# Chapter 1 : Introduction-Fundamentals

## 1.1.What Is Computer Science?

Computer science is fundamentally about is computational problem solving —that is, solving problems by the use of computation

### The Essence of Computational Problem Solving

In order to solve a problem computationally, two things are needed:

- A representation that captures all the relevant aspects of the problem.
- An algorithm that solves the problem by use of the representation.

### Limits of Computational Problem Solving

Any algorithm that correctly solves a given problem must solve the problem in a reasonable amount of time, otherwise it is of limited practical use.

## 1.2 Computer Algorithms

### What Is an Algorithm?

An algorithm is a finite number of clearly described, unambiguous “doable” steps that can be systematically followed to produce a desired result for given input in a finite amount of time.

### Algorithms and Computers: A Perfect Match

Because computers can execute instructions very quickly and reliably without error, algorithms and computers are a perfect match.

## 1.3 Computer Hardware

Computer hardware comprises the physical part of a computer system. It includes the all-important components of the central processing unit (CPU) and main memory. It also includes peripheral components such as a keyboard, monitor, mouse, and printer.

### Binary representation

All information within a computer system is represented using only two digits, 0 and 1, called binary representation.

### The Binary Number System

The term bit stands for binary digit. A byte is a group of bits operated on as a single unit in a computer system, usually consisting of eight bits.

### Operating Systems—Bridging Software and Hardware

An operating system is software that has the job of managing the hardware resources of a given computer and providing a particular user interface.

**BASE-2 REPRESENTATION :**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 | |
| 2^7 | 2^6 | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 | |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | = 99 |

### 1.4 Computer Software

Computer software is a set of program instructions, including related data and documentation, that can be executed by computer.

**Syntax, Semantics, and Program Translation :**

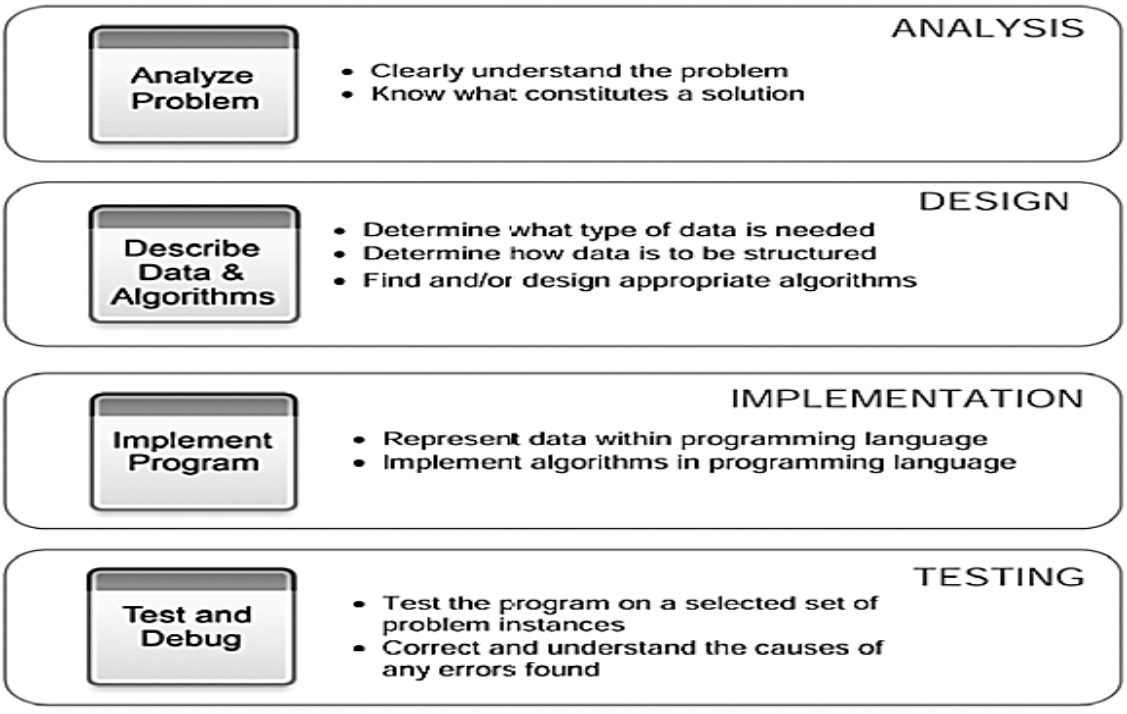
- The syntax of a language is a set of characters and the acceptable sequences of those characters.
- The semantics of a language is the meaning associated with each syntactically correct sequence of characters.
- A compiler is a translator program that translates programs directly into machine code to be executed by the CPU.
- An interpreter executes program instructions in place of ("running on top of") the CPU.

Syntax errors are caused by invalid syntax. Semantic (logic) errors are caused by errors in program logic.

**Procedural vs. Object-Oriented Programming**

- Python supports both procedural and object-oriented programming.
- Procedural programming and object-oriented programming are two major programming paradigms in use today.

### 1.5 THE PROCESS OF COMPUTATIONAL PROBLEM SOLVING



**1.6 What is python???**

- Python is a general purpose programming language that is often applied in scripting roles.
- Python is programming language as well as scripting language.
- Python is also called as Interpreted Language.

**History of python**

- Guido van Rossum is the creator of the Python programming language released in the early 1990s.
- Its name comes from a 1970s British comedy television show called Monty Python's Flying Circus.
- Open sourced from the beginning.

**Features of python**

- Easy to learn and use
- Interpreted
- Cross platform language
- Free and open source
- Object-Oriented
- Dynamically Typed Language
- Large Standard Library
- GUI programming support

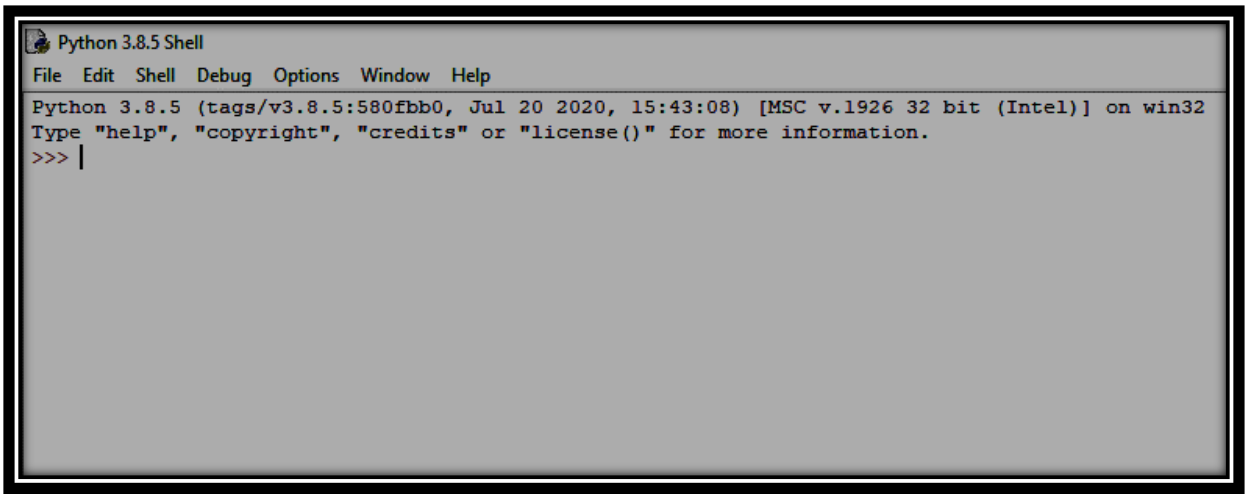
**Applications of python**

- Web and Internet Development
- GUI
- Software Development
- AI and Machine Learning
- Database access
- Network Programming
- Games and 3D Graphics

**Who uses python today??**


**Installing of python**

- Python is well supported and freely available at https://www.python.org/downloads/
- Latest version : 3.8.5




**Python IDLE**

- An Integrated Development Environment (IDLE) is a bundled set of software tools for program development.
- An editor for creating and modifying programs
- A translator for executing programs
- A program debugger provides a means of taking control of the execution of a program to aid in finding program errors

The Python Standard Library is a collection of modules, each providing specific functionality beyond what is included in the core part of Python.

**Python character set**

- Letters: Upper case and lower case letters
- Digits: 0,1,2,3,4,5,6,7,8,9
- Special Symbols: Underscore (_), (,), [,], {,}, +, -, *, &, ^, %, $, #, !, Single quote(‘), Double quotes(“), Back slash(\), Colon(:), and Semi Colon (;)
- White Spaces: (‘\t\n\x0b\x0c\r’), Space, Tab.

**Token**

- A program in Python contains a sequence of instructions.
- Python breaks each statement into a sequence of lexical components known as tokens.

**Variables**

A variable is “a name that is assigned to a value.”

- The assignment operator, = , is used to assign values to variables.
- An immutable value is a value that cannot be changed.

```
Eg:    >>>num=10
       >>> k=num
       >>> print(k)
O/P : 10
```

In Python the same variable can be associated with values of different type during program execution.

```
Eg :    var = 12          integer
        var = 12.45       float
        var = 'Hello'     string
```

**Basic Input and Output**

In Python, input is used to request and get information from the user, and print is used to display information on the screen.

**Variable Assignment and Keyboard Input**

- The value can come from the user by use of the input function.

Eg :

```
>>> name=input('what is your name : ')

        O/P : what is your name : ABC
```

- All input is returned by the input function as a string type.
- For the input of numeric values, Python provides built-in type conversion functions int() and float().

Eg :

```
>>> age=int(input('Enter your age : '))
    O/P : Enter your age : 18
```

All input is returned by the input function as a string type. Built-in functions int() and float() can be used to convert a string to a numeric type.

## Chapter 2 : Data and Expression

### 2.1 Literals

- A literal is a sequence of one or more characters that stands for itself.
- Literals often referred to as constant values that have a fixed value.

Python allows several kinds of literals:

- Numeric Literals
- String Literals

**Numeric Literals**

- A numeric literal is a literal containing only the digits 0–9, a sign character (1 or 2) and a possible decimal point.
- Commas are never used in numeric literals.

There are 3 Numerical types

- integers : positive or negative whole numbers with no decimal points ( Eg : 2, -6, 3, 5 )
- Floating point : Consists of Fractional part. ( Eg. 1.4, 5.6 )
- Complex numbers ( Eg. 2+3j )

**Limits of Range in floating point representation**

- No limit to the size of an integer that can be represented in Python.
- Floating-point values, Python providing a range of 10 -308 to 10 +308 with 16 to 17 digits of precision.

**Limitations of floating point representation**

- Arithmetic overflow problem :
  - If you are trying to multiply two long floating point integers, we get the result as inf ( infinity )

Eg : >>>1.5e200 * 2.0e210

>>> inf

- Arithmetic underflow problem :
  - This problem occurs in division.
  - If denominator is larger than numerator, then it will result in zero.
    Eg: 1/10000=0.00001

- Loss of precision problem :
  - This problem occurs in division.
  - If numerator divided by denominator, then if the result is never ending.
    Eg : 10/3 = 3.33333

**String literals in Python**

- Single Quote ➔ ‘ ‘
- Double Quote ➔ “ “
- Triple Quote ➔ ‘’’ ‘’’
  - ➔ Represent the string either in ‘ PYTHON ’ or “ PYTHON “
  - ➔ Triple Quote is used to represent “ multi line string “

Eg :

‘’’ WELCOME TO

PYTHON PROGRAMMING ‘’’

**Number Formatting**

The built-in format function can be used to produce a value containing a specific number of decimal places.

Syntax :

format(value, format_specifier)
where value is the value to be displayed
format_specifier can contain specific number of decimal places.

Eg: format(12/5, '.2f') -> '2.40'
format(5/7, '.2f') -> '0.71‘
format specifier '.2f' rounds the result to two decimal places.

**String Formatting**

- A built-in-function used to control how the strings are displayed.
  Syntax :
  format(value, format_specifier)
  where value is the value to be displayed
  format_specifier can contain a combination of formatting options.
- Formatting options can be left ( ‘<‘ ), right ( ‘ > ‘ ), center ( ‘ ^ ‘)
- Default formatting is LEFT justified
  Example :
- To produce the string 'Hello' left-justified in a field width of 20 characters would be done as follows :
  format('Hello', ' < 20') → 'Hello '
- To right-justify the string, the following would be used
  format('Hello', ' > 20') → ' Hello‘

- To center the string the '^' character is used

format('Hello', '^20') → '  Hello  '

**ESCAPE SEQUENCE IN PYTHON**

- Control characters are nonprinting characters used to control the display of output.
- Represented by a combination of characters called an escape sequence .
- An escape sequence is a string of one or more characters used to denote control characters.
- To print the special characters in the screen, we use the escape sequence.

The backslash (\) serves as the escape character in Python.

**Table 6.1 Escape sequences**

| Escape sequence | Character represented |
|---|---|
| \' | Single-quote character |
| \" | Double-quote character |
| \\ | Backslash character |
| \a | Bell character |
| \b | Backspace character |
| \f | Formfeed character |
| \n | Newline character |
| \r | Carriage return character (not the same as \n) |
| \t | Tab character |
| \v | Vertical tab character |

EXAMPLE :

```
>>>print('Hello\nJennifer Smith')
O/P:
   Hello
   Jennifer Smith
>>> print ('what\'s your name?')
    O/P : what's your name?
>>> print ('what's your name?')
     O/P : SyntaxError: invalid syntax
```

**Implicit Line Joining**

- Matching parentheses, square brackets, and curly braces can be used to span a logical program line on more than one physical line.

**Explicit Line Joining**

- Program lines may be explicitly joined by use of the backslash (\).

**Identifier**

- An identifier is a sequence of one or more characters used to name a given program element.

**Rules for Framing Identifiers :**

- Python is case sensitive.
  Eg : username is not same as userNAME.
- Identifiers may contain letters and digits, but cannot begin with a digit.
- The underscore character, _, is also allowed but it should not be used as the first character.
- Spaces are not allowed as part of an identifier.

**NAMING AN IDENTIFIER**

| **Valid identifiers** | **Invalid identifiers** | **Reason invalid** |
|---|---|---|
| totalSales | 'totalsales' | Quotes not allowed |
| totalsales | Total sales | Spaces not allowed |
| salesFor2020 | 2020sales | Cannot begin with a digit |
| Sales_for_2020 | _sales2020 | Should not begin with an underscore |

To check whether a given identifier is a keyword in python.

```
>>> 'exit' in dir(__builtins__)
O/P : True
>>> 'exit_program' in dir(__builtins__)
O/P : False
```

**Keywords in python**

- Python keywords are special reserved words that have specific meanings.
- Python 3.8, there are thirty-five keywords.
- A keyword is an identifier that has predefined meaning in a programming language.
- Cannot be used as a "regular" identifier.
  Eg: >>> and=10

O/P : SyntaxError: invalid syntax

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

The keyword module in python provides two helpful members for dealing with keywords.

- kwlist provides a list of all the python keywords for the version which you are running.
- iskeyword() provides a way to determine if a string is also a keyword.

Eg :

```
>>> import keyword
>>> keyword.kwlist
```

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

```
>>> len(keyword.kwlist)
```

O/P : 35

**Datatypes in python**

Value :

➔ A Value can be any letter, number or string.

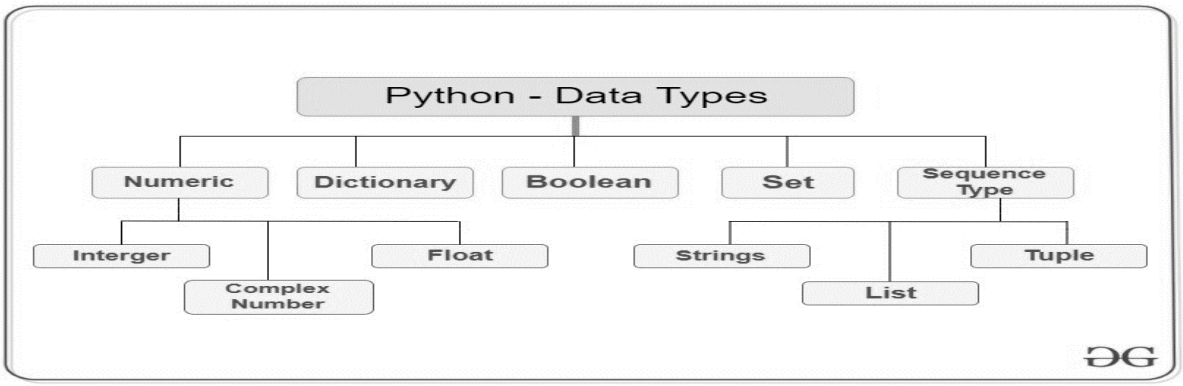
Eg, Values are 2, 42.0, and 'Hello, World!'. (These values belong to different datatypes.)

Data type:

➔ Every value in Python has a data type.

➔ It is a set of values, and the allowable operations on those values.

type() function is used to determine the type of datatype.




**NUMBERS**

- Number data type stores Numerical Values.
- This data type is immutable [i.e. values/items cannot be changed].
- Pythn supports integers, floating point numbers and complex numbers.

| Integers | Long | Float | Complex |
|---|---|---|---|
| They are often called as integers or int. They are positive or negative whole numbers with no decimal points. | They are long integers. They can also be represented in octal and hexa decimal representation | They are written with a decimal point dividing the integer and the fractional parts. | They are of the form a+bj Where a and b are floats and j represents the square root of -1(which is an imaginary number). Real part of the number is a, and the imaginary part is b. |

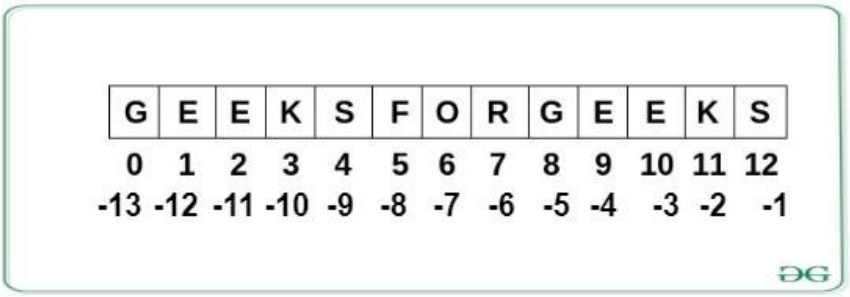
| Eg : 65 | Eg: 5692431L | Eg : 56.778 | Eg : 5+3j |
|---|---|---|---|

**Sequence**

- A sequence is an ordered collection of items, indexed by positive integers.
- It is a combination of mutable (value can be changed) and immutable (values cannot be changed) datatypes.
- There are three types of sequence data type available in Python, they are
  1. Strings, 2. Lists, 3. Tuples

**Strings**

- A String in Python consists of a series or sequence of characters.
  ➔Single quotes(' ') E.g., 'This a string in single quotes' ,
  ➔Double quotes(" ") E.g., "'This a string in double quotes"' ,
  ➔Triple quotes(""" """)E.g., """This is a paragraph. It is made up of multiple lines and sentences."""
- Individual character in a string is accessed using a subscript(index).
- Strings are Immutable i.e the contents of the string cannot be changed after it is created.




**Lists**

- List is an ordered sequence of items. Values in the list are called elements /items.
- It can be written as a list of comma-separated items (values) between square brackets[].
- Items in the lists can be of different datatypes.

Eg : lt = [ 10, -20, 15.5, ‘ABC’, “XYZ” ]

**Tuples**

- In tuple the set of elements is enclosed in parentheses ( ).
- A tuple is an immutable list.
- Once a tuple has been created, you can't add elements to a tuple or remove elements from the tuple.

**Benefit of Tuple:**

- Tuples are faster than lists.
- If the user wants to protect the data from accidental changes, tuple can be used.
- Tuples can be used as keys in dictionaries, while lists can't.
- Altering the tuple data type leads to error.

Eg : tpl = ( 10, -20, 15.5, ‘ABC’, “XYZ” )

**Dictionaries**

- A dictionary maps keys to values.

- Lists are ordered sets of objects, whereas dictionaries are unordered sets.
- Dictionary is created by using curly brackets. i,e.{ }
- Dictionaries are accessed via keys and not via their position.
- The values of a dictionary can be any Python data type. So dictionaries are unordered key-value pairs(The association of a key and a value is called a key- value pair)

Eg : Creating a dictionary:

```
>>> food = {"ham":"yes", "egg" : "yes", "rate":450 }
```

**Set :**

- Python also provides two set types, set and frozenset.
- The set type is mutable, while frozenset is immutable.
- They are unordered collections of immutable objects.

**Boolean :**

- The boolean data type is either True or False.
- In Python, boolean variables are defined by the True and False keywords.
- The keywords True and False must have an Upper Case first letter.

## OPERATORS IN PYTHON

- An operator is a symbol that represents an operation that may be performed on one or more operands.
- Operators that take one operand are called unary operators.
- Operators that take two operands are called binary operators.
  - ➔ 20 - 5 → 15 ( - as binary operator)
  - ➔ - 10 * 2 → -20 ( - as unary operator)

**Types of operators**

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operator
- Bitwise Operator
- Identity Operator
- Membership Operator

**Arithmetic Operators**

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

| Operator | Meaning | Example |
|---|---|---|
| + | Add two operands or unary plus | x + y+ 2 |
| - | Subtract right operand from the left or unary minus | x - y- 2 |

| | | |
|---|---|---|
| * | Multiply two operands | x * y |
| / | Divide left operand by the right one (always results into float) | x / y |
| % | Modulus - remainder of the division of left operand by the right | x % y (remainder of x/y) |
| // | Floor division - division that results into whole number adjusted to the left in the number line | x // y |
| ** | Exponent - left operand raised to the power of right | x**y (x to the power y) |

**Relational Operators**

- Relational operators are used to compare values. It returns either True or False according to the condition.

| Operator | Meaning | Example |
|---|---|---|
| > | Greater than - True if left operand is greater than the right | x > y |
| < | Less than - True if left operand is less than the right | x < y |
| == | Equal to - True if both operands are equal | x == y |
| != | Not equal to - True if operands are not equal | x != y |
| >= | Greater than or equal to - True if left operand is greater than or equal to the right | x >= y |
| <= | Less than or equal to - True if left operand is less than or equal to the right | x <= y |

**Logical operators**

- Logical operators are the and, or, not operators.

| Operator | Meaning | Example |
|---|---|---|
| and | True if both the operands are true | x and y |
| or | True if either of the operands is true | x or y |
| not | True if operand is false (complements the operand) | not x |

Eg:

```
x = True
y = False
print('x and y is',x and y)
print('x or y is',x or y)
print('not x is',not x)
```

**Assignment Operator**

- Assignment operators are used to assign the values to variables.

| Operator | Example | Equivalent to |
|---|---|---|
| = | 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 |

**BITWISE OPERATORS**

- Bitwise operators are working with individual bits of data.

| Operator | Meaning | Example |
|---|---|---|
| & | Bitwise AND | x & y = 0 (0000 0000) |
| \| | Bitwise OR | x \| y = 14 (0000 1110) |
| ~ | Bitwise NOT | ~x = -11 (1111 0101) |
| ^ | Bitwise XOR | x ^ y = 14 (0000 1110) |
| >> | Bitwise right shift | x >> 2 = 2 (0000 0010) |
| << | Bitwise left shift | x << 2 = 40 (0010 1000) |

For example, 2 is 10 in binary and 7 is 111.

In the table **:** Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

**Identity operators**

Identity operators : is and is not are used to check the memory locations of two objects.

- is and is not are the identity operators in Python.
- They are used to check if two values (or variables) are located on the same part of the memory.
- For list, if two variables that are equal does not imply that they are identical. It is because the interpreter locates them separately in memory although they are equal.

| Operator | Meaning | Example |
|---|---|---|
| is | True if the operands are identical (refer to the same object) | x is True |
| is not | True if the operands are not identical (do not refer to the same object) | x is not True |

Example :

```
0x1=5; y1=5
x2='Hello';y2='Hello'
x3=[1,2,3]; y3=[1,2,3]
```

```
print(x1 is not y1)
print(x2 is y2)
print(x3 is y3)
```

**Membership operators**

Membership operators : in and not in for determining the presence of items in a sequence such as strings, lists and tuples.

- in and not in are the membership operators in Python.
- They are used to test whether a value or variable is found in a sequence(string, list, tuple, set and dictionary)

| Operator | Meaning | Example |
|---|---|---|
| in | True if value/variable is found in the sequence | 5 in x |
| not in | True if value/variable is not found in the sequence | 5 not in x |

- In dictionary, we can only test for the presence of key not the value.

Example :

```
x= "Hello World"      print('H' in x)
y={1:'a',2:'b'}       print('Hello' not in x)
                      print(1 in y)
                      print('a' in y)
```

**What Is an Expression?**

- An expression is a combination of symbols or single symbol that evaluates to a value.
- A subexpression is any expression that is part of a larger expression.
- Expressions, most commonly, consist of a combination of operators and operands,

Eg : 4 + (3 * k)

**Operator precedence**

- **Precedence** : Defines the priority of an operator.
  **Associativity :**
- When an expression contains operators with equal precedence then the associativity property decides which operation is to be performed first.
- Associativity implies the direction of execution and is of two types,left to right and right to left.

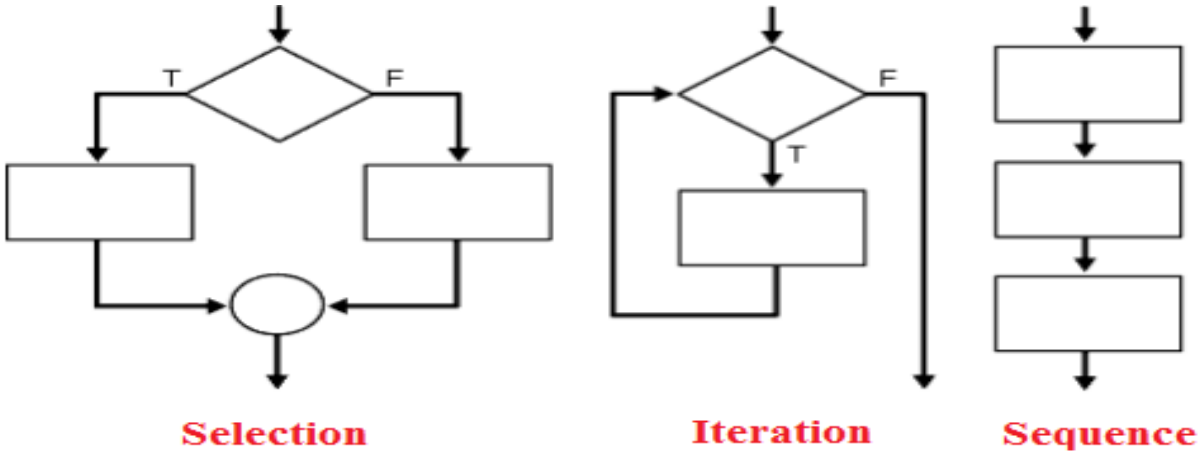
| Operator | Associativity |
|---|---|
| **(exponentiation) | Right-to-left |
| ~(negation) | Left-to-right |
| *(multi), / (div), // (truncating div), %(mod) | Left-to-right |
| +(addition), -(subtraction) | Left-to-right |

| Precedence | Operators | Associativity |
|---|---|---|
| Highest | () | Innermost to Outermost |
| | ** | Highest |
| | *,/,//,% | Left-to-right |
| lowest | +,- | Left-to-right |

**What Is a Control Structure?**

A control statement is a statement that determines the control flow of a set of instructions. A control structure is a set of instructions and the control statements controlling their execution. Three fundamental forms of control in programming are sequential, selection, and iterative control.

- Sequential control is an implicit form of control in which instructions are executed in the order that they are written.
- Selection control is provided by a control statement that selectively executes instructions.
- iterative control is provided by an iterative control statement that repeatedly executes instructions.




**Indentation in Python**

- A header in Python is a specific keyword followed by a colon.
- The set of statements following a header in Python is called a suite(commonly called a block).
- A header and its associated suite are together referred to as a clause.

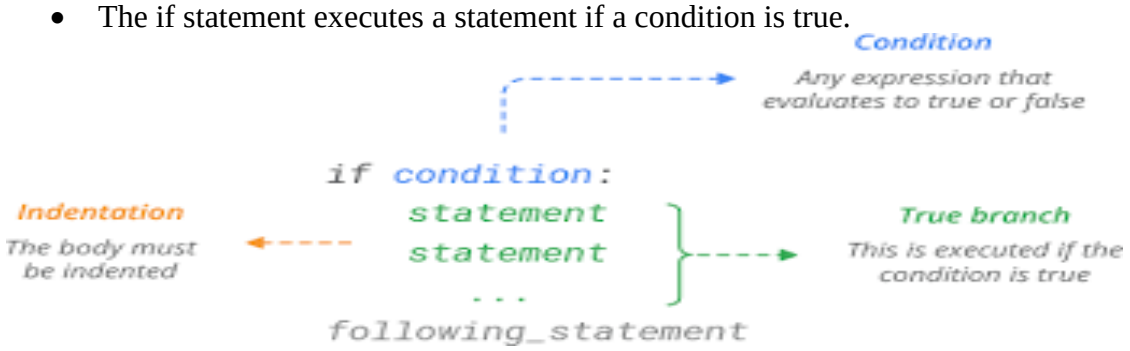
**Selection Control or Decision Making statement**

A selection control statement is a control statement providing selective execution of instructions.

- if statements
- if-else statements
- Nested if statements
- Multi-way if-elif-else statements

**if statement:**

- An if statement is a selection control statement based on the value of a given Boolean expression.
- The if statement executes a statement if a condition is true.



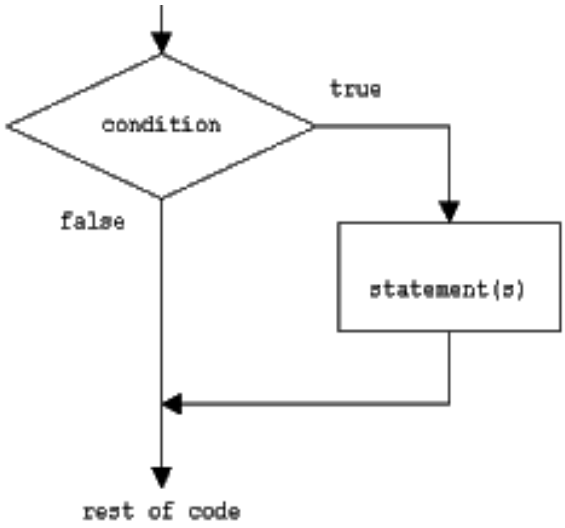
| Valid Indentation | | Invalid Indentation | |
|---|---|---|---|
| if condition:<br>  statement<br>  statement<br>else:<br>  statement<br>  statement | if condition:<br>  statement<br>  statement<br>else:<br>  statement<br>  statement | if condition:<br>  statement<br>  statement<br>else:<br>  statement<br>  statement | if condition:<br>  statement<br>  statement<br>else:<br>  statement<br>  statement |

**Details of the if Statement**

**Points to Remember**

- A colon (:) must always be followed by the condition.
- The statement(s) must be indented at least one space right of the if statement.
- In case there is more than one statement after the if condition, then each statement must be indented using the same number of spaces to avoid indentation errors.
- The statement(s) within the if block are executed if the boolean expression evaluates to true.

**Flow Chart for if statement**




**Example :**

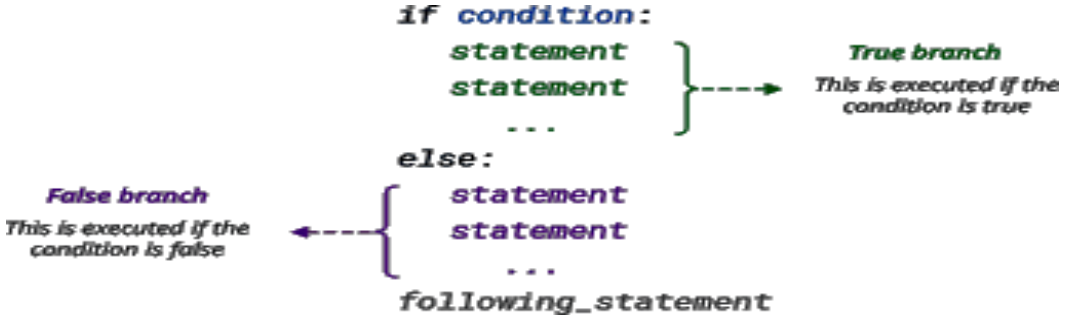
```
num1=eval(input("Enter First Number: "))
num2=eval(input("Enter Second Number: "))
if num1-num2==0:
   print("Both the numbers entered are Equal")
```

**Output :**

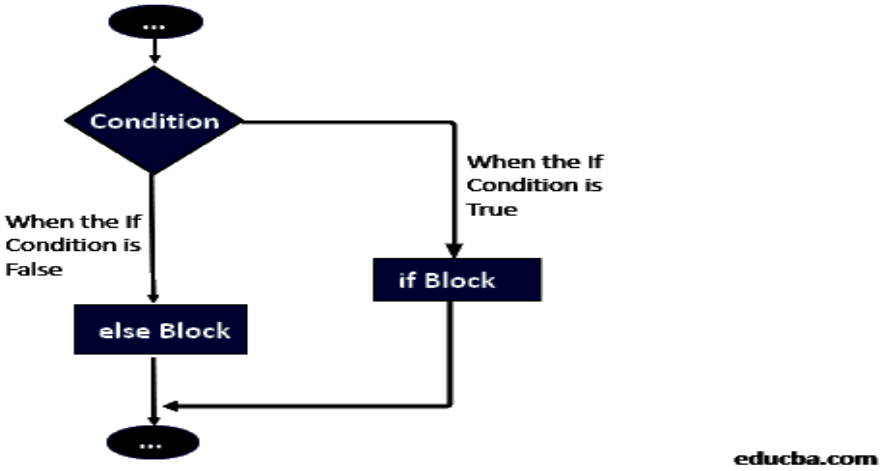
```
Enter First Number: 12
Enter Second Number: 12
Both the numbers entered are Equal
```

**if-else statements**

- The if-else statement takes care of a true as well a false condition.




**Flow Chart for if-else statement**




**Example :**

```
num1=eval(input("enter first number"))
num2=eval(input("enter second number"))
if num1>num2:
    print(num1,"is greater than ", num2)
else:
    print(num2,"is greater than ", num1)
```

**Output** :

```
enter first number  12
enter second number  45
45 is greater than  12
```

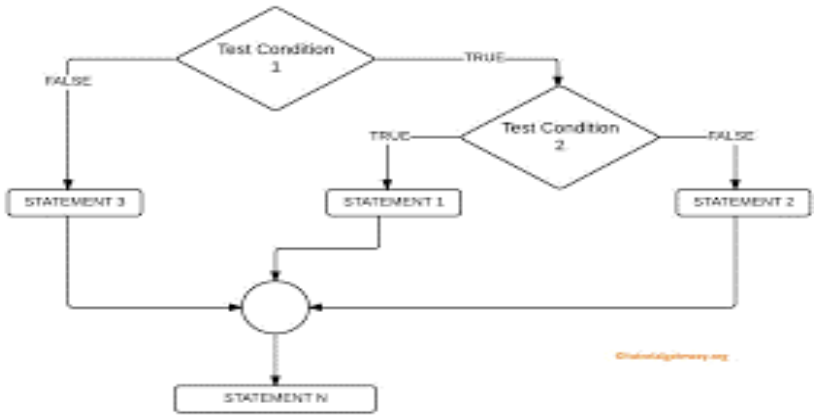
**Nested if statements**

- One if statement inside another if statement then it is called a nested if statement.

**Syntax :**

```
if Boolean-expression1:
    if Boolean-expression2:
       statement1
     else:
                statement2
  else:
          statement3
```

**Flow chart**




**Example :**

```
num=eval(input("Enter the number :"))
if num>=0:
  if num==0:
      print("Entered number ", num, " is Zero")
  else:
       print("Entered number ", num, " is positive")
else:
      print("Entered number ", num, " is negative")
```

**if...elif...else statement**

- Boolean expressions are checked from top to bottom.
- When a true condition is found, the statement associated with it is executed and the rest of the conditional statements are skipped.
- If none of the conditions are found true then the last else statement is executed.
- If all other conditions are false and if the final else statement is not present then no action takes place.

**Syntax of if...elif...else**

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

**Flow Chart:**

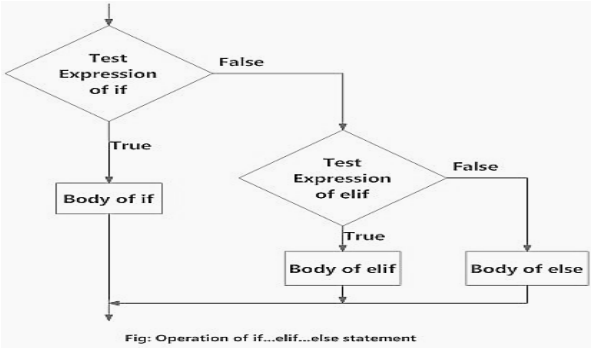



Fig: Operation of if...elif...else statement

**Example:**

```
day=int(input("Enter the day of week:"))
if day==1:
 print(" Its Monday")
elif day==2:
 print("Its Tuesday")
elif day==3:
 print("Its Wednesday")
elif day==4:
 print("Its Thursday")
elif day==5:
 print("Its Friday")
elif day==6:
 print("Its Saturday")
elif day==7:
 print(" Its Sunday")
else:
 print("Sorry!!! Week contains only 7 days")
```

**Conditional expressions**

**Syntax :**

Expression1 if condition else Expression2

**Example :**

```
num1=8; num2=9
print(num1) if num1<num2 else print(num2)
```

**Example:**

**To check whether the given year is leap year or not**

```
year=int(input("enter the year"))
if(((year%4==0)and(year%100!=0))or(year%400==0)):
  print(year,"is a leap year")
else:
  print(year,"is not a leap year")
```

**Boolean Expressions**

The Boolean data type contains two Boolean values, denoted as True and False in Python. A Boolean expression is an expression that evaluates to a Boolean value.

**Iteration statements**

- Iteration statements or loop statements allow us to execute a block of statements as long as the condition is true.
- An iterative control statement is a control statement that allows for the repeated execution of a set of statements.

**Type Of Iteration Statements**

- While Loop
- For Loop

**while Statement**

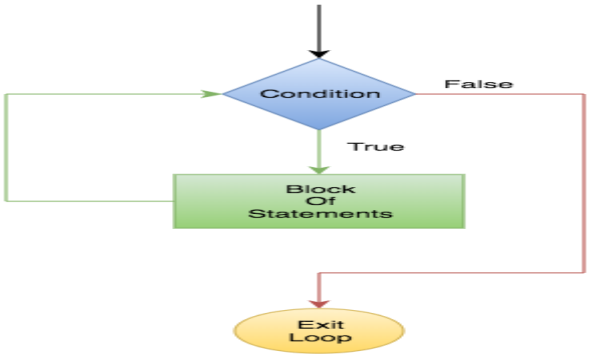
A while statement is an iterative control statement that repeatedly executes a set of statements based on a provided Boolean expression ( condition ).
Syntax :

while condition:
suite (or) statements

**Details of while statement**

- The reserved keyword while begins with the while statement.
- The test condition is a Boolean expression.
- The colon (:) must follow the test condition, i.e. the while statement be terminated with a colon (:).
- The statement(s) within the while loop will be executed till the condition is true, i.e. the condition is evaluated and if the condition is true then the body of the loop is executed.
- When the condition is false, the execution will be completed out of the loop or in other words, the control goes out of the loop.

**FLOW CHART**




**Example :**

```
x = 1
while x < 3:
  print(x)
  x = x + 1
```

OUTPUT :

1

2

**Execution**

| i) while x<3: | ii) while 2<3: | iii) while 3<3: |
|---|---|---|
| 1<3 ( condition is true ) | 2<3 ( condition is true ) | 3<3 ( condition is false ) |
| print 1 | print 2 | Exit from loop |
| x= 1+1; x=2 | x= 2+1; x=3 | |

**Python Looping Techniques**

- An infinite loop is an iterative control structure that never terminates (or eventually terminates with a system error).

**Example :**

```
while True:
  num = int(input("Enter an integer: "))
  print("The double of",num,"is",2 * num)
```

**Definite vs. Indefinite Loops**

- A definite loop is a program loop in which the number of times the loop will iterate can be determined before the loop is executed.
- A indefinite loop is a program loop in which the number of times the loop will iterate is not known before the loop is executed.

**Input Error Checking**

- The while statement is well suited for input error checking.

**Example :**

```
a=5;b=3
ch=int(input("Enter the choice"))
while ch!=1 and ch!=2:
    ch=int(input("Enter either 1 or 2"))
if ch==1:
    print(a+b)
else:
    print(a-b)
```

**range() function**

- range() built-in function is used for generating a sequence of integers that a for loop can iterate over.
- one, two or three parameters.
- The last two parameters in range() are optional.

The general form of the range function is:

range(begin, end, step)

- The 'begin' is the first beginning number in the sequence at which the list starts.
- The 'end' is the limit, i.e. the last number in the sequence.
- The 'step' is the difference between each number in the sequence.

| Example of range() function | Output |
|---|---|
| range(5) | [0,1,2,3,4] |
| range(1,5) | [1,2,3,4] |
| range(1,10,2) | [1,3,5,7,9] |
| range(5,0,-1) | [5, 4, 3, 2, 1] |
| range (-4,4) | [-4, -3, -2, -1, 0, 1, 2, 3] |
| range (-4,4,2) | [-4, -2, 0, 2] |
| range(0,1) | [0] |
| range(1,1) | Empty |
| range(0) | Empty |

**for loop**

- A for statement is an iterative control statement that iterates once for each element in a specified sequence of elements.
- Used to construct definite loops.

**Syntax :**

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

- for and in are essential keywords to iterate the sequence of values.
- var takes on each consecutive value in the sequence
- statements in the body of the loop are executed once for each value

**Flow chart**

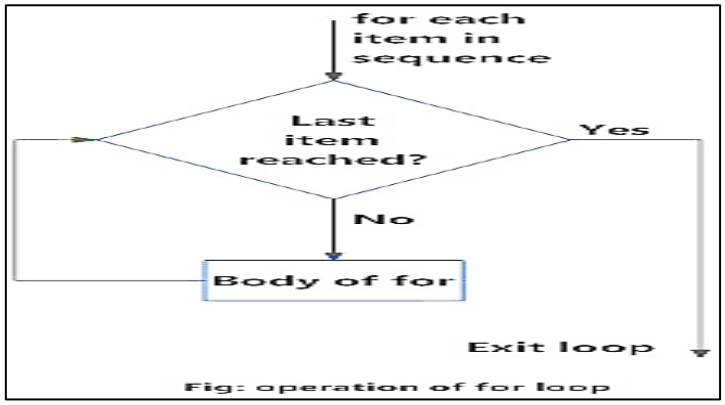



Fig: operation of for loop

**Example: for loop**

- The for loop repeats a group of statements for a specified number of times.

```
for var in range(m,n):
        print var
```

- function range(m, n) returns the sequence of integers starting from m, m+1, m+2, m+3…… n-1.

**Example :**

```
for i in range(1,6):
        print(i)
```

**Lists**

A list is a linear data structure, thus its elements have a linear ordering.

- A list in Python is a mutable linear data structure, denoted by a comma-separated list of elements within square brackets, allowing mixed-type elements.

Example :

- lst = [1,2,3,4,5]
- lst = [‘a,’b’,’c’,’d’]
- lst= [1,’ABC’,98.5,’HELLO’]

➢ Operations commonly performed on lists include retrieve, update, insert, remove, and append.

➢ A list traversal is a means of accessing, one-by-one, the elements of a list.

**Python List Type**

A list in Python is a mutable, linear data structure of variable length, allowing mixed-type elements. Mutable means that the contents of the list may be altered. Lists in Python use zerobased indexing.

All lists have index values 0 ... n-1, where n is the number of elements in the list. Lists are denoted by a comma-separated list of elements within square brackets

[1, 2, 3] ['one', 'two', 'three'] ['apples', 50, True]

An empty list is denoted by an empty pair of square brackets, [].

**lst = [10,20,30,40,50]**

lst[0]=10

lst[1]=20

lst[2]=30

lst[3]=40

lst[4]=50

- Negative index : Right to Left

lst[-1]=50

lst[-2]=40

lst[-3]=30

lst[-4]=20

lst[-5]=10

- Append ➔ Add elements at the end of the list.

Syntax : lst.append(element)

Eg : lst.append(60)

- Update ➔ Changing the elements in the list.

Eg : lst[2]=25

- Remove ➔ Delete element in the list.

Eg : del lst[3]

**Slicing ➔ Creating Sub-List**

lst[1:3]=[20,30]

lst[1:4]=[20,30,40]

lst[:4]=[10,20,30,40]

lst[2:]=[30,40,50]

**Built-in-function**

Return value but not change the list.

- listname.count(element)
- listname.index(element)
- listname.index(element,start)

Doesn't Return value but change the list.

- listname.append(element)
- listname.extend(list)
- listname.pop(index)
- listname.insert(index,element)
- listname.remove(element)
- listname.reverse()

Listname.sort()

**Assigning & Copying list**

>>> lst1=[1,2,3,4]

>>> lst2=lst1

>>> print(lst2)

[1, 2, 3, 4]

**List Comprehension**

List comprehensions in Python provide a concise means of generating a more varied set of sequences than those that can be generated by the range function.

- List of elements with squares.

**Syntax :**

lst= [ expression for variable in range ]

Eg:

```
lst = [ i**2 for i in range(1,6)]
print(lst)
```

**Example:**

```
lst1=[1,2,3,4]
lst2=[2,5,6,7]
pair=[(x,y)for x in lst1 for y in lst2 if x!=y]
print(pair)
```

**Looping in List**

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

O/P :

10

20

30

40

50

| **Operation** | **Fruit= [‘banana’, ’apple’, ’cherry’]** | |
|---|---|---|
| Replace | Fruit[2]=‘coconut’ | [‘banana’, ’apple’, ’coconut’] |

| | | |
|---|---|---|
| Delete | Del fruit[1] | [‘banana’, ’cherry’] |
| Insert | Fruit.insert(2,’pear’) | [‘banana’, ’apple’, ‘pear’, ‘cherry’] |
| Append | Fruit.append(‘peach’) | [‘banana’, ’apple’, ’cherry’, ’peach’] |
| Sort | Fruit.sort() | [‘apple’, ‘banana’, ‘cherry’] |
| Reverse | Fruit.reverse() | [‘cherry’, ‘banana’, ‘apple’] |

**Nested Lists**

lst = [[1, 2 ,3], [4, 5, 6], [7, 8, 9]]

**Tuples**

- A tuple is an immutable linear data structure denoted by parentheses(optional) which cannot be altered.

tup=()

tup1=(“apple”,”banana”,1997,2020)

tup2=(1,2,3,4,5)

tup3=“a”, ”b”, ”c”, ”d”

**Basic Operations**

**a=(10,20,20,40,50,60,10,80)** **b=(1,2,3,4,5,6)**

- Length : len(a) : 8
- Concatentation : a+b : (10, 20, 20, 40, 50, 60, 10, 80, 1, 2, 3, 4, 5, 6)

- Repetition : a*2 : (10, 20, 20, 40, 50, 60, 10, 80, 10, 20, 20, 40, 50, 60, 10, 80)
- Membership : 20 in a : True
- Maximum/Minimum : max(b) : 6 ; min(a) : 10
- Index : a.index(40) : 3
- Count : a.count(20) : 2
- Conversion to Tuple

**Converting List to Tuple using tuple() function**

```
lst=[]
for i in range(5):
    #ele=int(input())
    lst.append(i)
print(lst)
tup=tuple(lst)
print(tup)
```

**Sample program for tuple : Counting No.of ODD & EVEN numbers in tuple**

```
input_tuple=()
final_tuple=()
oddc=0
evenc=0
for i in range(1,11):
    input_tuple=(i)
    final_tuple=final_tuple+(input_tuple,)
print(final_tuple)
for x in final_tuple:
    temp=int(x)
    if temp%2==0:
        evenc=evenc+1
```

```
    else:
        oddc=oddc+1
print("Count of even numbers : ",evenc)
print("Count of Odd numbers : ",oddc)
```

# FUNCTIONS – UNIT III NOTES

**Program Routines:**

A program routine is a named group of instructions that accomplishes some task. A routine may be invoked (called) as many times as needed in a given program. A function is Python's version of a program routine.

**What is a Function routine?**

- A routine is a named group of instructions performing some task.
- A routine can be invoked ( called ) as many times as needed in a given program
- A function is Python's version of a program routine.

**Advantages of Function:**

- Helpful debugging your code.
- Reusability of code.
- Easier to understand.
- Easier to maintain.

**Function definition:**

- Every function should start with 'def' keyword.
- Every function should have name( not equal to any keyword)
- Parameters / Arguments (optional) included in b/w parenthesis.(formal)
- Every function name with/without arguments should end with(:)
- return ➔ empty / value
- Multi value return can be done ( tuples )
- Every function must be defined before it is called.

**Syntax:**

```
def function_name(arg1, arg2, …. arg n) :
                    program statement 1
                    program statement 2
                    --------
                    --------
                    return
```

**Function call:**

- Function name
- Arguments / Parameters(actual)

} Equal to the function definition

**Actual arguments & Formal parameters:**

- Actual arguments, or simply "arguments," are the values passed to functions to be operated on.
- Formal parameters, or simply "parameters," are the "placeholder" names for the arguments passed.

**Value-returning functions:**

- A value-returning function in Python is a program routine called for its return value, and is therefore similar to a mathematical function.

**Non-value-returning functions:**

- A non-value-returning function is a function called for its side effects, and not for a returned function value.

**The local and global scope of a variable:**

- Variables and parameters that are initialised within a function including parameters, are said to exist in that function's local scope. Variables that exist in local scope are called local variables.
- Variables that are assigned outside functions are said to exist in global scope. Therefore, variables that exist in global scope are called global variables.

**LOCAL VARIABLE :**

- A local variable is a variable that is only accessible from within the function it resides. Such variables are said to have local scope.

**GLOBAL VARIABLE :**

- A global variable is a variable defined outside of any function definition. Such variables are said to have global scope. The use of global variables is considered bad programming practice.

```
# Global scope
x = 0

def outer():
    # Enclosed scope
    x = 1
    def inner():
        # Local scope
        x = 2
```

**Local and global variables with the same name**

- Local Variables Cannot be Used in Global Scope
- Accessing a local variable outside the scope will cause an error.

**return statement**

- The return statement is used to return a value from the function.
- It is also used to return from a function, i.e. break out of the function.

Eg:

```
def minimum(a,b):
  if a<b:
    return a
 elif b<a:
   return b
 else:
   return "Both the numbers are equal"
print(minimum(100,85))
```

Output:

8 is minimum

**Returning multi-values**

```
def compute(num1):
        print("Number = ",num1)
        return num1*num1, num1*num1*num1
square,cube=compute(4)
print("Square = ",square,"Cube = ",cube)
```

**Call by value:**

- A copy of actual argument is passed to respective formal arguments.
- Any changes made to the formal arguments will not be visible outside the scope of the function.

**Call by reference:**

- Location of  of actual argument is passed to respective formal arguments.
- Any changes made to the formal arguments will also reflect in actual arguments.

**Built-in-function:**

- Mathematical Functions
- **ceil()** :-Returns the smallest integral value greater than the number. If number is already integer, same number is returned.

- **floor()** :- Returns the greatest integral value smaller than the number. If number is already integer, same number is returned.
- **fabs()** :- Returns the absolute value of the number.
- **gcd()** :- Used to compute the greatest common divisor of 2 numbers mentioned in its arguments

**Recursive function:**

- **A function is said to be recursive if a statement within the body of the function calls itself.**
- **Eg:**

```
def factorial(n):
        if n==0:
                return 1
return n*factorial(n-1)
print(factorial(5))
```

**Advantages:**

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

**Dis-Advantages:**

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

**Types of Arguments:**

- Positional Arguments
- Keyword Arguments
- Default Arguments
- Variable length Arguments

**Positional Arguments:**

- Number of arguments should be same in both function call and function definition.
- A positional argument is an argument that is assigned to a particular parameter based on its position in the argument list.
- Order or Position should be followed.

Eg:

```
def display(a,b):   #function definition
    print(a,b)
display(10,20)      #function call
```

**Keyword Arguments:**

- A keyword argument is an argument that is specified by parameter name.
- Order or Position should not be followed.
- Initialisation will be done based on the keyword. (name of identifier)

Eg:

```
def display(a,b):
  print(a,b)
display(b=10,a=20)
```

**Default Arguments:**

- An argument that can be optionally provided in a given function call. When not provided, the corresponding parameter provides a default value.
- Number of arguments need not be same in both function call and function definition.
- Some of arguments will be consider as default arguments.

Eg:

```
def display(a,b,c=30):
    print(a,b,c)
display(10,20)
```

**Variable Length Arguments:**

- Arbitrary number of arguments.
- By placing * as prefix to the argument of function definition.

Eg:

```
def display(*courses):
    for i in courses:
        print(i)
display("BCA","MCA","CS","IT")
```

# OOP'S CONCEPTS

**Main Concepts of Object-Oriented Programming (OOPs)**

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance

# Class

A class is a collection of objects.

A class contains the blueprints or the prototype from which the objects are being created.

It is a logical entity that contains some attributes and methods.

**Some points on Python class:**

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator.
- Eg.: Myclass.Myattribute

# Syntax

```
class ClassName:
    # Statement-1
    .
    .
    .
    # Statement-N
```

# Objects

The object is an entity that has a state and behavior associated with it.

**An object consists of :**

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

# The self

1. Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method that takes no arguments, then we still have to have one argument.
3. This is similar to this pointer in C++ and this reference in Java.

## Example: How to add two numbers using class and object in Python

```
class Test:

    def findsum(self, a, b):
        s = a + b
        return s


a = int(input("Enter first number:"))
b = int(input("Enter second number:"))

obj = Test()
s = obj.findsum(a, b)

print("Sum is:", s)
```

**Output:**

```
Enter first number:10
Enter second number:20
Sum is: 30
```

# __init__ method

- The __init__ method is similar to constructors in C++ and Java.
- Constructors are used to initializing the object's state.
- Like methods, a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation.
- It runs as soon as an object of a class is instantiated.
- The method is useful to do any initialization you want to do with your object.

# CODING

```
class stud:
    def __init__(self,rno,name):
        self.rno=rno
        self.name=name
    def display(self):
        print(self.rno)
        print(self.name)

obj=stud(101,"ABC")
obj.display()
```

# OUTPUT

```
Python 3.10.1 (tags/v3.10.1:2cd268a, Dec
2
Type "help", "copyright", "credits" or "licen

== RESTART: C:/Users/Selvakumar/AppD
101
ABC
```

# Polymorphism

```
class sample:
    def display(self,a=None,b=None):
        print(a)
        print(b)

obj=sample()
obj.display(5)
obj.display(3,4)
```

```
Python 3.10.1 (tags/v3.10.1
2
Type "help", "copyright", "c
>>>
== RESTART: C:/Users/Se
5
None
3
4
>>>
```

# Inheritance

```
lass Person( object ):                  # parent class
                        # __init__ is known as the constructor
                        def __init__(self, name, idnumber):
                                                self.name = name
                                                self.idnumber = idnumber
 child class
lass Employee( Person ):
                        def __init__(self, name, idnumber, salary, post):
                                                self.salary = salary
                                                self.post = post
                                                # invoking the __init__ of the parent class
                                                Person.__init__(self, name, idnumber)
                                                print(self.name)
                                                print(self.idnumber)
                                                print(self.salary)
                                                print(self.post)
 creation of an object variable or an instance
bj= Employee('Rahul', 886012, 200000, "Engineer")
```

```
Python 3.10.1 (tags/v3.1
2
Type "help", "copyright"

== RESTART: C:/Users
Rahul
886012
200000
Engineer
```