Python and It's History

- **Python** is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990.

- In late 1980s, Guido Van Rossum was working on the Amoeba distributed operating system group. He wanted to use an interpreted language like ABC (ABC has simple easy-to-understand syntax) that could access the Amoeba system calls. So, he decided to create a language that was extensible. This led to design of a new language which was later named Python.

- Python wasn't named after a dangerous snake. Rossum was fan of a comedy series from late seventies. The name "Python" was adopted from the same series "Monty Python's Flying Circus".

  Python is Interpreted − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.

  Python is Interactive − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs. Interactive mode is a command line shell which gives immediate feedback for each statement.

  Python is Object-Oriented − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.

Features of Python

1. **Simple**: Python is a simple and minimalistic language. Reading a good Python program feels almost like reading English, although very strict English! This pseudo-code nature of Python is one of its greatest strengths. It allows you to concentrate on the solution to the problem rather than the language itself.

2. **Easy to Learn**: As you will see, Python is extremely easy to get started with. Python has an extraordinarily simple syntax, as already mentioned.

3. **Free and Open Source:** Python is an example of a FLOSS (Free/Libré and OpenSource Software). In simple terms, you can freely distribute copies of this software, read it's source code, make changes to it, use pieces of it in new free programs, and that you know you can do these things. FLOSS is based on the concept of a community which shares knowledge. This is one of the reasons why Python is so good - it has been created and is constantly improved by a community who just want to see a better Python.

4. **High-level Language:** When you write programs in Python, you never need to bother about the low-level details such as managing the memory used by your program,etc.

5. **Portable:** Due to its open-source nature, Python has been ported (i.e. changed to make it work on) to many platforms. All your Python programs can work on any of these platforms without requiring any changes at all if you are careful enough to avoid any system-dependent features. You can use Python on Linux, Windows, FreeBSD, Macintosh, Solaris,

OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and even PocketPC !

You can move Python programs from one platform to another, and run it without any changes. It runs seamlessly on almost all platforms including Windows, Mac OS X and Linux.

6. **Interpreted:** This requires a bit of explanation.
A program written in a compiled language like C or C++ is converted from the source language i.e. C or C++ into a language that is spoken by your computer (binary code i.e. 0s and 1s) using a compiler with various flags and options. When you run the program, the linker/loader software copies the program from hard disk to memory and starts running it.

Python, on the other hand, does not need compilation to binary. You just *run* the program directly from the source code. Internally, Python converts the source code into an intermediate form called bytecodes and then translates this into the native language of your computer and then runs it. All this, actually, makes using Python much easier since you
don't have to worry about compiling the program, making sure that the proper libraries are linked and loaded, etc, etc. This also makes your Python programs much more portable, since you can just copy your Python program onto another computer and it just works!

7. **Object Oriented:** Python supports procedure-oriented programming as well as object oriented programming. In *procedure-oriented* languages, the program is built around procedures or functions which are nothing but reusable pieces of programs. In *object-oriented* languages, the program is built around objects which combine data and functionality. Python has a very powerful but simplistic way of doing OOP, especially when compared to big languages like C++ or Java.

8. **Extensible:** If you need a critical piece of code to run very fast or want to have some piece of algorithm not to be open, you can code that part of your program in C or C++ and then use them from your Python program.

9. **Embeddable** You can embed Python within your C/C++ programs to give 'scripting' capabilities for your program's users.

10. **Extensive Libraries** The Python Standard Library is huge indeed. It can help you do various things involving regular expressions, documentation generation, unit testing, threading, databases, web browsers, CGI, ftp, email, XML, XML-RPC, HTML, WAV files, cryptography, GUI (graphical user interfaces), Tk, and other system-dependent stuff. Remember, all this is always available wherever Python is installed. This is called the 'Batteries Included' philosophy of Python.

Besides, the standard library, there are various other high-quality libraries
such as wxPython [http://www.wxpython.org], Twisted
[http://www.twistedmatrix.com/products/twisted], Python Imaging Library
[http://www.pythonware.com/products/pil/index.htm] and many more.

Reasons to Choose Python as First Language

1. Simple Elegant Syntax

Programming in Python is fun. It's easier to understand and write Python code. Why? The syntax feels natural. Take this source code for an example:

```
a = 2
b = 3
sum = a + b
print(sum)
```

Even if you have never programmed before, you can easily guess that this program adds two numbers and prints it.

2. Not overly strict

You don't need to define the type of a variable in Python. Also, it's not necessary to add semicolon at the end of the statement.

Python enforces you to follow good practices (like proper indentation). These small things can make learning much easier for beginners.

3. Expressiveness of the language

Python allows you to write programs having greater functionality with fewer lines of code. Here's a link to the source code of Tic-tac-toe game with a graphical interface and a smart computer opponent in less than 500 lines of code. This is just an example. You will be amazed how much you can do with Python once you learn the basics.

Install and Run Python in Windows

1. Go to Download Python page on the official site and click Download Python 3.6.0 (You may see different version name).

2. When the download is completed, double-click the file and follow the instructions to install it.

When Python is installed, a program called IDLE is also installed along with it. It provides graphical user interface to work with Python.

3. Open IDLE, copy the following code below and press enter.

```
print("Hello, World!")
```

4. To create a file in IDLE, go to File > New Window (Shortcut: Ctrl+N).

5. Write Python code (you can copy the code below for now) and save (Shortcut: Ctrl+S) with .py file extension like: hello.py or your-first-program.py

```
print("Hello, World!")
```

6. Go to Run > Run module (Shortcut: F5) and you can see the output. Congratulations, you've successfully run your first Python program.

Python Variables

A variable is an identifier name that refers to a value. An assignment statement creates new variable and gives it a value.

(in python we need not to assign datatype to a variable)

Creating Variables

Unlike other programming languages, Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

```
x = 5
y = "John"
print(x)
print(y)
```

Variable Names/Rules for writing identifiers

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Output Variables

The Python print statement is often used to output variables.

To combine both text and a variable, Python uses the + character:

```
x = "awesome"
print("Python is " + x)

x = "Python is "
y = "awesome"
z = x + y
print(z) # Python is awesome

x = 5
y = "John"
print(x + y)  #error

a, b, c = 5, 3.2, "Hello"
print (a)
print (b)

print (c)
```

Python Indentations

Where in other programming languages the indentation in code is for readability only, in Python the indentation is very important. Python uses indentation to indicate a block of code.

Instead of curly braces { }, indentations are used to represent a block.

```
if 5 > 2:
        print("Five is greater than two!")
```

Python will give you an error if you skip the indentation:

```
if 5 > 2:
print("Five is greater than two!")
```

Comments

Python has commenting capability for the purpose of in-code documentation. Comments start with #, and Python will render the rest of the line as a comment:

```
#This is a comment. print("Hello, World!")
```

Docstrings

Docstrings are also comments:

Python also has extended documentation capability, called docstrings. Docstrings can be one line, or multiline. Python uses triple quotes at the beginning and end of the docstring: """This is a multiline docstring.""" print("Hello, World!")

Python Keywords

Keywords are the reserved words in Python.

We cannot use a keyword as a variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.

In Python, keywords are case sensitive.

There are 33 keywords in Python 3.7. This number can vary slightly in the course of time.

All the keywords except True, False and None are in lowercase and they must be written as it is. The list of all the keywords is given below.

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

Your First Python Program

Often, a program called "Hello, World!" is used to introduce a new programming language to beginners. A "Hello, World!" is a simple program that outputs "Hello, World!".

However, Python is one of the easiest language to learn, and creating "Hello, World!" program is as simple as writing print("Hello, World!"). So, we are going to write a different program.

Program to Add Two Numbers

```
# Add two numbers
num1 = 3
num2 = 5
sum = num1+num2
print(sum)
```

How this program works?

Line 1: # Add two numbers

Any line starting with # in Python programming is a comment.

Comments are used in programming to describe the purpose of the code. This helps you as well as other programmers to understand the intent of the code. Comments are completely ignored by compilers and interpreters.

Line 2: num1 = 3

Here, num1 is a variable. You can store a value in a variable. Here, 3 is stored in this variable.

Line 3: num2 = 5

Similarly, 5 is stored in num2 variable.

Line 4: sum = num1+num2

The variables num1 and num2 are added using + operator. The result of addition is then stored in another variable sum.

Line 5: print(sum)
The print() function prints the output to the screen. In our case, it prints 8 on the screen.

"""Program 1:

Add two numbers

"""

num1 = 15

num2 = 12

Adding two nos

sum = num1 + num2

printing values

print("Sum of {0} and {1} is {2}" .format(num1, num2, sum))

Output:

Sum of 15 and 12 is 27

Input/Output Statements:

- A statement is a unit code that the python interpreter can execute.
- Python provides a built-in function called *input* that gets/reads input from the keyboard.
- When the user presses return or enter, the program resumes and input returns what the user typed as a *string*.
- Input taken from user is always in string format.
- Inbuilt function *int* will convert string to integer.
- OUTPUT: *print* is an executable statement.

Python Numbers
There are three numeric types in Python:

- int
- float
- complex

```
x = 1 # int
y = 2.8 # float
z = 1j # complex
```

int

```
x = 1
y = 35656222554887711
z = -3255522
```

Float
x = 1.10
y = 1.0
z = -35.59

x = 35e3
y = 12E4
z = -87.7e100

Complex
x = 3+5j
y = 5j
z = -5j

Python Casting
There may be times when you want to specify a type on to a variable. This can be done with casting. Python is an object-orientated language, and as such it uses classes to define data types, Casting in python is therefore done using constructor functions:

int() - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)

float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)

str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

```
x = int(1) # x will be 1
y = int(2.8) # y will be 2
z = int("3") # z will be 3
x = float(1) # x will be 1.0

y = float(2.8) # y will be 2.8

z = float("3") # z will be 3.0

w = float("4.2") # w will be 4.2

x = str("s1") # x will be 's1' y = str(2) # y will be '2' z = str(3.0) # z will be '3.0'
```

String Literals
String literals in python are surrounded by either single quotation marks, or double quotation marks.
'hello' is the same as "hello".
Strings can be output to screen using the print function. For example: print("hello").
Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

a = "Hello, World!" print(a[1]) #e
Substring. Get the characters from position 2 to position 4
b = "Hello, World!" print(b[2:5]) #llo
The len() method returns the length of a string:
a = "Hello, World!" print(len(a))
The lower() method returns the string in lower case:
a = "Hello, World!" print(a.lower())
The upper() method returns the string in upper case:
a = "Hello, World!" print(a.upper())
The replace() method replaces a string with another string:
a = "Hello, World!" print(a.replace("H", "J"))

Python Operators

Operators are used to perform operations on variables and values.
Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Python Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations:

Operator	**Name**	**Example**	**Arithmetic Operators : x=10 and y=20**
+	Addition	x + y	30
-	Subtraction	x - y	-10
*	Multiplication	x * y	200
/	Division	x / y #Floating point division returns floating value after division	0.5
%	Modulus	x % y	10
**	Exponentiation	x ** y	10 to power 20
//	Floor division	x // y #Integer Division c=a//b return integer value after division	0

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	**Description**	**Example**
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c – a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a

```
a = 21
b = 10
c = 0

c = a + b
print ( c )

c += a          #c=c+a
print ( c )

c *= a   #c=c*a
print ( c )

c /= a          #c=c/a
print ( c )

c  = 2
c %= a
print ( c )

c **= a         #c=c**a
print ( c )
```

Python Comparison Operators
Comparison operators are used to compare two values:

Operator	Name	Example	
==	Equal	x == y	If the values of two operands are equal, then the condition becomes true.
!=	Not equal	x!= y	if values of two operands are not equal, then condition becomes true.
>	Greater than	x > y	If the value of left operand is greater than the value of right operand, then condition becomes true.
<	Less than	x < y	If the value of left operand is less than the value of right operand, then condition becomes true.
>=	Greater than or equal to	x >= y	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.

Python Logical Operators
Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

```
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)
```

OUTPUT:
x and y is False
x or y is True
not x is False

Python Identity Operators
Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns true if both variables are the same object (True if the	x is y

	operands are identical)	
is not	Returns true if both variables are not the same object (True if the operands are not identical)	x is not y

X1 = 'Welcome'

X2 = 1234

Y1 = 'Welcome'

Y2 = 1234

print(X1 **is** Y1) #True

print(X1 **is not** Y1) #False

print(X1 **is not** Y2) #True

print(X1 **is** X2) #False

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x
print(x is z)
# returns True because z is the same object as x
print(x is y)
# returns False because x is not the same object as y, even if they have thew same content
print(x == y)
# to demonstrate the difference betweeen "is" and "==": this comparison returns True because x is equal to y

x = ["apple", "banana"]
y = ["apple", "banana"]
z = x
print(x is not z)
# returns False because z is the same object as x
print(x is not y)
# returns True because x is not the same object as y, even if they have the same content
print(x != y)
# to demonstrate the difference between "is not" and "!=": this comparison returns False because x is equal to y
```

Python Membership Operators

Membership operators are used to test if a sequence is presented in an object:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

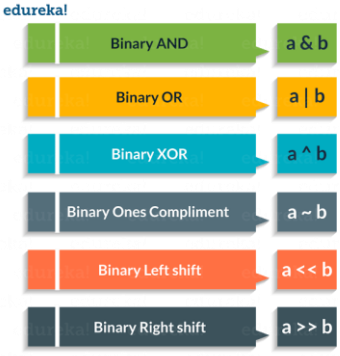
```
x = ["apple", "banana"]
print("banana" in x)
# returns True because a sequence with the value "banana" is in the list

x = ["apple", "banana"]
print("pineapple" not in x)
# returns True because a sequence with the value "pineapple" is not in the list


X = [1, 2, 3, 4]
A = 3
print(A in X) #True
print(A not in X) #False
```

Bitwise Operators

These operations directly manipulate bits. In all computers, numbers are represented with bits, a series of zeros and ones. In fact, pretty much everything in a computer is represented by bits.




```
Eg1:
a = 58       # 111010
b = 13       # 1101
c = 0

c = a & b
print ( c )   # 8 = 1000

c = a | b
print ( c )   # 63 = 111111
```

```
c = a ^ b
print ( c )   # 55 = 110111

c = a >> 2
print ( c )   # 232 = 11101000

c = a << 2
print ( c )   # 14 = 1110
```

eg2:

```
a = 10
b = 4

print('Bitwise AND=', a & b)
print('Bitwise OR=', a | b)
print('Bitwise NOT=', ~ a)    # -(10+1)
print('Bitwise XOR=', a ^ b)
print('Bitwise right shift=', a >> 2)
print('Bitwise left shift=', a << 2)
```

OUTPUT:

```
Bitwise AND= 0
Bitwise OR= 14
Bitwise NOT= -11
Bitwise XOR= 14
Bitwise right shift= 2
Bitwise left shift= 40
```

Conditional Statements:

Conditional statements are used to execute a statement or a group of statements when some condition is true. There are namely three conditional statements – If, Elif, Else.

1. if statement:

"if statement" is written by using the if keyword.

```
a = 33
b = 200
if b > a:
Tab spacing print("b is greater than a")
```

In this example we use two variables, a and b, which are used as part of the if statement to test whether b is greater than a. As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

- The Boolean expression after the if statement is called the condition.
- The if statement with a colon character and the lines after the of statement are indented (tab spacing).

- These statements are called compound statements because they stretch across more than one line.
- There is no limit on the number of statements that can appear in the body, but there must be atleast one.
- If body has no statements then pass statement may be used, which does nothing.

Ex:

```
X=100
if x>99:
        x=x+5
if x<101 and x>99:
        y=x+100
        x=x-50
        x=x+y
print(x)
```

2. **Alternative Execution (if else)**

Ex:

```
#x=100
x=input("input a number:")
x=int(x)
if x%2==0:
        print(x,"is even number")
else:
        print(x,"is odd number")
```

OR

```
num = int(input("Enter a number: "))
if (num % 2) == 0:
   print("{0} is Even".format(num))
else:
   print("{0} is Odd".format(num))
```

3. **Chained Conditionals**

- If there are more than two possibilities and we need more than two branches. This can be done by chained condition.
- Here, elif is an abbreviation of “else if” .

Ex: greater of two integers

```
x=10
y=20
if x<y:
        print('x is less than y')
elif x>y:
        print('x is greater than y')
else:
        print('x and y are equal')
```

WAP in python to accept an integer from the user and determine whether its positive, negative or zero.

```
num=input(‘Enter  a number’)
num=int(num)
If num>0:
        print(num,‘is a positive number’)
elif num<0:
        print(num,‘is a negative number’)
else:
        print(num,‘is zero’)
```

4. **NESTED CONDITION:**

Write a program in python to take three integers and determine the largest of the three.

```
n1=input(‘Input the first integer’)
n1=int(n1)
n2=input(‘Input the second integer’)
n2=int(n2)
n3=input(‘Input the third integer’)
n3=int(n3)
```

```
if n1>n2:
        if n1>n3:
                print(n1,'is the largest')
        else:
                print(n3,'is the largest')
elif n2>n3:
        print(n2,'is the largest')
else:
        print(n3,'is the largest')
```

Python Loops:

- In general, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on.
- There may be a situation when you need to execute a block of code several number of times

There are two types of loops:
Infinite: When condition will never become false
Finite: At one point, the condition will become false and the control will move out of the loop

Python has two primitive loop commands:
while loops
for loops

The while Loop

The while statement allows you to repeatedly execute a block of statements as long as a condition is true.
While Loop: Here, first the condition is checked and if it's true, control will move inside the loop and execute the statements inside the loop until the condition becomes false.
We use this loop when we are not sure how many times we need to execute a group of statements or you can say that when we are unsure about the number of iterations.

Syntax:

```
while expression:
   statement(s)
print("Outside While")
```

Example :Print i as long as i is less than 6:

```
i = 1
while i < 6:
        print(i)
         i += 1
```

Note: remember to increment i, or else the loop will continue forever.

The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

The break Statement

Sometimes it's required to terminate the current iteration and exit the loop. In that case break statement may be used. With the break statement we can stop the loop even if the while condition is true:

Example: Exit the loop when i is 3

```
i = 1
while i < 6:
        print(i)
        if i == 3:
                break
        i += 1
```

The continue Statement

Sometimes it's required to skip the current iteration of a loop and immediately jump to the next iteration. In that case continue statement may be used.

With the continue statement we can stop the current iteration, and continue with the next:

Example: Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
        i += 1
        if i == 3:
                continue
        print(i)
```

Python For Loops

For Loop: Like the While loop, the For loop also allows a code block to be repeated certain number of times. The difference is, in For loop we know the amount of iterations required unlike While loop, where iterations depends on the condition. You will get a better idea about the difference between the two by looking at the syntax:

```
1  for variable in Sequence:
2      statements
```

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

The range(i) function generates an iterator to progress the integer numbers starting with 0 and ending with (n-1) i.e 9.

 # Factorial of a number

```
num = int(input("enter a number: "))
fac = 1
for i in range(1, num + 1):
   fac = fac * i
print("factorial of ", num, " is ", fac)
```

Write a program in Python to add the sum of first 500 numbers.

```
N=500
sum=0
for i in range(1,N+1):
  sum=sum+i
print("sum  of 1 to %d is %d" %(N,sum))
print("sum=",sum)
print("sum  of 1 to {} is {}" .format(N,sum))
```

The range() Function

To loop through a set of code a specified number of times, we can use the range() function, The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example: Using the range() function:

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

Note that range(6) is not the values of 0 to 6, but the values 0 to 5.

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

Example: Using the start parameter:

```
for x in range(2, 6):
        print(x)
```

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, **3**):

Example: Increment the sequence with 3 (default is 1):

```
for x in range(2, 30, 3):
        print(x)
```

pass Statement:

- pass statement can be used to indicate empty functions, classes and loops. Often nothing happens.
- Indicate a “null” block. Pass can be places on the same line, or on a separate line.
- *Can be used to add a delay in a program by just iterating through the loop.*

Ex: Counts from 1 to 1000 without doing anything.

```
for i in range(1, 1001):
        pass
print('done')
```

Nested Loops: It basically means a loop inside a loop. It can be a For loop inside a While loop and vice-versa. Even a For loop can be inside a For loop or a While loop inside a While loop.

 # Program to check if a number is prime or not

```
N = int(input("Enter a number: "))
for num in range(2, N+1):
   flag = 0
   for j in range(2, num//2+1):
      if ((num % j) == 0):
# if factor is found, set flag to True
         flag=1
# break out of loop
         break
# check if flag is True
   if flag==0:
      print(num)
```

Consider the example:

```
count = 1
for i in range(10):
    print (str(i) * i)

    for j in range(0, i):
        count = count +1
```

```
Output =
1
22
333
4444
55555
666666
7777777
88888888
999999999
```

Standard Data Types

- The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.
  Python has five standard data types −
- Numbers
- String
- List
- Tuple
- Dictionary

List :

- Lists are the most versatile of Python's compound data types.
- A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.
- The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

For example −

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']
print(list) # Prints complete list
print(list[0]) # Prints first element of the list
print(list[1:3]) # Prints elements starting from 2nd till 3rd
print(list[2:]) # Prints elements starting from 3rd element
print(tinylist * 2) # Prints list two times
print(list + tinylist) # Prints concatenated lists
print(list,tinylist)
```

Output:

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
['abcd', 786, 2.23, 'john', 70.2] [123, 'john']
```

- Lists are mutable : means lists can be altered or modified
- Unlike strings, lists are mutable because the order of items in a list can be changed or an item in a list can be reassigned.

#SLICING A list

```
t=['a','b','c','d','e','f']
print(t[1:3]) #displays from 2nd element to 3-1=2 i.e 3rd Element
print(t[:4]) #displays from 0th element to 4-1=3 i.e 4th element
print(t[4:])
print(t[:]) # Copy of the whole list
```

#OUTPUT:

```
['b', 'c']
['a', 'b', 'c', 'd']
['e', 'f']
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, A slice operator on the left side of an assignment can update multiple elements:

```
t=['a','b','c','d','e','f']
t[1:3]=['x','y']
print(t)
#OUTPUT:
['a', 'x', 'y', 'd', 'e', 'f']
```

 # LISTS METHODS: Python provides methods that operate on LISTS

 # 1. append: adds a new element to the end of a list

```
t=['a','b','c','d','e','f']
t.append('g')
print(t)  # ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

 # 2. extend: takes a list as an argument and appends all of the elements while it leaves t2 unmodified.

```
t1=['a','b','c']
t2=['d','e']
t1.extend(t2)
print(t1) # ['a', 'b', 'c', 'd', 'e']
print(t2) #['d', 'e']
```

 # 3.Sorting: arranges elements of list from low to high

```
t=['g', 'f', 'e', 'd', 'c', 'b', 'a']
t.sort()
print(t) #['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

NOTE: Most lists methods are void; they modify the list and return None.

 # DELETING ELEMENTS:

- if the index of the element to be deleted is known, pop can be used.

```
t=['a','b','c']
x=t.pop(1) #not a void method, it returns a value that is deleted
print(t) #['a', 'c']
```

NOTE: pop modifies the list and returns the element that was removed. If an index is not provided, it deletes and returns the last element.

- if value removed is not desired i.e not to be returned, then del operator can be used:

```
t=['a','b','c']
del t[1]
print(t) #['a', 'c']
```

#remove: if the element to be removed is known(but not the index), remove can be used:

```
t=['a','b','c’]
t.remove('b’)
print(t) #['a', 'c’]
# the return value from remove is None.
```

'''Write a Python program that accepts a list and modifies it by removing the first and last elements and then first prints those deleted items and then it prints the rest of the list.'‘’

```
names=['Ram','Ramses','Ramesh',[11,2020]]
print(names.pop(0))
print(names.pop(2))
print('Modified List is:’)
for var in names:
        print(var,",end=‘’)
```

'''Write a Python program that accepts a list and modifies it by removing the first and last elementsand then copies the remaining list to a new list. Print the new list.'‘’

```
names=['Ram','Ramses','Ramesh',[11,2020]]
del names[0]
del names[2]
newlist=names
for var in newlist:
        print(var,",end='')
```

Python Tuples

- A tuple is another sequence data type that is similar to the list.
- A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.
- The main differences between lists and tuples are:
- Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as read-only lists.

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')
print tuple # Prints complete list
print tuple[0] # Prints first element of the list
print tuple[1:3] # Prints elements starting from 2nd till 3rd
print tuple[2:] # Prints elements starting from 3rd element
print tinytuple * 2 # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

OUTPUT:

```
('abcd', 786, 2.23, 'john', 70.2)
Abcd
 (786, 2.23)
 (2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

- Syntactically, a tuple is a comma- separated list of values:

t='a','b','c','d','e’

- Although it is not necessary, it is common to enclose tuples in parentheses:

t=('a','b','c','d','e’)

#To create a tuple with a single element, you must include the final comma:

t1=('a’,)
print(type(t1))
<type 'tuple’>
#without the comma python treats ('a') as an expression with a string in parentheses that evaluates to a string:
t2=('a’)
print(type(t2))
<type 'str'>

- Another way to construct a tuple is the built-in function tuple. With no argument, it creates an empty tuple:
  t=tuple()
  print(t) #()

Because tuple is the name of a constructor, should be avoided to be used as a variable name. Most list operators also work on tuples. The bracket operator indexes an element:

t=('a','b','c','d','e’)
print(t[0]) #a
t=tuple('graphic’)
print(t) # ('g', 'r', 'a', 'p', 'h', 'i', 'c’)

# Slice operator selects a range of elements.
t=tuple('graphic’)
print(t[1:3]) #('r', 'a’)

# But if one of the elements of the tuple is modified, it generates an error.
t=tuple('graphic’)
t[0]=‘A’ # TypeError: 'tuple' object does not support item assignment

# the elements of a tuple can't be modified, but can be replaced with another tuple:
t=('a','b','c','d','e’)
t=('A',)+t[1:]print(t) #('A', 'b', 'c', 'd', 'e')

Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example −

```
dict = {}
 dict['one'] = "This is one"
dict[2] = "This is two“
 tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
print dict['one'] # Prints value for 'one' key
print dict[2] # Prints value for 2 key
 print tinydict # Prints complete dictionary
print tinydict.keys() # Prints all the keys
print tinydict.values() # Prints all the values
```

OUTPUT:

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john’]
```

 # DATA STRUCTURE FILES

```
d={}
i=65
for item in range(5):
        d[item]=chr(i)
        i=i+1
print(d)
```

OUTPUT: {0: 'A', 1: 'B', 2: 'C', 3: 'D', 4: 'E'}

- A dictionary is like a list. In a list, the index positions have to be integer; in a dictionary, the indices can be (almost) any type.
- Dictionary can be visualized as a mapping between a set of indices(which are called keys) and a set of values.
- Each **key** maps to a **value**. The association of a key and a value is called a **key: value** pair or sometimes an item.

- **dict: built in function**
- The function dict creates a new dictionary with no items. Because dict is the name of a built-in function, should be avoided as a variable name.

d=dict()

print(d) # {} #represents an empty dictionary. To add items square brackets can be used:

d=dict()

d['one']='uno'

print(d)

This line creates an item that maps from the key ‘one’ to the value “uno”.

d[‘one’]=‘uno’

If the dictionary is printed, it prints a key-value pair with a colon between the key and value:

print(d) #{‘one’:’uno’}

This input format is also an input format. For example, to create a new dictionary with three items.

d={‘one’:’uno’, ‘two’:’dos’, ‘three’: ‘tres’}

print(d)

{‘one’:’uno’, ‘three’:’tres’,’two’:’dos’}

The order of the key-value pairs is not the same. The elements of a dictionary are never indexed but can be accessed with keys that can be used to look up the corresponding values.

print(d[‘two’]) #’dos’

- # len function and in operator
- # len function works on dictionaries; it returns the number of key-value pairs:

d={'one':'uno', 'two':'dos', 'three': 'tres'}

L=len(d)

print(L) #3

in operator works on dictionaries; it tells you whether something appears as a key in the dictionary.

d={'one':'uno', 'two':'dos', 'three': 'tres'}

D='one'in d

print(D) # True

‘uno’in d

```
ab = { 'Swaroop' : 'swaroopch@byteofpython.info',
'Larry' : 'larry@wall.org',
'Matsumoto' : 'matz@ruby-lang.org',
'Spammer' : 'spammer@hotmail.com'
}
print("Swaroop's address is %s",ab['Swaroop'])
# Adding a key/value pair
ab['Guido'] = 'guido@python.org'
# Deleting a key/value pair
del ab['Spammer']
print('\nThere are %d contacts in the address-book\n' % len(ab))
for name, address in ab.items():
    print('Contact %s at %s' % (name, address))
if 'Guido' in ab: # OR ab.has_key('Guido')
    print("\nGuido's address is %s" % ab['Guido'])
```

#LOOPING AND dictionary: If we use a dictionary as the sequence in for statement, it traverses the keys of the dictionary.

In the example below, it prints each key and its corresponding value:

```
Words={'Chuck':1, 'Annie':42, 'Jan':100}
for key in Words:
        print(key,Words[key])
```

OUTPUT: Chuck 1 Annie 42 Jan 100

ex: To find all the entries in a dictionary with a value above ten:

```
counts={'Chuck':1, 'Annie':42, 'Jan':100}
for key in counts:
        if counts[key]>10:
                print(key,counts[key])
```

OUTPUT: Jan 100 Annie 42

DATA STRUCTURE FILES

- Stored on a **persistence storage device** ex: Secondary Memory, Pen drive, Hard Drive etc.
- Secondary memory is **not erased when power is turned off**.
- The main focus will be on reading and writing text files such as those created in a text editor.

Opening Files:

- When file has to be read or written(onto the hard drive), first it must be opened.
- Opening the file communicates with the operating system running on that computer, which knows where the data for each file is stored.
- When a file is opened, the operating system finds the file by name and makes sure the file exists.

General Syntax:

- **filehandle=open("filename",mode)**
- filename: any file name present/to be created in the path.
- The modes are:

'r'- This is the default mode. It opens file for reading.

'w' –This mode opens file for writing. If file does not exist, it creates a new file. If file exists it truncates the file. (any existing file with same name will be erased).

'a' – Append mode, is used to add new data to the end of the file; that is new information is automatically append to the end.

'x' – Create- will create a file, returns an error if the file exist.

't': This is the default mode. It opens in text mode.

'rt': It opens in text file in read mode.

'b': This opens in binary mode. Binary format is usually used when dealing with images, videos, etc.

'+': This will open a file for reading and writing (updating).

'rb': Opens the file as read-only in binary format.

'wb': Opens a write-only file in binary mode.

- In this ex: the file mbox.txt is opened in read mode by default and is stored in the same folder as that off where Python interpreter is started.

- **fhand=open(‘mbox.txt’)**
- If the **open** call is successful, the operating system returns a file handle i.e if the requested file exists and has the proper permissions to read the file.
- The file handle is not the actual data contained in the file, but instead it is a “handle” that can be used to read the data.
- It’s good practice to close a file: **filehandle.close()**
- If the file does not exist, open will fail with the message Traceback, and the handle is not available to access the contents of the file:

fhand=open(‘stuff.txt’)

- try and except model can be used to handle the situation of opening a file that does not exist.
- **Write:**
- file = open('geek.txt','w')
- file.write("This is the write command")
- file.write("It allows us to write in a particular file")
- file.close()
- **Read:**
- # Python code to illustrate read() mode
- file = open("file.txt", "r")
- print (file.read())
- **Append:**
- # Python code to illustrate append() mode
- file = open('geek.txt','a')
- file.write("This will add this line")
- file.close()

READING FILE:

- The file handle does not contain the data for the file, however it can be read as the whole file into on string using the read method on the file handle especially suitable on the small size files.

fhand=open(‘m-box.txt’,r)

inp=fhand.read()

print(inp) #displays the entire content of the file

```
print(len(inp)) # displays the total number of chars.
2582
fhand=open('m-box.txt')
inp= fhand.readlines()
print(inp)
```

 # read lines will separate each line and present the file in a readable format.

```
fhand= open('m-box.txt')
inp= fhand.read(100)
print(inp)
```

#Displays first hundred characters from beginning of the file.

Writing To a text file

write(): inserts a string in a single line in the text file.

Ex: To write a string to a file

```
str1= 'An apple a day keeps the Doctor away.'
str2='\nMake hay while sun shines.\n'
try
        fh=open('Adage.txt', 'w+')
        fh.write(str1) or fh.write(str2)
        print('File was created successfully')
        print('File was appended successfully')
        fh.close()
except:
        print('File Error!!!')
```

"""Program 1:

Add two numbers

"""

```
num1 = 15
num2 = 12
```

```
# Adding two nos
sum = num1 + num2
  # printing values
print("Sum of {0} and {1} is {2}" .format(num1, num2, sum))
```

Program 2:

Add Two Numbers Provided by The User

```
"""Program 3:
Find area of rectangle
"""

# Store L & B
length= int(input('Enter L: '))
bredth = int(input('Enter B: '))

area= length*bredth

# Display the result
print(area)

"""Program 4:
Find area of Circle
"""

# Store radius
rad=float(input('Enter radius: '))
```

```
area=3.14*rad*rad

# Display the result using different method
print(area)
print('The area of the circle is ' ,area)
print('The area of the circle is %0.2f' %area)
print('The area of the circle is {0}' .format(area))
```

"""Program 5:

Find the area of triangle

If a, b and c are three sides of a triangle. Then

s = (a+b+c)/2

area = √(s(s-a)*(s-b)*(s-c))

"""

```
a = float(input('Enter first side: '))
b = float(input('Enter second side: '))
c = float(input('Enter third side: '))

# calculate the semi-perimeter
s = (a + b + c) / 2

# calculate the area
area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
print(area)
print('The area of the triangle is %0.2f' %area)
print('The area of the triangle is %0.4f' %area)
```

"""Program 6:

Find roots of quadratic equation ax2 + bx + c = 0**

ax2 + bx + c = 0, where**

a, b and c are real numbers and

a ≠ 0

"""

```
# import complex math module
import cmath
a = float(input('Enter a: '))
b = float(input('Enter b: '))
c = float(input('Enter c: '))
# calculate the discriminant
d = (b**2) - (4*a*c)
# find two solutions
sol1 = (-b-cmath.sqrt(d))/(2*a)
sol2 = (-b+cmath.sqrt(d))/(2*a)
print('The solution are {0} and {1}'.format(sol1,sol2))
```

"""**Program 7:**

Swap two variables

"""

```
# To take inputs from the user
x = input('Enter value of x: ')
y = input('Enter value of y: ')
# create a temporary variable and swap the values
temp = x
x = y
y = temp
print(x,y)
print('The value of x and y after swapping are: {0} and {1}'.format(x,y))
```

Program Based on If Else Elif

(Python Decision Making)

"""Program 8:

Check if a Number is Positive, Negative or 0

"""

```
num = float(input("Enter a number: "))
if num > 0:
   print("Positive number")
elif num == 0:
   print("Zero")
else:
   print("Negative number")
```

"""Program 9:

Nested if else

Check if a Number is Positive, Negative or 0

"""

```
num = float(input("Enter a number: "))
if num >= 0:
   if num == 0:
      print("Zero")
   else:
      print("Positive number")
else:
   print("Negative number")
```

"""Program 10:

Check if the input number is odd or even.

A number is even if division by 2 gives a remainder of 0.

If the remainder is 1, it is an odd number.

"""

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

if (num % 2) == 0:
   print(num," is Even",)
else:
   print(num," is Odd")
```

"""**Program 11:**

Find the largest number among the three input numbers

"""

```
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
num3 = float(input("Enter third number: "))

if (num1 >= num2) and (num1 >= num3):
   largest = num1
elif (num2 >= num1) and (num2 >= num3):
   largest = num2
else:
   largest = num3

print("The largest number is", largest)
```

"""**Program 12:**

Check entered character is alphabet or not

"""

```
# taking user input
ch = input("Enter a character: ")

if((ch>='a' and ch<= 'z') or (ch>='A' and ch<='Z')):
    print(ch, "is an Alphabet")
else:
    print(ch, "is not an Alphabet")
```

"""**Program 13:**

Program to Check Vowel or Consonant

"""

```
# taking user input
ch = input("Enter a character: ")

if(ch=='A' or ch=='a' or ch=='E' or ch =='e' or ch=='I'
 or ch=='i' or ch=='O' or ch=='o' or ch=='U' or ch=='u'):
    print(ch, "is a Vowel")
else:
    print(ch, "is a Consonant")
```

"""**Program 14: Print first n natural numbers starting from 1** """

```
n=int(input("Enter range>"))
i=1
print("The List of Natural Numbers from 1 to {0} are".format(n))
while (i<=n) :
    print(i)
    i=i+1
```

"""Program 15: Sum of first n natural number """

```
n=int(input("Enter range "))

if n < 0:
   print("Enter a positive number")
else:
   sum = 0
   # use while loop to iterate until zero
   while(n > 0):
      sum += n
      n-= 1
   print("The sum is", sum)
```

"""Program 16: Find factorial of inputted number """

```
n=int(input("Enter any number "))
i=1
fact=1
while (i<=n):
    fact=fact*i
    i=i+1
print("The factorial of  {0} is {1}".format(n,fact))
```

"""**Logic Suppose n is 5 we are running loop while i doesn't become 5 starting from 1 each time we are executing** fact=fact*i

i.e. for i=1 fact=fact*i will give i.e 1*1=1 (fact is 1 now)

i.e. for i=2 fact=fact*i will give i.e 1*2=2 (fact is 2 now)

i.e. for i=3 fact=fact*i will give i.e 2*3=6 (fact is 6 now)

i.e. for i=4 fact=fact*i will give i.e 6*4=24 (fact is 24 now)

i.e. for i=5 fact=fact*i will give i.e 24*5=120 (fact is 120 now)

since i is 5 now (i<=n) i.e. 5<=5 will become false so loop exits"""

"""Program 17: Find sum of following series 1+3+5+7 upto n terms """

```
n = int(input("Enter range "))
i = 1
sum = 0
while (i <= 2*n-1):
  sum = sum+i
  i = i+2
print("The sum is", sum)
```

"""Logic Condition in while loop is (i <= 2*n-1) because we are increasing i using i = i+2 so its jumping two number each time so suppose user enter 5 for value of n condition will become i<=2*5-1 i.e i<=9 ans for n=5 series will be 1+3+5+7+9 you can see the last term is 9 """

"""Program 18: Find sum of following series 1-3+5-7 upto n terms """

```
n = int(input("Enter range "))
i = 1
sign=1
sum = 0
while (i <= n):
  sum = sum+(2*i-1)*sign
  sign=sign*(-1)
  i = i+1
print("The sum is", sum)
```

"""Logic i 1 to n i.e. n times i=1 2*i-1 will give 1 i=2 2*i-1 will give 3 i=3 2*i-1 will give 5

.....

.....

First time

sign=1

sign=sign*(-1)

1*(-1)

-1

Next time

sign=sign*(-1)

-1*(-1)

1

"""

"""Program 19: Print square of n numbers """

```
n = int(input("Enter range "))
i=1
while (i <= n):
   num=int(input("Enter number "))
   print(num*num)
   i = i+1
```

"""Program 20: Find sum of digit of inputted number """

```
num = int(input("Enter any number "))
i=1
sod=0
while (num != 0):
   rem=num % 10
   sod=sod+rem
   #num=num/10
   num=num//10
print(sod)
```

""" Logic

Suppose input 153 i.e. num is 153 and condition num != 0 is true we are running loop while 153 does not become 0

We need last digit of 153 so we will use

rem=num % 10 i.e. 153%10 will give remainder 3 (3rd digit extracted)

Now we need 15 (after extracting 3 from 153)

num=num//10 i.e. 153//10 will give 15 (floor division)

Repeat same since num is still 15 and condition num != 0 is true

rem=num % 10 i.e. 15%10 will give remainder 5 (2nd digit extracted)

num=num//10 i.e. 15//10 will give 1 (floor division)

Repeat same since num is still 1 and condition num != 0 is true

rem=num % 10 i.e. 1%10 will give remainder 1 (1st digit extracted)

num=num//10 i.e. 1//10 will give 0 (floor division)

Exit loop since num has become 0 and condition num != 0 is false

"""

"""Program 21:

Program to print elements of list and find the sum of all numbers stored in a list

"""

```
# List of numbers
numbers = [1,3,5,7,9,11,13,15,17,19]
# iterate over the list to print list elements
for val in numbers:
    print(val)
# variable to store the sum
sum = 0
# iterate over the list to find sum of elements of list
for val in numbers:
    sum = sum+val
print("The sum is ", sum)
```

```
# Finding average of elements of list
avg=sum/len(numbers)# len function used to calculate length of 'numbers' list
print("The avg is ", avg)

"""Program 22:
For loop and printing various collection data types
"""

thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
print()
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
print()
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
print()
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
for x in thisdict:
  print(x)
print()
```

```
for x in thisdict.values():
  print(x)
print()
for x, y in thisdict.items():
  print(x, y)
```

"""Program 23:

Precision Handling in Python Python end parameter in print()

"""

```
import math
# initializing value
a = 3.4536
# using trunc() to print integer after truncating
# truncate function is used to round up or down towards zero
print ("1-The integral value of number is : ",math.trunc(a))
# using ceil() to print number after ceiling
print ("2-The smallest integer greater than number is : ",math.ceil(a))
# using floor() to print number after flooring
print ("3-The greatest integer smaller than number is : ",math.floor(a))

# Python code to demonstrate precision
# and round()

# initializing value
a = 3.4536

# using "%" to print value till 2 decimal places
print ("4-The value of number till 2 decimal place(using %) is :")
print ('%.2f'%a)
```

```
# using format() to print value till 2 decimal places
print ("5-The value of number till 2 decimal place(using format()) is : ",end="")
print ("{0:.2f}".format(a))

# using round() to print value till 2 decimal places
print ("6-The value of number till 2 decimal place(using round()) is : ",end="")
print (round(a,2))

"""Program 24:
Understanding range function
"""

n=int(input("1-Enter range "))
#When 3 parameters i.e. Start,Range & Increment
for val in range(0,n,1):
    print(val,end=" ")
print()
#When 2 parameters i.e. Start and Range,Default increment 1
for val in range(0,n):
    print(val,end=" ")
print()
#When only 1 parameter i.e.only Range,Default Starts from 0, Default increment 1
for val in range(n):
    print(val,end=" ")
print()

n=int(input("2-Enter range "))
for x in range(2, n, 3):
  print(x,end=" ")
print()
```

```
#Range function to print a series without usign any loop
n=int(input("3-Enter range "))
print(list(range(2, n, 3)))

"""Program 25:
For loop with Break and Continue
"""

# program to display all the elements before number 88
for num in [11, 9, 88, 10, 90, 3, 19]:
  print(num)
  if(num==88):
            print("The number 88 is found")
            print("Terminating the loop")
            break
print()
# program to display only odd numbers
for num in [20, 11, 9, 66, 4, 89, 44]:
   # Skipping the iteration when number is even
   if num%2 == 0:
      continue
   # This statement will be skipped for all even numbers
   print(num)
```