```

In the first example above, we're getting the last element of the list using -1 as index.
In the second example, we're getting the slice of numbers from index -3 to end of the list it returns the list [3,4,5].

62. For a given list num=[45,22,14,65,97,72], write a python program to replace all the integers divisible by 3 with "ppp" and all integers divisible by 5 with "qqq" and replace all the integers divisible by both 3 and 5 with "pppqqq" and display the output.

```
num = [45, 22, 14, 65, 97, 72]
for i in range(len(num)):
    if num[i] % 3 == 0 and num[i] % 5 == 0:
        num[i] = "pppqqq"
    elif num[i] % 3 == 0:
        num[i] = "ppp"
    elif num[i] % 5 == 0:
        num[i] = "qqq"

print(num)
```

63. Write a python program to read n numbers from the keyboard, store them in a list and print only odd numbers among them.

```
n = int(input("Enter the number of numbers: "))
numbers = []

for i in range(n):
    number = int(input("Enter a number: "))
    numbers.append(number)

odd_numbers = [x for x in numbers if x%2 != 0]
print("Odd numbers in the list: ", odd_numbers)
```

64. Write a python program to read two sets of numbers and extract the numbers from the first set which are not present in the second set.

```
set1 = set(map(int, input("Enter numbers of first set separated by space: ").split()))
set2 = set(map(int, input("Enter numbers of second set separated by space: ").split()))

not_in_set2 = set1.difference(set2)
print("Numbers in first set but not in second set:", not_in_set2)
```

65. Explain the following functions performed on list. i. append ii. Pop iii. del iv. extend v. remove vi. sort

i. append: The append() method is used to add an element to the end of a list. It takes a single argument, which is the element to be added. For example:

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

```
print(numbers)  # Output: [1, 2, 3, 4]
```

ii. pop: The pop() method is used to remove and return an element from a list. If no index is specified, it removes and returns the last element. If an index is specified, it removes and returns the element at that index. For example:

```
numbers = [1, 2, 3]
print(numbers.pop())  # Output: 3
print(numbers)  # Output: [1, 2]

print(numbers.pop(0))  # Output: 1
print(numbers)  # Output: [2]
```

iii. del: The del keyword is used to delete elements from a list. It takes the index of the element to be deleted as an argument. For example:

```
numbers = [1, 2, 3]
del numbers[1]
print(numbers)  # Output: [1, 3]
```

iv. extend: The extend() method is used to add multiple elements to a list. It takes an iterable (such as a list or a tuple) as an argument and adds each element of the iterable to the list. For example:

```
numbers = [1, 2, 3]
numbers.extend([4, 5, 6])
print(numbers)  # Output: [1, 2, 3, 4, 5, 6]
```

v. remove: The remove() method is used to remove the first occurrence of a specified element from a list. It takes a single argument, which is the element to be removed. For example:

```
numbers = [1, 2, 3, 2, 4, 2]
numbers.remove(2)
print(numbers)  # Output: [1, 3, 2, 4, 2]
```

vi. sort: The sort() method is used to sort the elements of a list in ascending order. The method sorts the elements in place and does not return a new list. For example:

```
numbers = [3, 2, 1, 4, 5]
numbers.sort()
print(numbers)  # Output: [1, 2, 3, 4, 5]
```

It can also take a key and reverse argument to sort the list in descending order or using a specific key

```
numbers = [3, 2, 1, 4, 5]
numbers.sort(key=lambda x: -x)
print(numbers)  # Output: [5, 4, 3, 2, 1]
```