NASA

4 | 11 | 25

Python Keywords

Python Keywords define the rules and Structure of python programs. Python Keywords cannot be used them as the names of variables, functions, classes or any other identifiers

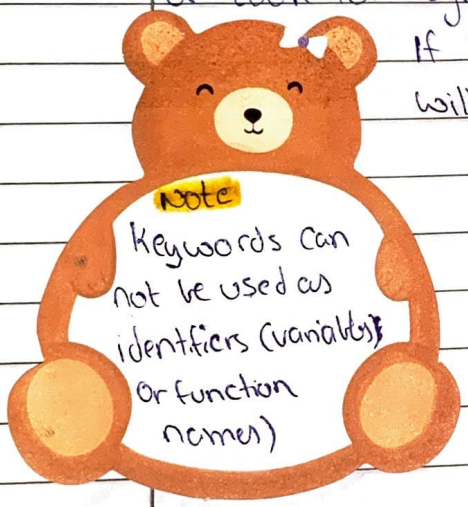
We can also get all get the keyword names using the below :

```
import keyword
# Printing all keywords at once using "kwlist()
print("The list of keywords is : ")
Print (keyword.kwlist)
```

We can identify Python Keywords either with Syntax Highlighting or Look for Syntax Error

If we attempt to use a keyword as variable, Python will raise a Syntax Error

```
for = 10  → will throw an error
print (for)
```



Category	Keywords
value keywords	True, False, None
Operator Keywords	and, or, not, is in
control flow keywords	if, else, elif, for, while, break, continue, pass, try, except, finally, raise, assert,
Function and class	def, return, lambda, yield, class
context management	with, as
Import and module	import, from
Scope and namespace	global, nonlocal
Async Programming	async, await.

Python print() function

print() function is used to print a python object(s) in python as standard output

Syntax → `print(object(s), sep, end, file, flush)`

```
# Sample python objects
list = [1, 2, 3]
tuple = ("A", "B")
String = "Geeks for Geeks"
# printing the object
print(list, tuple, string)
```

```
# Sample python object
list = [1, 2, 3]
tuple = ("A", "B")
String = "Geeks for Geeks"
# printing the Object
print(list, tuple, string, sep = "<<..>>")
```

```
# Sample python object
list = [1, 2, 3]
tuple = ("A", "B")
String = "Geeks for Geeks"
# Printing the objects
print(list, tuple, String, sep = "<<..>>")
```

We use open() function for opening exertnal file

```
# open and read the file
my_file = open("geeks for geek.txt", "r")
# print the contents of the file
print(my_file.read())
```

We use read() function to read the file

```
# Python code for Python to Stderr
# importing the package
# for sys.stderr
import sys
# variables
Company = "Geeks for Geeks.org"
Location = India
Email = "contact@geeksforgeeks.org"
# print to stderr
print(Company, Location, Email, file=sys.stderr)
```

Python output formatting

Output formatting refers to the way data is presented when printed or logged. Proper formatting makes information more understandable and actionable.

The String modulo operator (%) is one of the oldest ways to format strings in python. It allows you to embed values within a string by placing format specifiers in the string. Each specifier starts with a % and ends with a character that represents the type of value being formatted.

%d → decimal
%o → octal
%f → exponential

```
# String modulo Operator (%)
print ("Geeks: %.2d, Portal: %5.2f" %(1, 05.333))
print ("Total Students: %3d, Boys: %.2d" %(240, 120))
# print integer value
print ("%.7.3o" %(25))  # print octal value
print ("%.10.3E" %(356.08977)) # print exponential value
```

The format() method allows for a more flexible way to handle string interpolation by using curly braces {} as place holders for substituting values into a string. It supports both positional and named arguments, making it versatile for various formatting needs.

```
# Positional formatting with format() method
#Using indexed placeholders for string formatting
print("I love {0} for \"{1}!\".format("
Geeks", "Geeks"))
#{0} is replaced by the 1st argument "Geeks"
 Print("{0} and Portal".format("Geeks"))
# formatting with placeholder {0} replaced by'Geek
print("Portal and {0}".format("Geeks"))
```

The format() method also supports detailed formatting such as setting widths, alignment, no formatting and more which can be useful for creating tabulated data/report.

```
#Advanced formatting with positional and
 named arguments
# Mixing positional and named arguments
template = "Number one portal {0}, {1}, and
            {other}."
print (template.format("Geeks"for", other="Gee")
# Format integers and floats with specified
 width and precision
 print ("Geeks : { 0:2d}, Portal : {1:8.2f}".
    fo          format(12, 0.5534))
#Demonstrate order swapping and formatting
 Precision
 print("Second argument: {1:3d}, 1st one:
            {0:8.2f}".format(47.42, 11))
#Using named arguments for clarity in
 Complex formats
 print("Geeks : {a:5d}, Portal: {p:8.2f}".format
  (a=453, p=59.058))
```

Python string methods such as str.center(), str.ljust(), and str.rjust() provide straightforward ways to format strings by aligning them within a specified width.

```
cstr = "I love geekfor geeks"
# Printing the center aligned string with fill char
print ("center aligned: ")
print (cstr.center(40, '#'))
# printing the left aligned string with "-"
print ("left aligned: ")
print (cstr.ljust(40, '-'))
# printing the right aligned string with "-"
print ("right aligned: ")
print (cstr.rjust(40, '-'))
```

d → Decimal Integer
b → binary format
o → octal format
u → obsolete and equivalent to 'd'
x or X → Hexadecimal format
e or E → exponential notation
f or F → Floating - point decimal
g or G → General format
c → single character
r → string format (repr())
s → string format (str())
% → percentage.

10/11/25

Python input() function

The input() function in Python is used to take input from the user

By default, input() always returns data as a string. In order to have other types like integer (int()) you to use typecast

Syntax → input(prompt)

```
a = input("what color is the rose?")
print("The rose is", a)
```

```
n = int(input("How many roses? "))
print("Total no of roses: ", n)
```

```
p = float(input("price of each rose? "))
print("price entered: ", p)
```

You can take multiple values in a single line, seperated by spaces. Python allows splitting them using .split()

```
x,y = input("Enter nos: ").split()
print("First no: ", x)
print("2nd no: ", y)
```

```
name = input("Name: ")
age = input("Age: ")
print("Name and age: %s %s" % (name, age))
```

```
n1 = int(input("No 1: "))
n2 = int(input("No 2: "))
s = n1 + n2
print("sum", s)
```

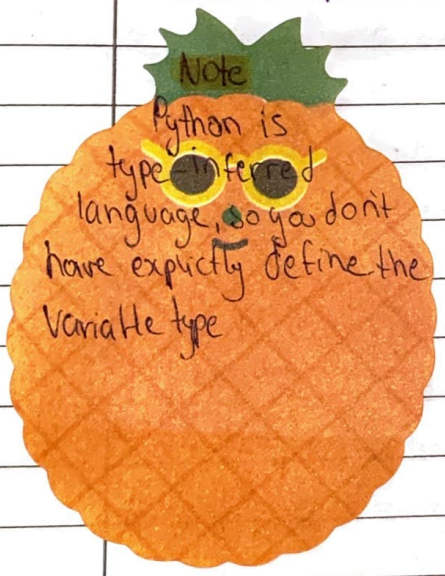
```
a = list (input ("List 1: ")
b = list (input("List 2: ")
for i in b:
    a. append (i)
print ("Final list: ", a)
```

Python Variables and Literals

A variable is a container (storage ~~data~~ area) to hold data

```
number =10
```

```
# assign value to site_name variable
Site_name = 'programiz.pro'
print (site_name)
# output :- programiz.pro
```




```
Site_name = 'programiz.pro'
print (site_name)
# assigning a new value to site name
Site_name = 'apple.com
print (site_name)
```

```
a, b, c = 5, 3.2, 'Hello'
print(a)
Print (b)
Print (c)
```

```
Site1 = site2 = 'Programiz.pro'
Print(x) # x here is site1
Print (y) # y here is site2
```

Literals are representation of fixed values in a program.
Literals are often used to assign to variable or constant

```
Site_name = 'programiz.pro'
```

↳ variable ↳ literal

Numeric literals are immutable (unchangable). Numeric literal can be long to 3 different types: Intgers, float, and Complex

Numeric Literals

- **Integer literal** → Nos without decimal part → can be +ve or -ve
- **Floating point literal** → Nos with decimal parts → can be +ve or -ve
- **Complex literal** → represent complex numbers

Texts wrapped inside quotation marks are called string literals. We can also use single quots to create Strings.

Python Operators

Operators are used to perform operations on variables and values

Operators: Special Symbols like -, +, *, /, etc

Operands: value on which operator is applied.

Floor division // is used for rounding off the result.

Operator	Operand Type
+, -, *, /, %, //, **	Arithmetic Operator
<, <=, >, >=, ==, !=	Relational operator
AND, OR, NOT	Logical operator
&, |, <<, >>, ~, ^	Bitwise operators
=, +=, -=, *=, %=	Assignment operator

Python Arithmetic operators are used to perform basic mathematical operations like addition, subtraction, multiplication and division

Relational operators compares values. It is either returns True or False according to the condition

```
# Arithmetic operators
# variables
a = 15
b = 4
```

NASA

```
#addition
print(f"addition: ", a+b)
# subtraction
print ("subtraction: ", a-b)
# Multiplication
Print ("Multiplication: ", a*b)
# Division
Print ("Division: ", a/b)
# Floor Division
print ("Floor Division: ", a//b)
# Modulus
print ("Modulus: ", a%b)
# Exponentiation
print ("Exponentiation: ", a**b)
```

```
# Relational operators   = => Assignment operator
a = 13                   == => checks equality.
b = 33
print (a > b)
print (a < b)
print (a == b)
print (a != b)
print (a >= b)
print (a <= b)
```

Logical operators perform Logical AND, Logical OR and Logical NOT operations. It is used to combine conditional statement.

```
a = True
b = False
print (a and b)
Print (a or b)
Print (not a)
```

Bitwise operators act on bits and perform bit-by-bit operations. These are used to operate on binary numbers.

```
a = 10
b = 4
print (a & b)
print (a | b)
print (~a)
print (a ^ b)
print (a >> 2)
print (a << 2)
```

Assignment operators are used to assign values to the variables

```
a = 10
b = a
print (b)
b += a
print (b)
b -= a
print (b)
b *= a
print (b)
b <<= a
print (b)
```

is and is not are identifier operators both are used to check if two values are located on the same part of memory.

```
a = 10
b = 20
c = a
Print (a is not b)
print (a is c)
```

in and in not are membership operaton that are 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.

```
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")
```

Ternary operators also known as conditional expressions are operators that evaluate something based on a condition being True or false

Syntax → [on_true] if [expression] else [on_false]

```
a, b = 10, 20
min = a if a < b else b
print min
```

Operator precedence and associativity determine the properties of the operator

```
Expr = 10 + 20 * 30
print(expr)
name = Alex
age = 0
if name == "Alex" or name == "Jhon" and age >= 2:
    Print("Hello! Welcome")
else:
    Print("GoodBye!")
```

```
print(100/10*10)
print(5-2+3)
print(5-(2+3))
print(2**3**2)
```

Python if...else

An if statement executes a block of statement executes a block of code only when specified condition is satisfied.

Syntax → if condition:
    # body of if statement

```
num = int(input("Enter a number"))
# Checks if number is greater than 0
if num > 0:
    print(f"{num} is a positive no.")
print("A statement outside the if statement")
```

Python uses indentation to define a block of code, such as the body of an if

```
x = 1
total = 0
# start of the if statement
if x != 0:
    total += x
    print(total)
# end of if statement
print("This is always executed.")
```

An if statement can have an optional else clause. The else statement executes if the condition in the if statement evaluates false

Syntax → if condition:
    # body of if statement
else:
    # body of else statement

```
number = int(input('Enter a number'))
if number > 0:
    Print('Positive number')
else:
    Print('Not a positive number')
Print('This statement always executes')
```

The if...else is used execute a block of code among alternatives. For multiple conditions we use if...elif...else

Syntax → if condition1:
    # code block 1
elif condition2:
    # code block2
else
    # code block 3

```
number = -5
if number > 0:
    Print('Positive number')
elif number < 0:
    print('negative number')
else:
    Print('zero')
Print('This statement is always executed')
```

It is possible to include an if statement inside and ~~this~~ if statement

```
number = 5
# outer if statement
if number >= 0:
    # inner if statement
    if number == 0:
        Print('Number is 0')
    # inner else statement
    else:
        Print('number is +ve')
else:
    Print('No is -ve')
```

NASA

27 | 11 | 25

Python Range()

The Python range() function returns a sequence of numbers in a given range.

```
for i in range(5):
    print(i, end=" ")   [optional]
Print()
```

Syntax → range(start, stop, step)

- start: Optional, start value of Sequence
- stop: Next value after the end value of the sequence
- step: difference b/w two nos of sequence

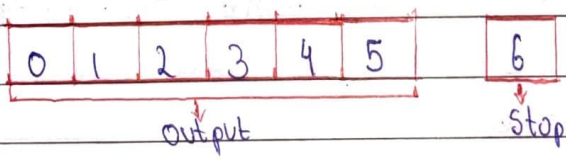
Python range() function takes can be initialised in 3 ways

→ range(stop) takes one argument

→ range(start, stop) take 2 argument

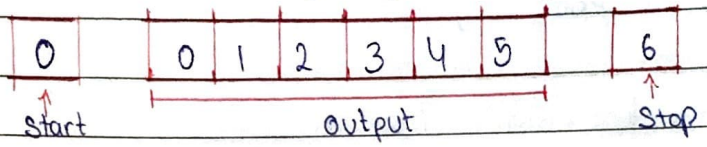
→ range(start, stop, step) takes 3 arguments

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



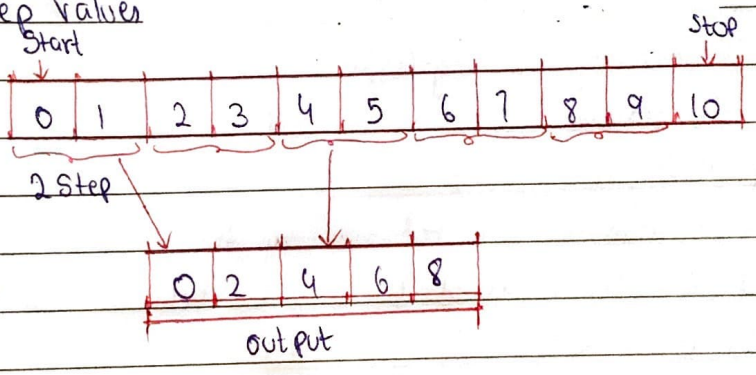

```
# printing first 6
# whole number
for i in range(6):
    print(i, end = " ")
Print()
```

When the user calls range() with 2 arguments, the user gets to decide not only where the series of numbers stops but also where it starts, so the user doesn'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).




```
# Printing a natural
# number from 5 to 20
for i in range(5, 20):
    Print(i, end=" ").
```

Calling range() with three arguments lets you set start, stop, and step values




```
for i in range(0, 10, 2):
    Print(i, end=" ")
print()
```

If a user wants to increament, then the user needs step to be +ve number

```
# increament by 4
for i in range(0, 30, 4):
    print(i, end=" ")
Print()
```

If a user wants to decrement, then the user needs steps to be negative number.

```
# increament by -2
for i in range(25, 2, -2):
    print(i, end=" ")
print()
```

Python range() function doesn't support float numbers. i.e user cannot use floating point or non-integer in any of its arguments users can only integer numbers

The result from two range() functions can be concatenated by using the Chain() method. The chain() method is used to print all the values in iterable targets one after another mentioned in arguments

```
from itertools import chain
# Using Chain method
print("concaten ation the result")
res = Chain(range(5), range(10,20,2))
for i in res:
    print(i, end=" ")
```

A sequence of numbers is returned by the range() function as it object that can be accessed by its index value. Both +ve and -ve indexing is supported by its object.

```
ele = range(10)[0]
Print("First element: ", ele)
ele = range(10)[-1]
print("\n Last element: ", ele)
ele = range(10)[4]
Print("\n Fifth element: ", ele)
```

```
fruits = ["apple", "banana", "cherry", "date"]
for i in range(len(fruits))
    print(fruits[i])
```

NASA

28 | 11 | 25

Python for loop

In Python, we use a for loop to iterate over sequences such as lists, Strings, dictionaries, etc.

```
languages = ['Swift', 'Python', 'Go']
# access elements of the list one by one
for lang in languages:
        print(lang)
```

Syntax → for val in sequence:
        # run the code

In python we use indentation to define a block of code, such as the body of the loop

```
languages = ['Swift', 'Python', 'Go']
# start of the loop
for lang in languages:
        print(lang)
        print('------')
# end of the for loop
print('last statement')
```

If we iterate through a string, we get individual characters of the string one by one.

```
language = 'Python'
# iterate over each character in language
for x in language:
    print(x)
```

In Python, the range() function returns a sequence of numbers.

```
# iterate from i=0 to i=3
for i in range(0,4):
    print(i)
```

The break and continue statements are used to alter the flow of loops.
The break statement terminate the for loop immediately before it loops through all items.

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

The continue statement skips the current iteration of the loop and continues with next iteration

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

A loop can also contain another loop another loop inside it. These loops are called nested loops. In nested loops, the inner loop is executed for each iteration of the outer loop.

```
#outer loop
attributes = ['Electric', 'Fast']
cars = ['Tesla', 'Porsche', 'Mercedes']
for attribute in attributes:
    for cars in cars:
        print(attribute, car)
    # this statement is outside the inner loop
    print("------")
```

If we dont intend to use items of sequence inside the body of a loop, it is clearer to use the (_) underscore as the loop variable.

```
# iterate from i = 0 to 3
for _ in range(0, 4):
    Print('Hi')
```

Python While loop

while loop is used to repeat a block of code until a certain condition is met

```
number = 1
While number <= 3:
    Print(number)
    number = number + 1
```

Syntax → While condition
    # body While loop

```
#Print numbers until the user enters 0
number = int(input("Enter a number: "))
# iterate until the user enters 0
While numb != 0
    Print(f'You entered {number}')
    number = int(input('Enter a number: '))
print('the end')
```

If the condition of a while loop always evaluates to True, the loop runs continously, forming infinite while loop.

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

NASA

4 | 12 | 2025

Python Break and Continue

The break and continue statements are used to alter the flow of loops

Break exits the loop entirely

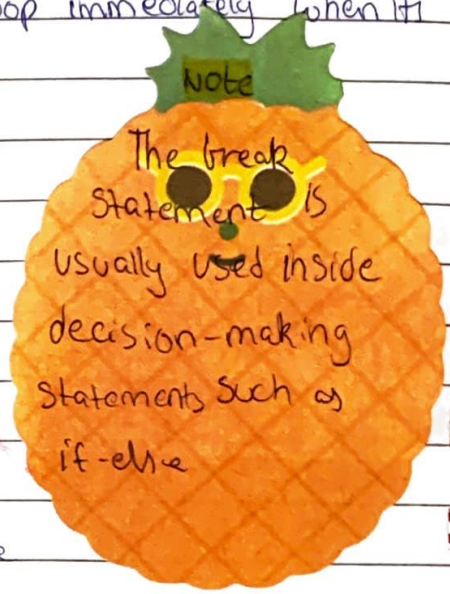
Continue skips the current iteration and proceeds to the next one

The break statement terminates the loop immediately when it's encountered.

Syntax → break

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

We can also terminate the while loop using a break statement.



The continue statement skips the current iteration of loop and control flow of the program goes to the next iteration.

Syntax → continue

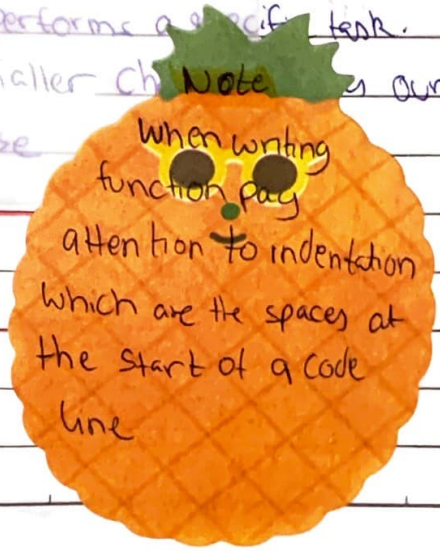
```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

We can also the continue Statement with a while loop

Python Functions

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

Dividing a complex problem into smaller ch[illegible]s our program easy to understand and reuse



used to create a function

```
def greet():
    print("Hello, World!")
```

greet → name of function

function body

Creating a function doesn't mean we are executing the code inside it. It means the code is there for us to use if we want to. To use this function, we need to call the function.

```
def greet():
    print("Hello, world!")
# Call the function
greet().
print("outside function")
```

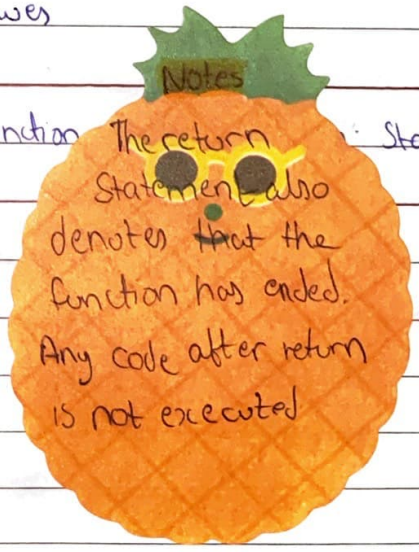
Arguments are inputs given to the function

```
def greet (name)
    print("Hello," name)
#pass argument
greet ("john")
```

```
#function with two arguments
def add_numbers(num1, num2):
    sum = num1 + num2
    print ("Sum: ", sum)
# function call with 2 values
add_numbers(5,4)
```

We return a value from the function : Statement

```
# function defination
def find_square(num):
    result = num * num
    return result
# function call
Square = find_square(3)
Print(Square:', square)
```



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

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

Python provides some built-in functions that can directly used in our program. we don't need to create the function, we just need to call them.

Some Python library functions are:

① print()
② sqrt() - returns square root of a number
③ pow() - return the power of number

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

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

Python Object Oriented Programming

Python is a versatile programming language that supports various programming styles, including object-oriented (OOP) through the use of objects and classes.

An object is any entity that has attributes and behaviours.

attributes - names, age, color etc
behaviour - dacing, singing, etc

Similarly, a class is a live prob for the object

```
Class Parrot:
    #class attribute
    name = ""
    age = 0
# create parrot1 object
Parrot1 = Parrot()
Parrot1.name = "Blu"
Parrot1.age = 10
# create another object parrot2
parrot2 = Parrot()
Parrot2.name = "Woo"
Parrot2.age = 15
# access attributes
Print(f"{parrot1.name} is {parrot1.age} years old")
Print(f"{parrot2.name} is {parrot2.age} years old")
```

Inheritance is a way of creating a new class for using details of an existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class)

```
# base class
class Animal:
    def eat(self):
        print("I can eat!")
    def sleep(self):
        print("I can sleep!")
# derived
Class Dog(Animal):
    def bark(self):
        Print("I can bark! Woof! Woof!")
# Create object of the Dog class
dog1 = Dog()
```

```
# calling members of the base class
dog1.eat()
dog1.sleep()
# calling members of the derived class
dog1.bark()
```

Encapsulation is one of the key features of oop. Encapsulation refers to the bundling of attributes and methods inside a single class. It prevents outer classes from accessing and changing attributes and methods of class. This also helps to achieve data hiding. In python, we denote private attributes using underscore as the prefix i.e. single _ or double __

```
class computer:
    def __init(self):
        self.__maxprice = 900
    def sell(self):
        print("Selling Price: {}".format(self.__maxprice))
    def setmaxPrice(self, price):
        self.__maxprice = price
c = computer()
c.sell()
# change the price
c.__maxprice = 1000
c.sell()
#using setter function
c.setMaxPrice(1000)
c.sell
```

Polymorphism simply means more than one form. That is, same entity can perform different operations in different scenarios

```
Class Polygon:
    # Method to render a shape
    def render(self):
        print("Rendering polygon...")
class Square(Polygon):
    # renders square
    def render(self):
        print("Rendering Square...")
class Circle(Polygon):
    # renders circle
    def render(self):
        print("Rendering circle...")
# Create an object of Square
S1 = Square()
S1.render()
# Create an Object of Circle
C1 = Circle()
C1.render()
```

Pattern Programs

Pyramid Pattern are sequences of characters or numbers arrang-ed in a way that resembles a pyramid, with each level having one more element than the level above.

```
# Function to print full pyramid pattern
def full_pyramid(n):
    for i in range(1, n+1):
        # Print leading spaces
        for j in range(n-i):
            print(" ", end=" ")
        # Print asterisks for the current row
        for k in range(1, 2*i):
            print("*", end=" ")
        print()
full_pyramid(5)
```

A full pyramid pattern is a series of lines that form a pyramid-like structure. Each line contains a specific no of characters, and the no of characters on each line increases symmetrically as we move down the pyramid.

```
def print_space(space):
    if space > 0:
        print(" ", end = " ")
        print_space(space - 1)
def print_star(star):
    if star > 0:
        print("*", end = " ")
        print_star(star - 1)
def print_pyramid(n, current_row = 1):
    if current_row > n:
        return
    spaces = n - current_row
    stars  = 2 * current_row - 1
    # Print spaces
    print_space(spaces)
    # Print stars
    print_star(stars)
    # Move to the next line for the next row
    print()
    # Print the pyramid for the next row
    print_pyramid(n, current_row + 1)
# Set the number of rows of pyramid
n = int(input("Enter the no of rows"))
# print the pyramid pattern
print_pyramid(n)
```

```
n=5
alph=65
for i in range(0,n):
    print(" "*(n-i), end=" ")
    for j in range(0,i+1)
        print(chr(alph), end=" ")
        alph+=1
    alph=65
    print()
```

```
def print_number_pyramid(rows):
    for i in range(1, rows+1):
        # Print spaces
        for j in range(rows-i):
            print(" ", end=" ")
        # Print numbers
        for j in range(2*i-1):
            print(j+1, end=" ")
        # Move to the next line after each row
        print()
# Example usage
num_rows = int(input("Enter number of rows: "))
print_number_pyramid(num_rows)
```

```
# Function to print inverted full pyramid pattern
def inverted_full_pattern(n):
    # outer loop for the number of rows
    for i in range(n, 0, -1):
        # Inner loop for leading spaces in each row
        for j in range(n - i):
            print(" ", end = " ")
        # Inner loop for printing asterisks in each
        row
        for k in range(2 * i - 1):
            print ("*", end = " ")
        # Move to next line after each row
        print(" ")
# Set the value of n (no of rows)
n = int (input ("Enter no of rows: "))
# Call the function to print the inverted full
pyramid
inverted_full_pyramid(n)
```

```
def hallow_pyramid(n):
    for i in range(1, n+1):
        for j in range(1, 2*n):
            if j == n - i + 1 or j == n + i - 1 or
            i == n:
                print("*", end = " ")
            else:
                print(" ", end = " ")
        print()
# Set the number of rows for the pyramid
rows = int(input("Enter the no of rows: "))
# Call the function with the specified number of rows
hallow_pyramid(rows)
```

The half pyramid starts with one asterik in the 1st raw and adds one more asterisk in each subsequent raw. The print("\n") statement is used to move to the next line after printing each row.

```
# Function to print a half pyramid pattern
def half_pyramid(n):
    for i in range(1, n+1):
        for j in range(1, i+1):
            print("* ", end = "")
        print()
# Example : print a half pyramid
n = int(input("Enter no of rows: "))
half_pyramid(n)
```

```
def print_half_pyramid(n):
    if n>0:
        # call the function recursively with a smaller value
        of n
        print_half_pyramid(n-1)
    # Print '*' characters for the current raw
    print('*' * n)
# Get the number of rows from the user
rows = int(input("Enter the number of rows: "))
# call the function to print the half pyramid pattern
print_half_pyramid(rows)
```

```
# function to demonstrate printing pattern of
numbers
def numpat(n):
    # initializing starting number
    num = 1
    # outer loop to handle number of rows
    for i in range(0, n):
        # re assigning num
        num = 1
        # inner loop to handle no of columns
            # values changing acc to outer loop
        for j in range(0, i+1):
            # printing number
            print(num, end = " ")
            # incrementing num at each column
            num = num+1
        # ending line after each row
        print(" ")
# Driver code
n = int(input("Enter no of rows: "))
numpat(n)
```

```
# Function to demonstrate printing pattern of
alphabets
def alphabet(n):
    # initializing value corresponding to 'A'
      ASCII value
    num = 65
    # outer loop to handle number of rows 5 in this
      case
    for i in range(0, n):
        # inner loop to handle number of columns
        # values changing acc to outer loop
        for j in range(0, i+1):
            # explicity converting to char
            ch = chr(num)
            # printing char value
            print(ch, end=" ")
        # increment number
        num = num+1
        # ending line after row
        print("\r")
# Drivers code
n=5
alphabet(n)
```

```
# Function to print inverted half pyramid pattern
def inverted_half_pyramid(n):
    for i in range(n, 0, -1):
        for j in range(1, i+1):
            print("* ", end="")
        print("\r")
# Example : Inverted half pyramid
n= int(input("Enter the no of rows: "))
inverted_half_pyramid(n)
```

```
def print_hallow_inverted_half_pyramid(rows):
    for i in range(rows, 0, -1):
        for j in range(rows - i):
            print(" ", end = " ")
        for j in range(i):
            if j == 0 or j == i-1 or i == rows:
                print("*", end=" ")
            else:
                print(" ", end = " ")
        print()
# Example usage
num_rows = 5
print("Hallow Inverted Half pyramid: ")
print_hallow_inverted_half_pyramid(num_rows)
```