Why Python

Python is a high-level, interpreted, interactive and object-oriented scripting language.

» **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.

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

» **Python is a Beginner's Language**: Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Why only Python: (Reasons to learn Python)

» **Dynamically Typed**: No type when declaring a variable, no need of type casting.

» **Simple Syntax**: No need of parenthesis, brackets, braces, commas. Less time in debugging.

» **One-liner**: More line in single line. Elegant one-line solutions.

Eg: x,y=y,x #swaping

» **English-like commands**: Readability is more.

Ex: name=”Bob”

print (name)

» **Intuitive Data Structures**: Lists, tuples, sets, dictionaries are powerful. Yet simple and intuitive to use.

History of Python

» Python was developed by Guido van Rossum in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.

» Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages

Major Versions of Python

There are three major versions of Python

» Version 1: Python 1.0: In January 1994, the first version of Python 1.0 was released. This version 1 includes the major new features like the functional programming tools filter, reduce, map, lambda, etc.

» Version 2: Python 2.0: After Six and a half years later, Python 2.0 was introduced in October 2000. In this release, a full garbage collector, python list comprehensions were included, and it also supports Unicode.

» Version 3: Python 3.0: Python then after 8 years, the next major release was made. This release was Python 3.0 also known as” Py3K” or “Python 3000”.

The major changes in Python 3.0 are:

» In this version, Print is a Python function

» Instead of lists, in this version, we have Views and iterators

» In this version, we have more simplified rules for ordering comparisons. For example, we cannot sort a heterogeneous list, because each element of a Python List must be comparable to other elements.

» In this python version, int. long is also an int as there is only one integer type.

- In this python version, when we divide two integers it resultant returns is a float instead of an integer. We can use "//" to have the "old" behavior.
- In this python version, Instead of Unicode Vs. 8-bit we have Text Vs. Data
- The one drawback of Python 3.0 is that it is not backward compatible with Python 2.x.

Python Features

- **Easy-to-learn**: Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read**: Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain**: Python's source code is fairly easy-to-maintain. since small quantity of lines easy to debug.
- **A broad standard library**: Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode**: Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Broad**: Can be used in all the fields.
- **Portable**: Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable**: You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases**: Python provides interfaces to all major commercial databases.
- **GUI Programming**: Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable**: Python provides a better structure and support for large programs than shell scripting.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below:

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- IT supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

How Python differs from C, C++ and Java

	C	C++	JAVA	PYTHON
Language Type	**Procedure Oriented**	**Object Oriented**	**Object Oriented**	**Both Procedure and Object Oriented**
Building Block	**Function Driven**	**Object Driven**	**Both Object and Class Driven**	**Function, Object and Class driven**
Extension	**.c**	**.cpp**	**.java**	**.py**
Platform	**Dependent**	**Independent**	**Independent**	**Independent**
Comment Style	**/* Multi Line */**	**//Single Line**	**//Single Line**	**#Single Line**
		/*Multi Line*/	**/*Multi Line*/**	**" " " Multi-line " " "**
Translator Type	**Compiled**	**Compiled**	**Compiled & Interpreted**	**Interpreted**

Representing Block of Statements	**{ }**	**{ }**	**{ }**	**Indentation**
Declaring Variables	**Required**	**Required**	**Required**	**Not Required**
Applications	**Compilers, Interpreters, Embedded Programming etc.,**	**Simple desktop applications, Embedded Systems etc.,**	**Desktop GUI, Mobile, Web, Gaming etc.,**	**Desktop, Web, Gaming, Network Programming etc.,**
Editors/ IDE	**Turbo C, Code Blocks**	**Turbo C++, Code Blocks**	**Ecllipse, Netbeans**	**PyCharm, PyDev, IDLE, Jupyter Notebook, Spyder, VS Code**

Python Applications

Python is known for its general-purpose nature that makes it applicable in almost every domain of software development. Python makes its presence in every emerging field. It is the fastest-growing programming language and can develop any application.

Following are the various application areas where Python can be applied.

» Web Applications:

Python can be used to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautifulSoup, Feedparser, etc. One of Python web-framework named Django is used on Instagram. Python provides many useful frameworks like Django and Pyramid framework, Flask and Bottle , Plone and Django CMS

» Desktop GUI Applications:

The GUI stands for the Graphical User Interface, which provides a smooth interaction to any application. Python provides a Tk GUI library to develop a user interface. Some popular GUI libraries are Tkinter or Tk, wxWidgetM, Kivy, PyQt or Pyside

» Console-based Application:

Console-based applications run from the command-line or shell. These applications are computer program which are used commands to execute. Python provides many free library or module which helps to build the command-line apps. The necessary IO libraries are used to read and write. It helps to parse argument and create console help text out-of-the-box. There are also advance libraries that can develop independent console apps.

» Software Development:

Python is useful for the software development process. It works as a support language and can be used to build control and management, testing, etc.

- » SCons is used to build control.
- » Buildbot and Apache Gumps are used for automated continuous compilation and testing.
- » Round or Trac for bug tracking and project management.

» Scientific and Numeric:

Python language is the most suitable language for Artificial intelligence or machine learning. It consists of many scientific and mathematical libraries, which makes easy to solve complex calculations. Python has many libraries for scientific and numeric such as Numpy, Pandas, Scipy, Scikit-learn, Matplotlib for developing Artificial Intelligence and Numeric applications

» Business Applications:

Business Applications differ from standard applications. E-commerce and ERP are an example of a business application. This kind of application requires extensively, scalability and readability, and Python provides all these features Python provides Oddo and Tryton platforms which is used to develop the business application.

» Audio or Video-based Applications:

Python is flexible to perform multiple tasks and can be used to create multimedia applications. Some multimedia applications which are made by using Python are TimPlayer, cplay, etc. Python provides multimedia libraries like Gstreamer, Pyglet, QT Phonon

» 3D CAD Applications:

The CAD (Computer-aided design) is used to design engineering related architecture. It is used to develop the 3D representation of a part of a system. Python can create a 3D CAD application by using Fandango, CAMVOX, HeeksCNC, AnyCAD, RCAM

» Enterprise Applications:

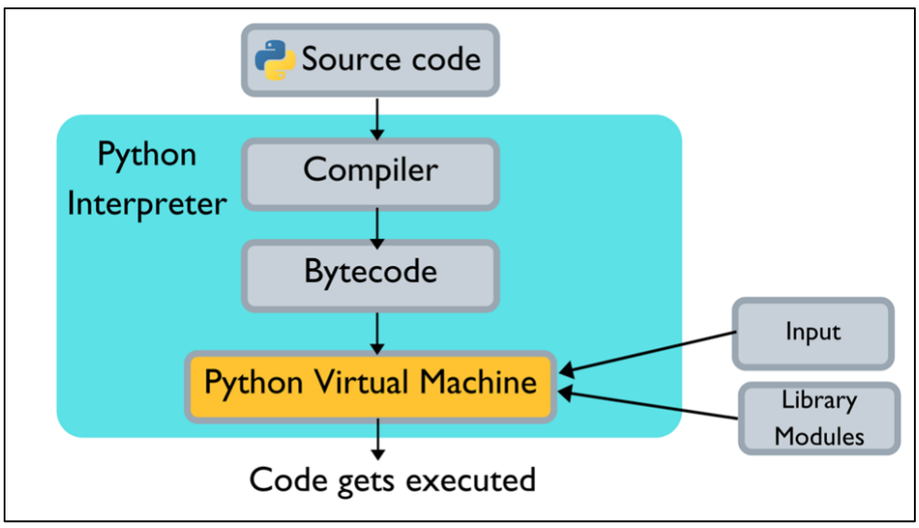
Python can be used to create applications that can be used within an Enterprise or an Organization. Some real-time applications are OpenERP, Tryton, Picalo, etc.

» Image Processing Application:

Python contains many libraries that are used to work with the image. The image can be manipulated according to our requirements. Some libraries of image processing are OpenCV, Pillow, SimpleITK

Python Virtual Machine

Python Virtual Machine (PVM) is a program which provides programming environment. The role of PVM is to convert the byte code instructions into machine code so the computer can execute those machine code instructions and display the output. Interpreter converts the byte code into machine code and sends that machine code to the computer processor for execution. PVM is also called Python Interpreter and this is the reason Python is called an Interpreted language.



» When we run a Python program, two steps happen

» The code gets converted to another representation called 'Byte Code'

» 'Byte Code' gets converted to Machine Code (which is understandable by the computer)

» The second step is being done by PVM or Python Virtual Machine.

» So PVM is nothing but a software/interpreter that converts the byte code to machine code for given operating system.

» We can't see the Byte Code of the program because this happens internally in memory.

» To get the byte code use –m D:\> python -m Sample.py

» Creates a pyc file consisting Bytecode

Python Keywords

Keywords are the reserved words in Python. They cannot be used as 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 3.3. This number can vary slightly in course of time. All the keywords generally written in lowercase except True, False and None.

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	

Python Identifiers

Identifier are user defined names which can be given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.

Rules for writing identifiers

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like `myClass`, `var_1` and `print_this_to_screen`, all are valid example.
2. An identifier cannot start with a digit. `1variable` is invalid, but `variable1` is perfectly fine.
3. Keywords cannot be used as identifiers.

   >>> global = 1 SyntaxError: invalid syntax
4. We cannot use special symbols like !, @, #, $, % etc. in our identifier.

   >>> a@ = 0 SyntaxError: invalid syntax
5. Identifier can be of any length.

Naming conventions for Python identifiers

» Class names start with an uppercase letter. All other identifiers start with a lowercase letter.

» Starting an identifier with a single leading underscore indicates that the identifier is private.

» Starting an identifier with two leading underscores indicates a strongly private identifier.

» If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Note: Python is a case-sensitive language. This means, Variable and variable are not the same. Always name identifiers that make sense. While, c = 10 is valid. Writing count = 10 would make more sense and it would be

easier to figure out what it does even when you look at your code after a long gap. Multiple words can be separated using an underscore, this_is_a_long_variable.

Blocks and Indentation

Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation. Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python provides no braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced. Python uses indentation to indicate a block of code.

A code block (body of a function, loop etc.) starts with indentation and ends with the first unintended line. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally, four whitespaces are used for indentation and is preferred over tabs.

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

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

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

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

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

Comments

Comments are very important while writing a program. It describes what's going on inside a program so that a person looking at the source code does not have a hard time figuring it out. You might forget the key details of the program you just wrote in a month's time. So, taking time to explain these concepts in form of comments is always useful. In Python, hash (#) symbol to start writing a comment. It extends up to the newline character. Comments are for programmers for better understanding of a program. Python Interpreter ignores comment.

Eg:

```
#This is a single line comment
# every line should precede with hash symbol
```

Multi-line comments

Python allows create multi line comments using triple quotes, either ''' or """. These triple quotes are generally used for multi-line strings. But they can be used as multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

```
"""This is also a perfect example of multi-line comments
    can extend to any number of lines"""
```

Docstring in Python

Docstring is short for documentation string. It is a string that occurs as the first statement in a module, function, class, or method definition. We must write what a function/class does in the docstring. Triple quotes are used while writing docstrings.

```
Eg:   def double(num):
          """Function to double the value"""
          return 2*num
```

Docstring can be accessed using the attribute __doc__ of the function.

```
>>> print(double.__doc__) # prints the docstring of the function
>>>Function to double the value
```

Python Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. This is handled internally according to the type of value we assign to the variable. The equal sign (=) is used to assign values to variables. A variable is a location in memory used to store some data (value). They are given unique names to differentiate between different memory locations. The rules for writing a variable name are same as the rules for writing identifiers in Python. No need of declaration a variable before using it. In Python, simply assign a value to a variable and it will exist. Not even declaration of the type of the variable is required. This is handled internally according to the type of value we assign to the variable.

Variable assignment

Assignment operator (=) is used to assign values to a variable. Any type of value can be assigned to any valid variable.

```
a = 5
b = 3.2
c = "Hello"
```

the above 5 is an integer assigned to the variable a.

Similarly, 3.2 is a floating-point number and "Hello" is a string (sequence of characters) assigned to the variables b and c respectively.

Built-in types in Python

Every value in Python has a data type. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes. There are various data types in Python.

Python has the following data types built-in by default, in these categories:

Numeric Types	**:**	int, float, complex
Text Type	**:**	str
Sequence Types	**:**	list, tuple, range

Mapping Type : dict

Set Types : set, frozenset

Boolean Type : bool

None Type : NoneType

Numbers

Python supports four different numerical types −

- int (signed integers)
- float (floating point real values)
- complex (complex numbers)

They are defined as int, float and complex class in Python.

We can use the type() function to know which class a variable or a value belongs to and the isinstance() function to check if an object belongs to a particular class.

```
a = 5
print(a, "is of type", type(a))
a = 2.0
print(a, "is of type", type(a))
a = 1+2j
print(a, " is of type", type(a))
```

Integers can be of any length; it is only limited by the memory available. A floating-point number is accurate up to 15 decimal places. Integer and floating points are separated by decimal points. 1 is integer, 1.0 is floating point number. Complex numbers are written in the form, a + bj, where a is the real part and b is the imaginary part.

```
>>> a = 1234567890123456789
>>> print(a)                    # Output: 1234567890123456789
>>> b = 0.1234567890123456789
>>> print(b)                    # Output: 0.12345678901234568          #considers only 15 places
>>> c = 1+2j
>>> print(c)                    # Output:       (1+2j)
```

Strings

String is sequence of Unicode characters. Both single quotes and double quotes to represent strings. Multi-line strings can be denoted using triple quotes, ''' or """. Strings are immutable.

```
Eg:     s = "This is a string"
        s = """a multiline
                can extend multiple lines"""
```

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.

Eg: Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"
print(a[1]) #prints e
```

Lists

Lists are used to store multiple items in a single variable. List is an ordered sequence of items. List items are ordered, changeable, and allow duplicate values. Lists are ordered, means that the items have a defined order, and that order will not change. When a new items is added to a list, the new items will be placed at the end of the list. List items are indexed, the first item has index [0], the second item has index [1] etc. Lists are created using square brackets:

Eg:

```
fruits = ["apple", "banana", "cherry"]
print(thislist)
```

All the items in a list do not need to be of the same type

```
>>> a = [1, 2.2, 'python']
```

Lists are mutable, meaning the list is changeable, meaning that we can change, add, and remove items in a list after it has been created.

```
>>> a = [1,2,3]
>>> a[2]=4
>>> print(a)
>>>[1, 2, 4]
```

Lists allow Duplicates i.e., lists can have items with the same value:

Eg:

```
fruits= ["apple", "banana", "cherry", "apple", "cherry"]
```

Tuples

Tuple is an ordered sequence of items same as list. The only difference is that tuples are immutable. Tuples once created cannot be modified. Tuples are used to write-protect data and are usually faster than list as it cannot change dynamically. It is defined within parentheses () where items are separated by commas.

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 items are ordered, unchangeable, and allow duplicate values. Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Eg:

```
fruits= ("apple", "banana", "cherry")
print(fruits)
>>> t = (5,'program', 1+3j)      #tuples can have different types
```

Set

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. A set is a collection which is unordered, unchangeable, unindexed and do not allow duplicate values. Sets are written with curly brackets.

Eg:

a = {5,2,3,1,4}

print("a = ", a)

Sets cannot have two items with the same value. Duplicate values will be ignored

>>> a = {1,2,2,3,3,3}

>>> print(a) #prints {1, 2, 3}

Note: Since, set are unordered collection, indexing has no meaning. Set items are unchangeable, but you can remove items and add new items

Dictionary

Dictionary is an unordered collection of key-value pairs. Dictionaries are optimized for retrieving data. Dictionaries are used to store data values in key: value pairs. A dictionary is a collection which is ordered (As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered), changeable and do not allow duplicates.

Dictionaries are defined within braces {} with each item being a pair in the form key: value. Key and value can be of any type. 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. Dictionaries cannot have two items with the same key:

>>> d = {“rollNo”:1234,”name”: ”ramu”}

To access the items in the dictionary use key i.e., d[key] returns values associated with the specified key

d[“rollNo”] #returns 1234

NoneType

NoneType in python used to define null value. Python doesn’t have null value instead its has None. None is keyword is used to define a null value, or no value at all. None is not the same as 0, False, or an empty string.

Eg: x = None

print(x)

Getting the data Type – type()

In order to get the data type of any object type () function is used. type() method returns class type of the argument(object) passed as parameter. type() function is mostly used for debugging purposes.

Syntax: type(object)

Eg: L=[10,20,30]

print(type(L)) # returns <class ‘list’>

Data Type Conversion

It is possible convert between different data types by using different type conversion functions like int(), float(), complex(), str() etc.

Eg:

x = 1 # int

y = 2.8 # float

z = 1j # complex

#convert from int to float:

```
a = float(x)
#convert from float to int:
b = int(y)
#convert from int to complex:
c = complex(x)
```

Note: You cannot convert complex numbers into another number type.

It is also possible to convert one sequence to another.

```
L=[1,2,3]
T=(5,6,7)
S={10,20,30}
#convert from list to tuple:
a = tuple(L)
#convert from tuple to set:
b = set(T)
#convert from set to list:
c = list(S)
```

Python Operators:

Operators are special symbols in Python that carry out arithmetic or logical computation. Operators are used to perform operations on variables and values. The value that the operator operates are called as Operand. Python divides the operators in the following groups:

- » Arithmetic operators
- » Comparison (Relational) operators
- » Logical operators
- » Bitwise operators
- » Assignment operators
- » Identity operators
- » Membership operators

Arithmetic operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication etc. These operators are used to build arithmetic expressions the result of these expressions is always numeric.

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)

Eg: # Python script to illustrate Arithmetic operators in Python

```
x = 15
y = 4
print('x + y =',x+y)
print('x - y =',x-y)
print('x * y =',x*y)
print('x / y =',x/y)
print('x // y =',x//y)
print('x ** y =',x**y)
```

Output:

```
x + y = 19
x - y = 11
x * y = 60
x / y = 3.75
x // y = 3
x ** y = 50625
```

Comparison (Relational) operators

Comparison operators are used to compare values. These operators are used to construct relational expressions which are used in control structures. The result of relational expressions is either True or False.

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

Eg: # Python script to illustrate Comparison operators in Python

```
x = 10
y = 12
print('x > y  is',x>y)
print('x < y  is',x<y)
print('x == y is',x==y)
print('x != y is',x!=y)
print('x >= y is',x>=y)
print('x <= y is',x<=y)
```

Output:

```
x>y is False
x<y is True
x==y is False
x!=y is True
x>=y is False
x<=y is True
```

Logical operators

Logical operators are used to combine more than one relational expression i.e., to form complex conditions. The and, or are binary operators where as the not is unary operator.

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

Truth table for and, or

A	B	A and B	A or B
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

Eg:# Python script to illustrate Logical Operators in Python

```
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

Bitwise operators

Bitwise operators operate on the operands at bit level. These are classified into two categories

- » Bitwise Logical Operators – Bitwise AND (&), Bitwise OR (|), Bitwise exclusive OR (^), Bitwise NOT (~)
- » Bitwise Shift Operators – Bitwise Left shift (<<), Bitwise Right shift (>>)

Bitwise logical operators are evaluated as follows

A	B	A & B	A | B	A^B
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Bitwise Shift Operators

- » The << left shift operators shifts the number to left wards by specified number of bits. The result of the number is multiplied by the 2 raised to number of places shifted

  i.e, a<<n → $a=a*2^n$

  Eg: a =8 a<< 1 → $a=a*2^1$ →16
- » The >> right shift operators shifts the number to right wards by specified number of bits. The result of the number is multiplied by the 2 raised to number of places shifted

  i.e, a<<n → $a=a / 2^n$

  Eg: a =8 a>> 3 → $a=a / 2^3$ → 1

Let x = 10 (0000 1010 in binary) and y = 4 (0000 0100 in binary)

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)

Assignment operators

Assignment operators are used in Python to assign values to variables. Python has only one assignment operator (=). Along with it a set of operators are defined called as short hand assignment operators. There are classified into two categories

- Arithmetic Shorthand assignment operators
- Bitwise Shorthand assignment operators

Operator	Example	Equivalent to		Operator	Example	Equivalent to
+=	x += 5	x = x + 5		&=	x &= 5	x = x & 5
-=	x -= 5	x = x - 5		|=	x |= 5	x = x | 5
*=	x *= 5	x = x * 5		^=	x ^= 5	x = x ^ 5
/=	x /= 5	x = x / 5		>>=	x >>= 5	x = x >> 5
%=	x %= 5	x = x % 5		<<=	x <<= 5	x = x << 5
//=	x //= 5	x = x // 5				
**=	x **= 5	x = x ** 5				

Identity operators

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. Two variables that are equal does not imply that they are identical.

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

Eg: #Python script to illustrate Identity operators in Python

```
x1 = 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)
```

Output:

```
False
True
False
```

Note: x1 and y1 are integers of same values, so they are equal as well as identical. Same is the case x2 and y2 (strings).

Whereas x3 and y3 are lists. They are equal but not identical. Since list are mutable (can be changed), interpreter locates them separately in memory although they are equal.

Membership operators

in 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). In a dictionary we can only test for presence of key, not the value.

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

Eg: # Python script to illustrate Membership operators in Python

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

Output:

```
True
True
True
False
```

Here, H is in x but “hello” is not in x, (remember, Python is case sensitive). Similarly, 1 key and ‘a’ is the value in dictionary y. Hence, ’a’ in y returns False.

Python Statements

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

Assignment Statement:

Python assignment statements to assign objects to names. The target of an assignment statement is written on the left side of the equal sign (=), and the object on the right can be an arbitrary expression that computes an object.

There are some important properties of assignment in Python:-

- Assignment creates object references instead of copying the objects.
- Python creates a variable name the first time when they are assigned a value.
- Names must be assigned before being referenced.
- There are some operations that perform assignments implicitly.

Assignment statement forms:

1. Basic form: This form is the most common form.

   Eg:student = 'SSBN'

   print(student) # prints SSBN

2. Tuple assignment: Python allows you to assign values to multiple variables in one line

   Eg: x, y = 50, 100 # equivalent to: (x, y) = (50, 100)

   print('x = ', x) # prints x = 50

   print('y = ', y) #prints y = 100

   Here the tuple on the left side of the =, Python pairs objects on the right side with targets on the left by position and assigns them from left to right. Therefore, the values of x and y are 50 and 100 respectively.

3. List assignment: This works in the same way as the tuple assignment.

   Eg: [x, y] = [2, 4]

   print('x = ', x) # prints x = 50

   print('y = ', y) #prints y = 100

4. Sequence assignment: In recent version of Python, tuple and list assignment have been generalized into instances of what we now call sequence assignment – any sequence of names can be assigned to any sequence of values, and Python assigns the items one at a time by position.

Eg: a, b, c = 'HEY'

print('a = ', a)

print('b = ', b)

print('c = ', c)

OUTPUT

a = H
b = E
c = Y

5. Extended Sequence unpacking: It allows us to be more flexible in how we select portions of a sequence to assign.

Eg: p, *q = 'Hello'

print('p = ', p)

print('q = ', q)

OUTPUT

p = H
q = ['e', 'l', 'l', 'o']

Here, p is matched with the first character in the string on the right and q with the rest. The starred name (*q) is assigned a list, which collects all items in the sequence not assigned to other names. This is especially handy for a common coding pattern such as splitting a sequence and accessing its front and rest part.

ranks = ['A', 'B', 'C', 'D']

first, *rest = ranks

print("Winner: ", first)

print("Runner ups: ", ', '.join(rest))

OUTPUT

Winner: A

Runner ups: B, C, D

6. Multiple- target assignment: Python allows to assign the same value to multiple variables in one line. Python assigns a reference to the same object (the object which is rightmost) to all the target on the left

Eg: x = y = 75

print(x, y) # prints 75 75

7. Augmented assignment: The augmented assignment is a shorthand assignment that combines an expression and an assignment.
Eg: x = 2

x += 1 # equivalent to: x = x + 1

print(x) # prints 3

There are several other augmented assignment forms: -=, **=, &=, etc.

Print statement

The Print statement actually is Python function used to output data to the standard output device (screen). Print () can also be used to output data to a file.

Syntax of the print() function:

print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Here, objects are the value(s) to be printed.

The sep separator is used between the values. It defaults into a space character.

After all values are printed, end is printed. It defaults into a new line.

The file is the object where the values are printed and its default value is sys.stdout (screen).

Eg: print('This sentence is output to the screen') # prints This sentence is output to the screen

a = 5

print('The value of a is', a) # prints The value of a is 5

```
print(1,2,3,4)                  # prints: 1 2 3 4
print(1,2,3,4,sep='*')          #prints : 1*2*3*4
print(1,2,3,4,sep='#',end='&')  #prints: 1#2#3#4&
```

Output formatting

To format the output so as to make it look attractive, str. format () method is used.

Eg: x = 5; y = 10

```
print('The value of x is {} and y is {}'.format(x,y))
```

The value of x is 5 and y is 10

Here the curly braces {} are used as placeholders and the variables are printed in the place holders as they appear in format().

Eg:
```
print('I love {0} and {1}'.format('bread','butter'))  #prints: I love bread and butter
print('I love {1} and {0}'.format('bread','butter'))  #prints: I love butter and bread
```

it possible to use keyword arguments to format the string.

Eg:
```
print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name = 'John'))
# prints Hello John, Goodmorning
```

it possible to format strings like the old printf() style used in C language. The % (format specifier) is used to accomplish this.

Eg:
```
x = 12.3456789
print('The value of x is %3.2f' %x)  # prints The value of x is 12.35
print('The value of x is %3.4f' %x)  # prints The value of x is 12.3457
```

Input Statement

Developers often have a need to interact with users, either to get data or to provide some sort of result. Most programs today use a dialog box as a way of asking the user to provide some type of input. While Python provides us with a built function to read the input from the keyboard. The input() function takes the input from the user and convert it into string. Type of the returned object always will be <type 'str'>

Syntax input(prompt)

Here prompt A String, representing a default message before the input it is optional.

Eg:
```
num = input('Enter a number: ')
print(num)
```
Output :
```
Enter a number: 10
num '10'
```

Here the entered value 10 is a string, not a number. To convert this into a number use int() or float() functions. This same operation can be performed using the eval() function. But it takes it further. It can evaluate even expressions, provided the input is a string

Eg `eval('2+3')` # prints 5

Import Statement

Import in python is similar to #include header file in C/C++. Python modules can get access to code from another module by importing the file/function using import. A module is a file containing Python definitions and statements. Python modules have a filename and end with the extension .py. Definitions inside a module can be imported to another module or the interactive interpreter in Python. The import keyword is used to accomplish this.

Syntax: import module_name

Eg: import the math module by typing in import math.

```
import math
print(math.pi)
```

Now all the definitions inside math module are available. It is also allows to import some specific attributes and functions only, using the from keyword.

Eg:

```
from math import pi
print(pi)          # prints 3.141592653589793
```

Lists

Python offers a range of compound data types often referred to as sequences. List is one of the most frequently used and very versatile data type used in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

Creating a list:

In Python programming, a list is created by placing all the items (elements) inside a square bracket [ ], separated by commas. It can have any number of items and they may be of different types (integer, float, string etc.).

```
# empty list
my_list = []
# list of integers
my_list = [1, 2, 3]
# list with mixed datatypes
my_list = [1, "Hello", 3.4]
```

Also, a list can even have another list as an item. This is called nested list.

```
# nested list
my_list = ["mouse", [8, 4, 6], ['a']]
```

Indexing:

To access the elements in the list index is used. Index is an integer representing the position of the element in the List. Index always starts from 0 and ends at n-1. i.e., a list having 5 elements will have index from 0 to 4.

Trying to access an element beyond the upper bound(n-1) will raise an IndexError. The index must be an integer can't use float or other types; this will result into TypeError.

Nested list are accessed using nested indexing.

Eg:

```
my_list = ['p','r','o','b','e']
print(my_list[0])          # Output: p
print(my_list[2])          # Output: o
```

```
print(my_list[4])          # Output: e
print(my_list[4.0])        # Error! Only integer can be used for indexing
# Nested List
n_list = ["Happy", [2,0,1,5]]
print(n_list[0])           # Output: Happy
print(n_list[0][1])        # Output: a
print(n_list[1][3])        # Output: 5
```

Negative indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
Eg:  my_list = ['p','r','o','b','e']
     print(my_list[-1])    # Output: e
     print(my_list[-5])    # Output: p
```

Slicing lists:

Slicing operator (:) is used to get range of indexes. When specifying a range, the return value will be a new list with the specified items.

Syntax: listName [start : end]

here start is inclusive and end is exclusive

```
Eg:  fruits=["apple","banana","cherry","orange","kiwi","melon","mango"]
     fruits[2:5]           # returns [“cherry”,”orange”,”kiwi”]
     If start is skipped then returns items from beginning
     fruits[:2]            #returns [“apple”,”banana”]
     If end is skipped then returns items till end .
     fruits[4:]            #returns [“kiwi”,”melon”,”mango”]
     Slicing allows negative indexing also
     fruits[-6:-3]         # returns ["banana","cherry","orange"]
```

Updating Lists:

Python allows to update single or multiple elements of lists by giving the value on the left-hand side of the assignment operator (=) to change an item or a range of items.

```
Eg:  odd = [2, 4, 6, 8]
     # change a single element
     odd[0] = 1
     print(odd)            # Output: [1, 4, 6, 8]
     # change multiple elements
     odd[1:4] = [3, 5, 7]
     print(odd)            # Output: [1, 3, 5, 7]
```

Delete list elements:

Python allows to delete one or more items from a list using the keyword del. It can even delete the list entirely. List items can also be deleted by assigning an empty list to a slice of elements.

Eg:

```
my_list = ['p','r','o','b','l','e','m']
# delete one item
del my_list[2]
print(my_list)          # Output: ['p', 'r', 'b', 'l', 'e', 'm']
# delete multiple items
del my_list[1:5]
print(my_list)          # Output: ['p', 'm']
# delete entire list
del my_list
print(my_list)          # Error: List not defined
my_list = ['p','r','o','b','l','e','m']
my_list[2:3] = []
print(my_list)          # Output: ['p', 'r', 'b', 'l', 'e', 'm']
 my_list[2:5] = []
print(my_list)          # Output: ['p', 'r', 'm']
```

Basic List Operations

Following are basic operations that can be performed on the Lists

Python Expression	**Description**	**Results**
len([1, 2, 3])	Length	3
[1, 2, 3] + [4, 5, 6]	Concatenation	[1, 2, 3, 4, 5, 6]
['Hi!'] * 4	Repetition	['Hi!', 'Hi!', 'Hi!', 'Hi!']
3 in [1, 2, 3]	Membership	True
for x in [1, 2, 3]: print x,	Iteration	1 2 3

The + operator is used to combine two lists. This is also called concatenation.

```
A=[10,20,30,40]
B=[30,40,50,60]
print(A+B)              # Output: [10,20,30,40,30,40,50,60]
```

The * operator repeats a list for the given number of times.

```
print(A*2)              # Output: [10,20,30,40,10,20,30,40]
```

List Comprehension

Python List comprehensions are used for creating new lists from other iterables like tuples, strings, arrays, lists, etc. A list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element.

Syntax:

newList = [expression(element) for element in oldList if condition]

Example:

```
# Python program to demonstrate list comprehension in Python

# below list contains square of all odd numbers from range 1 to 10

odd_square = [x ** 2 for x in range(1, 11) if x % 2 == 1]

print(odd_square)
```

List Methods

Python provides methods that can be used with list object. These methods are accessed as list. Method ().

Function	**description**
append()	Add an element to the end of the list
extend()	Add all elements of a list to another list
insert()	Insert an item at the defined index
remove()	Removes an item from the list
pop()	Removes and returns an element at the given index
clear()	Removes all items from the list
index()	Returns the index of the first matched item
count()	Returns the count of number of items passed as an argument
sort()	Sort items in a list in ascending order
reverse()	Reverse the order of items in the list
copy()	Returns a shallow copy of the list

Built-in Functions with List

Python provides Built-in functions those can be used with lists to perform different tasks.

Function	**Description**
all()	Return True if all elements of the list are true (or if the list is empty).
any()	Return True if any element of the list is true. If the list is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of list as a tuple.
len()	Return the length (the number of items) in the list.
list()	Convert an iterable (tuple, string, set, dictionary) to a list.
max()	Return the largest item in the list.
min()	Return the smallest item in the list
sorted()	Return a new sorted list (does not sort the list itself).
sum()	Return the sum of all elements in the list.

Tuples

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets. Tuple items are ordered, unchangeable, and allow duplicate values. Tuple items are indexed.

Eg: fruits = ("apple", "banana", "cherry")

Tuples can be empty or can have different types or can have duplicates

Eg: names=()

values=(10,”ramu”,35.369,False,10,20,30,”ramu”,False)

Advantages of Tuple over List:

Since, tuples are quite similar to lists, both of them are used in similar situations as well. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- Generally, tuples are used for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.
- Tuple consume less memory as compared to the list

List vs Tuple

LIST	**TUPLE**
Lists are mutable	Tuples are immutable
Implication of iterations is Time-consuming	The implication of iterations is comparatively Faster
The list is better for performing operations, such as insertion and deletion.	Tuple data type is appropriate for accessing the elements
Lists consume more memory	Tuple consume less memory as compared to the list
Lists have several built-in methods	Tuple does not have many built-in methods.
The unexpected changes and errors are more likely to occur	In tuple, it is hard to take place.

Creating a Tuple:

Creating a tuple is as simple as putting different comma-separated values. Optionally put these comma-separated values between parentheses also. A tuple can have any number of items and they may be of different types (integer, float, list, string etc.)

Eg: # empty tuple

my_tuple = ()

print(my_tuple) # Output: ()

tuple having integers

```
my_tuple = (1, 2, 3)
print(my_tuple)                          # Output: (1, 2, 3)
# tuple with mixed datatypes
my_tuple = (1, "Hello", 3.4)
print(my_tuple)                          # Output: (1, "Hello", 3.4)
# nested tuple
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)                          # Output: ("mouse", [8, 4, 6], (1, 2, 3))
# tuple can be created without parentheses
my_tuple = 3, 4.6, "dog"
print(my_tuple)                          # Output: 3, 4.6, "dog"
```

Unpacking Tuples

“packing” a tuple means assigning values to the tuple while creation

Eg: fruits = ("apple", "banana", "cherry")

But Python also allows to extract the values back into variables.

This is called "unpacking“

```
Eg:     fruits = ("apple", "banana", "cherry")
        (a, b, c) = fruits
        print(a)            #returns apple
        print(b)                #returns banana
        print(c)            #returns cherry
```

The number of variables must match the number of values in the tuple, if not, you must use an asterisk to collect the remaining values as a list.

```
Eg:     fruits=("apple", "banana", "cherry", "strawberry", "raspberry")
        (a, b, *c) = fruits
        print(a)            #returns apple
        print(b)            #retruns banana
        print(c)        #retruns ['cherry', 'strawberry', 'raspberry']
```

Accessing Elements:

Indexing

To access the elements in the tuple index is used. Index is an integer representing the position of the element in the tuple. Index always starts from 0 and ends at n-1. i.e., a tuple having 5 elements will have index from 0 to 4.

Trying to access an element beyond the upper bound(n-1) will raise an IndexError. The index must be an integer can't use float or other types; this will result into TypeError.

Nested tuples are accessed using nested indexing.

```
Eg:     my_tuple = ('p','r','o','b','e')
        print(my_tuple [0])         # Output: p
```

```
print(my_tuple [2])          # Output: o
print(my_tuple [4])          # Output: e
print(my_tuple[4.0])         # Error! Only integer can be used for indexing
# nested Tuples
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(n_tuple[0][3])         # Output: 's'
print(n_tuple[1][1])         # Output: 4
```

Negative Indexing

Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
Eg:   my_tuple = ('p','e','r','m','i','t')
      print(my_tuple[-1])     # Output: 't'
      print(my_tuple[-6])     # Output: 'p'
```

Slicing Tuples

Slicing operator (:) is used to get range of indexes. When specifying a range, the return value will be a new tuple with the specified items.

```
Syntax :   tupleName [ start : end ]
           here start is inclusive and end is exclusive
Eg:   fruits=("apple","banana","cherry","orange","kiwi","melon","mango")
      fruits[2:5]          # returns (“cherry”,”orange”,”kiwi”)
      If start is skipped then returns items from beginning
      fruits[:2]           #returns (“apple”,”banana”)
      If end is skipped then returns items till end .
      fruits[4:]           #returns (“kiwi”,”melon”,”mango”)
      Slicing allows negative indexing also
      fruits[-6:-3]        # returns ("banana","cherry","orange")
```

Note: Unlike lists, tuples are immutable. This means that elements of a tuple cannot be changed once it has been assigned. But, if the element is itself a mutable datatype like list, its nested items can be changed.

```
my_tuple = (4, 2, 3, [6, 5])
my_tuple[3][0] = 9
print(my_tuple)              # Output: (4, 2, 3, [9, 5])
```

Also, tuples can be reassigned

```
my_tuple = ('p','r','o','g','r','a','m','i','z')
print(my_tuple)              # Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

Delete Tuple Elements

Removing individual tuple elements is not possible. But entire tuple can be deleted using del keyword

Syntax: del tupleName

Eg: values=(10,20,30,40,50)

del values

print(values) #NameError: name 'values' is not defined

Basic Tuples Operations:

Following are basic operations that can be performed on the Tuples

Python Expression	**Results**	**Description**
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Python Tuple Methods

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

- **count(x)**: Return the number of items that is equal to x

  Eg: name = ('a','p','p','l','e',)

  name.count('p') # Output: 2

- **index(x)**: Return index of first item that is equal to x. if no match is found then error is returned

  Eg: name = ('a','p','p','l','e',)

  name.index('p') # Output: 1

Function	**Description**
all()	Return True if all elements of the tuple are true (or if the tuple is empty).
any()	Return True if any element of the tuple is true. If the tuple is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of list as a tuple.
len()	Return the length (the number of items) in the tuple.
tuple()	Convert an iterable (list, string, set, dictionary) to a tuple.
max()	Return the largest item in the tuple.
min()	Return the smallest item in the tuple
sorted()	Return a new sorted tuple (does not sort the tuple itself).
sum()	Return the sum of all elements in the tuple.

Built-in Functions with Tuple:

Python provides Built-in functions those can be used with tuples to perform different tasks

Strings

A string is a sequence of characters. A character is simply a symbol. For example, the English language has 26 characters. Computers do not deal with characters; they deal with numbers (binary). Even though you

may see characters on your screen, internally it is stored and manipulated as a combination of 0's and 1's. This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encodings used. String is basically a sequence of characters. Technically string is a sequence of Unicode characters in Python. Strings are written in single or double quotes. Strings in Python are immutable i.e., modifying the string is not possible

Creating a String:

Strings can be created by enclosing characters inside a single quote or double quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

```
Eg:     # all of the following are equivalent
        my_string = 'Hello'
        my_string = "Hello"
        my_string = '''Hello'''
        # triple quotes string can extend multiple lines
         msg = """ Python is a Object oriented Language """“
        msg = ‘’’ Python is a Object oriented  Language ‘’’
```

Indexing:

Individual characters of a string can be accessed using indexing. Index is an integer number representing the position of character in the string it starts from 0 and ends at n-1 (n-length). Trying to access a character out of index range will raise an IndexError. The index must be an integer. We can't use float or other types; this will result into Type Error.

```
Eg:     name= "Python".
        print(name[0])              #returns      P
        print(name[2])              #returns      t
        print(name[7])              # index error occurs
```

Negative Indexing:

Python allows negative indexing on strings. Negative index starts at -1 and go up to –n.

```
Eg:     name= "Python".
        print(name[-1])             #returns      n
        print(name[-2])             #returns      o
```

Slicing:

Slicing operator (:) is used to get range of indexes. When specifying a range, the return value will be a new string with the specified items.

```
        Syntax          stringName [ start : end ]  # start is inclusive and end is exclusive
Eg:     name= "Programming"
        name[2:5] # returns ogr
```

-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
P	r	o	g	r	a	m	m	i	n	g
0	1	2	3	4	5	6	7	8	9	10

If start is skipped then returns items from beginning, If end is skipped then returns items till end. Slicing allows negative indexing also

```
        name[-6:-3]   # returns amm
```

Advanced Slicing:

Syntax stringName [start : end: step] # start is inclusive and end is exclusive and step is increment

Eg: name= "Programming”

[low:high:+ve] – normal order

name[2:10:2] # returns ormi

[high:low:-ve] – reverse order

name[10:2:-1] # returns gnimmarg

name[::1] # returns Programming

name[::-1] # returns gnimmargorP

Changing or deleting a string:

Strings are immutable. This means that elements of a string cannot be changed once it has been assigned. We can simply reassign different strings to the same name.

Eg: my_string = 'programiz'

my_string[5] = 'a' # Output: TypeError: 'str' object does not support item assignment

my_string = 'Python'

my_string # Output: 'Python'

It is not possible to delete or remove characters from a string. But deleting the string entirely is possible using the keyword del.

del my_string[1] # Output: TypeError: 'str' object doesn't support item deletion

del my_string

print(my_string) # Output: NameError: name 'my_string' is not defined

String Basic Operations:

Following are basic operations that can be performed on the Strings.

Python Expression	**Results**	**Description**
len(“Python”)	6	Length
“Python” + “Program”	“PythonProgram”	Concatenation
Py* 4	“PyPyPy”	Repetition
y in “Python”	TRUE	Membership
for x in “Python”: print x	P y t h o n	Iteration

Joining of two or more strings into a single one is called concatenation. The + operator does this in Python. Simply writing two string literals together also concatenates them. And * operator is used for repetition

Eg: str1 = “Python”

str2 ='Programming'

print(str1 + str2) # Output: PythonProgramming

print(str1 * 3) # Output: PythonPythonPython

String Class Methods:

Python provides multiple methods in String class which are used manipulate strings.

Method	Description
count()	Returns the number of times a specified value occurs in a string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isupper()	Returns True if all characters in the string are upper case
join()	Converts the elements of an iterable into a string
lower()	Converts a string into lower case
replace()	Returns a string where a specified value is replaced with a specified value
split()	Splits the string at the specified separator, and returns a list
strip()	Returns a trimmed version of the string
title()	Converts the first character of each word to upper case
upper()	Converts a string into upper case

String Formatting:

format() method allows you to format selected parts of a string. It takes the passed arguments, formats them, and places them in the string where the placeholders {} are:

Syntax:string.format(value1, value2...)

The placeholders can be identified using named indexes {price}, numbered indexes {0}, or even empty placeholders {}.

Eg:

```
txt1 = "Python {name}, is {ver}".format(name = "Language", ver = 3.10)   #Output: Python Language is 3.10
txt2 = "Python {0}, is {1}".format("Language",3.10)                       #Output: Python Language is 3.10
txt3 =  "Python {}, is {}".format("Language",3.10)                        #Output: Python Language is 3.10
```

Note:

The format () method can have optional format specifications. They are separated from field name using colon. Eg: To left-justify <, right-justify > or center ^ a string in the given space. Also format () allows to format integers as binary, hexadecimal etc. and floats can be rounded or displayed in the exponent format

Escape Sequences:

To insert characters that are illegal in a string, use an escape character. An escape character is a backslash \ followed by the character you want to insert. An example of an illegal character is a double quote inside a string that is surrounded by double quotes. Here is a list of all the escape sequence supported by Python.

Code	Result
\'	Single Quote
\\	Backslash
\n	New Line
\t	Tab
\b	Backspace
\ooo	Octal value
\xhh	Hex value

Eg: print("C:\\Python32\\Lib") # Output: C:\Python32\Lib

print("This is printed\nin two lines") # Output: This is printed in two lines

print("This is \x48\x45\x58 ") # Output: This is HEX

Note:

In order to ignore the escape sequences inside a string just prefix r or R in front of the string. This will imply that it is a raw string and any escape sequence inside it will be ignored.

Eg: print("This is \x61 \ngood example") # Output: This is a good example

print(r"This is \x61 \ngood example") # Output: This is \x61 \ngood example

Sets:

Sets are used to store multiple items in a single variable. In Python programming, a set is created by placing all the items (elements) inside a curly braces { }. Set is a collection which is unordered, unchangeable, unindexed and do not allow duplicate values. Every element in a set is unique (no duplicates) and must be immutable (which cannot be changed). However, the set itself is mutable. Python allows to add or remove items using methods. Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

Creating a set:

A set is created by placing all the items (elements) inside curly braces {}, separated by comma or by using the built-in function set(). It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have a mutable element, like list, set or dictionary, as its element.

Eg: my_set = {1, 2, 3} # set of integers

my_set = {1.0, "Hello", (1, 2, 3)} # set of mixed datatypes

Creating an empty set:

Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements we use the set() function without any argument.

Eg:

```
# initialize a with {}
a = {}
# check data type of a
print(type(a))  # Output: <class 'dict'>
```

```
# initialize a with set()
a = set()
# check data type of a
print(type(a))  # Output: <class 'set'>
```

Indexing:

Sets are unordered, indexing have no meaning. It not possible to access or change an element of set using indexing or slicing. Set does not support it

Eg: names = {"apple", "banana", "cherry“}

names[0] # TypeError: 'set' object is not subscriptable

In order to access the elements in the Set generally for loop is used

```
Eg:         for x in names:
                print(x, end=” “)          # Output: cherry, apple, banana
```

Since indexing is not supported even slicing operator cannot be used on sets

Removing elements from a set:

A particular item can be removed from set using methods, discard() and remove(). The only difference between the two is that, while using discard() if the item does not exist in the set, it remains unchanged. But remove() will raise an error in such condition.

```
Eg:     my_set = {1, 3, 4, 5, 6}
        my_set.discard(4)           # discard an element
        print(my_set)               # Output: {1, 3, 5, 6}
        my_set.remove(6)            # remove an element
        print(my_set)               # Output: {1, 3, 5}
```

Similarly, an element can be removed and returned an item using the pop() method. Set being unordered, there is no way of determining which item will be popped. It is completely arbitrary. To remove all items from a set clear() is used.

Set Class Methods

Set class has the following methods which can be used to manipulate set objects

Method	**Description**
add()	Add an element to a set
clear()	Remove all elements form a set
copy()	Return a shallow copy of a set
difference()	Return the difference of two or more sets as a new set
difference_update()	Remove all elements of another set from this set
discard()	Remove an element from set if it is a member. (Do nothing if the element is not in set)
intersection()	Return the intersection of two sets as a new set
intersection_update()	Update the set with the intersection of itself and another
isdisjoint()	Return True if two sets have a null intersection
issubset()	Return True if another set contains this set
issuperset()	Return True if this set contains another set
pop()	Remove and return an arbitary set element. Raise KeyError if the set is empty

remove()	Remove an element from a set. If the element is not a member, raise a KeyError
symmetric_difference()	Return the symmetric difference of two sets as a new set
symmetric_difference_update()	Update a set with the symmetric difference of itself and another
union()	Return the union of sets in a new set
update()	Update a set with the union of itself and others

Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. These operations can be performed with operators or methods.

Let us consider the following two sets for the following operations.

Eg: A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

» **Set Union:**

Union of A and B is a set of all elements from both sets. Using " | " operator also does the same

Eg:
```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
# use | operator
print(A | B)                    # Output: {1, 2, 3, 4, 5, 6, 7, 8}
# use union function
print(A.union(B))               # Output:  {1, 2, 3, 4, 5, 6, 7, 8}
```

» **Set Intersection:**

Intersection of A and B is a set of elements that are common in both sets. Intersection is performed using & operator. Same can be accomplished using the method intersection ().

Eg:
```
# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

# use & operator

print(A & B)                    # Output: {4, 5}

# use intersection function on A

print(A.intersection(B))            # Output: {4, 5}
```

» **Set Difference**

Difference of A and B (A - B) is a set of elements that are only in A but not in B. Similarly, B - A is a set of elements in B but not in A. Difference is performed using - operator. Same can be accomplished using the method difference ().

Eg:
```
# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

# use - operator on A

print(A - B)                    # Output: {1, 2, 3}

# use difference function on A
```

```
print(A.difference(B))          # Output: {1, 2, 3}
# use - operator on B
print(B – A)                    # Output: {8, 6, 7}
# use difference function on B
print(B.difference(A))          # Output: {8, 6, 7}
```

» **Set Symmetric Difference:**

Symmetric Difference of A and B is a set of elements in both A and B except those that are common in both. Symmetric difference is performed using ^ operator. Same can be accomplished using the method symmetric_difference().

Eg:

```
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
# use ^ operator
print(A ^ B)                    # Output: {1, 2, 3, 6, 7, 8}
# use symmetric_difference function on A
print( A.symmetric_difference(B))  # Output: {1, 2, 3, 6, 7, 8}
# use symmetric_difference function on B
B.symmetric_difference(A)       # Output: {1, 2, 3, 6, 7, 8}
```

Built-in Functions with Set

Python provides Built-in functions those can be used with sets to perform different tasks

Function	Description
	Return True if all elements of the set are true (or if the set is empty).
any()	Return True if any element of the set is true. If the set is empty, return False.
enumerate()	Return an enumerate object. It contains the index and value of all the items of set as a pair.
len()	Return the length (the number of items) in the set.
max()	Return the largest item in the set.
min()	Return the smallest item in the set.
sorted()	Return a new sorted list from elements in the set(does not sort the set itself).
set()	Converts an iterable (list, tuple, string etc.,) to set object
sum()	Retrun the sum of all elements in the set.

Frozenset

Frozenset is the same as set except the frozensets are immutable which means that elements from the frozenset cannot be added or removed once created. While tuples are immutable lists, frozensets are immutable sets.

Frozensets can be created using the function frozenset (). The frozenset () is an inbuilt function in Python which takes an iterable object as input and makes them immutable. Simply it freezes the iterable objects and makes them unchangeable. This function takes input as any iterable object and converts them into an immutable object. The order of elements is not guaranteed to be preserved.

Syntax: frozenset(iterable_object_name)

Parameter: This function accepts iterable object as input parameter.

Return Type: This function returns an equivalent frozenset object.

Frozenset supports methods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric_difference() and union(). Being immutable it does not have method that add or remove elements.

```
Eg:   A = frozenset([1, 2, 3, 4])
      B = frozenset([3, 4, 5, 6])
      A.isdisjoint(B)          # Output: False
      A.difference(B)          # Output: frozenset({1, 2})
      A | B                    # Output: frozenset({1, 2, 3, 4, 5, 6})
      A.add(3)                 # AttributeError: 'frozenset' object has no attribute 'add'
```

Dictionary

Dictionaries are used to store data values in key: value pairs. A dictionary is a collection which is ordered (From Python version 3.7, dictionaries are ordered earlier, dictionaries are unordered.), changeable and do not allow duplicates.

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: { }.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Creating a Dictionary:

Creating a dictionary is as simple as placing items inside curly braces { } separated by comma or even using the dict () function. An item has a key and the corresponding value expressed as a pair, key: value. While values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

```
Eg:   my_dict = { }                          # empty dictionary
      my_dict = {1: 'apple', 2: 'ball'}      # dictionary with integer keys
      my_dict = {'name': 'John', 1: [2, 4, 3]}# dictionary with mixed keys
      my_dict = dict ({1:'apple', 2:'ball'}) # using dict ()
      my_dict = dict([(1,'apple'), (2,'ball')]) # from sequence having each item as a pair
```

Accessing Values in Dictionary:

In order to access dictionary elements dictionary name followed by square brackets along with the key name to obtain its value.

```
      Syntax:dictionaryName [ “keyName”]        # returns the value associated with the keyName
Eg:   details = {'Name': Ishanvi, 'Age': 5, 'Class': 'First'}
      print (details['Name'])      # Output: Ishanvi
      print ( details['Age'])      # Output: 5
      print(details['Class'])      # Output: First
```

print(details[‘dob’]) # Throws KeyError: 'dob'

Updating Dictionary

Python allows to update the value of a specific item by referring to its key name. It allows to add a new key-value pair, modifying an existing entry, or deleting an existing entry.

Ex: details={"Name":"Ishanvi","Age":5,"Class":"First"}

details[“age”]=4 # update value

print(details) # Output: details={"Name":"Ishanvi","Age":4,"Class":"First"}

details[“city”] = “Anantapur” # add a new key: value pair

print(details) # Output: details={"Name":"Ishanvi","Age":4,"Class":"First", ”city”:”Anantapur”}

Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary keys −

- More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins. For example −

  Eg: dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}

  print "dict['Name']: ", dict['Name'] # Output: dict['Name']: Manni

- Keys must be immutable. i.e., strings, numbers or tuples are allowed as dictionary keys but something like ['key'] is not allowed.

  Eg: dict = {['Name']: 'Zara', 'Age': 7} # TypeError: unhashable type: 'list'

Built-in functions with Dictionary

Python provides Built-in functions those can be used with Dictionary to perform different tasks

Function	**Description**
cmp(dict1, dict2)	Compares elements of both dict.
len(dict)	Returns length of the dictionary. This would be equal to the number of items in the dictionary.
str(dict)	Produces a printable string representation of a dictionary
type(variable)	Returns the type of the passed variable. i.e., the object type is returned

Dictionary Class Methods

Python provides multiple methods in Dictionary class which can be used to manipulate dictionaries.

Method	**Description**
update()	Updates the dictionary with the specified key-value pairs
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair

clear()	Removes all the elements from the dictionary
get()	Returns the value of the specified key
keys()	Returns a list containing the dictionary's keys
values()	Returns a list of all the values in the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
items()	Returns a list containing a tuple for each key value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value

Remove Elements from Dictionary

Python allows to either remove individual dictionary elements or clear the entire contents of a dictionary. Also allows to delete entire dictionary in a single operation. To explicitly remove an entire dictionary del statement is used.

```
Eg:     squares = {1:1, 2:4, 3:9, 4:16, 5:25}
        squares.pop(4)                  # remove a particular item
        print(squares)          # Output: {1: 1, 2: 4, 3: 9, 5: 25}
        squares.popitem()       # remove an arbitrary item
        print(squares)          # Output: {2: 4, 3: 9, 5: 25}
        del squares[5]          # delete a particular item
        print(squares)          # Output: {2: 4, 3: 9}
        squares.clear()         # remove all items
        print(squares)          # Output: {}
        del squares             # delete the dictionary itself
        print(squares)          # Throws Error
```

Processing Dictionaries using for loops

Generally, for loop is used to process the Dictionaries. When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.

Eg: stu={"name":"ramu","age":25,"class":"mpcs","medium":"english"}

In order to get the keys from the dictionary for loop is as follows

```
#get keys
for x in stu:
      print(x)
```

OR

```
#get keys
for x in stu.keys():
      print(x)
```

In order to get the values from the dictionary for loop is as follows

```
#get values
for x in stu:
      print(stu[x])
```

OR

```
#get values
for x in stu.values():
      print(x)
```

In order to get both the keys and values from the dictionary for loop is as follows

```
#getting both keys and values
for x,y in stu.items():
    print(x,y)
```

Conditional Statements

Conditional statements are used in situations where the order of execution of statements has to be changed based on certain conditions. Conditional statements decide the direction of the flow of program execution. Python has six conditional statements that are used in decision-making

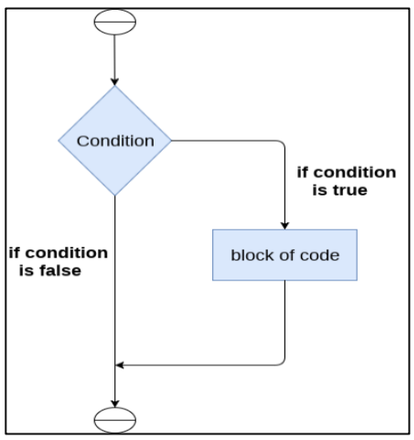
1. if the statement
2. if else statement
3. if…elif ladder
4. Nested if statement
5. Short Hand if statement
6. Short Hand if-else statement

if statement:

Decision making is required when we want to execute a code only if a certain condition is satisfied.

Syntax

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



Here, test expression is a relational expression. The program evaluates the test expression and will execute statement(s) only if the text expression is True. If the text expression is False, the statement(s) is not executed.
In Python, the body of the if statement is indicated by the indentation. Body starts with an indentation and the first unintended line marks the end. Python interprets non-zero values as True. None and 0 are interpreted as False.

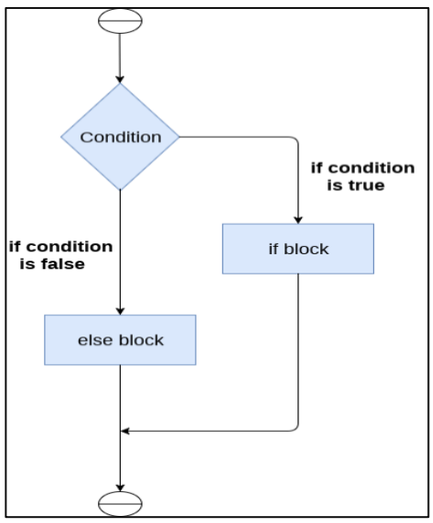
Eg:

```
num=10
if num > 0:
        print(num, " is positive number")
print("Statement after if block")
```

Output:
10 is positive number
Statement after if block

In the above example, num > 0 is the test expression. The body of if is executed only if this evaluates to True. The print() statement falls outside of the if block (unintended). Hence, it is executed regardless of the test expression.

f...else Statement



Syntax:

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

if. Else statement evaluates test expression and will execute body of if only when test condition is True.

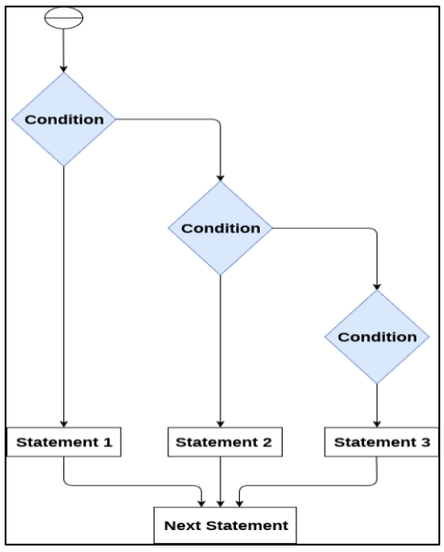
If the condition is False, body of else is executed. Indentation is used to separate the blocks.

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

Output:
a is greater than b

In the above example the expression b>a is evaluated and if it is true then “b is greater than a” is printed otherwise “a is greater than b” is printed.

if...elif...else (elif ladder)



Syntax:

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

The elif is short for else if. It allows us to check for multiple expressions. If the condition for if is False, it checks the condition of the next elif block and so on. If all the conditions are False, body of else is executed. Only one block among the several if...elif...else blocks is executed according to the condition. The if block can have only one else block. But it can have multiple elif blocks.

Eg:
```
a=75
b=25
if(a>b):
   print(a,"is biggest number")
elif a==b:
   print(" both are equal")
else:
   print(b,"is biggest")
print("Statement after if else")
```

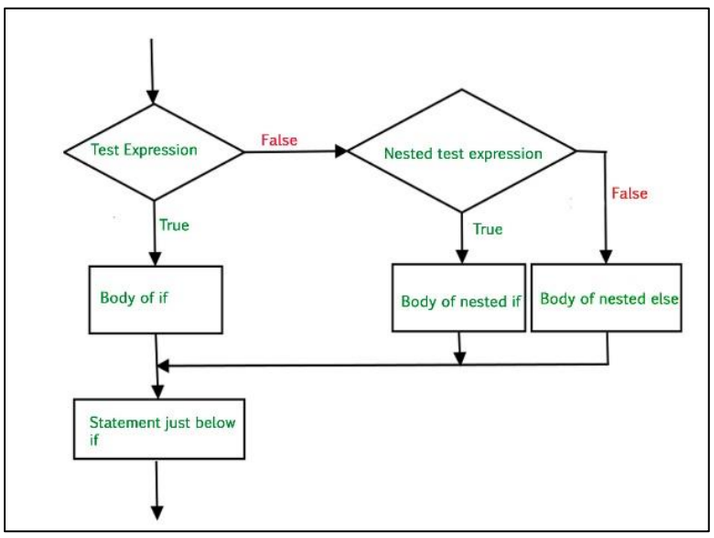
Output:
75 is biggest

In this example the first condition is evaluated if it is false then second condition in the elif is executed if that is true then elif block is executed otherwise else block is executed

Nested if statements

It is possible to have a if...elif...else statement inside another if...elif...else statement. This is called nesting.

Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting.



Syntax:

```
if test expression1:
        if test expression2:
                Body of nested if
        else:
                Body of nested else
else:
   Body of else
```

Eg
```
num = 15
if num >= 0:
        if num == 0:
                print("Zero")
        else:
                print("Positive number")
else:
        print("Negative number")
```

Output:

Positive Number

Shorthand if and if else:

If there is only one statement to execute in if block, then it can be put on the same line as the if statement.

Syntax: if condition: #true block

Eg: if a > b: print("a is greater than b")

If there is only one statement to execute in if block as well as in else block then it can be put on the same line

Syntax: #true block if condition else #false block

Eg: print("A is big ") if a > b else print("B is big")

Note: This technique is known as Ternary Operators, or Conditional Expressions.

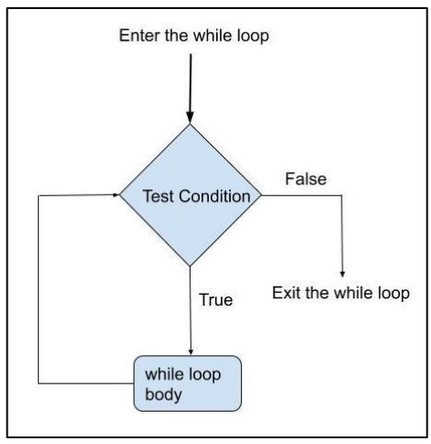
Loops

In computer programming, a loop is a sequence of instructions, often called as body of loop that is continually repeated until a certain condition is reached. Python supports entry control looping.

Python has two primitive loop statements:

- » while loops
- » for loops

While Statement: The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.



Syntax

```
while test_expression:
    Body of while
```

Here the test_expression is a relational expression which yields True or False. First the test expression is evaluated. If the expression yields True then the body of the loop is executed and after successful execution of body of loop again the expression is evaluated and this process continues till the test_expression evaluates to false.

Note:

- Python interprets any non-zero value as True. None and 0 are interpreted as False
- In Python, the body of the while loop is determined through indentation.
- Body starts with indentation and the first unintended line marks the end.
- All the statements in the indentation becomes the part of the while loop.

Eg:

```
i = 1
n=10
sum=0
while i <= n:
sum=sum+i
        i += 1
print("Sum of ", n ,"natural numbers is:", sum)
```

Output:

```
Sum of 10 natural numbers is:55
```

In this example, the test expression will be True as long as the counter variable i is less than or equal to n (i.e,10). While dealing with loops always the loop control variable (i) must be incremented or decremented in order to put an end to the loop otherwise infinite loops (never ending loops) occurs.

While loop with else

While loop executes the block until a condition is satisfied. When the condition becomes false, the statement immediately after the loop is executed. The else clause is only executed when your while condition becomes false. While loop sometimes can be terminated using break statement or if an exception is raises in such cases, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is false.

Eg:

Example to illustrate the use of else statement with the while loop

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

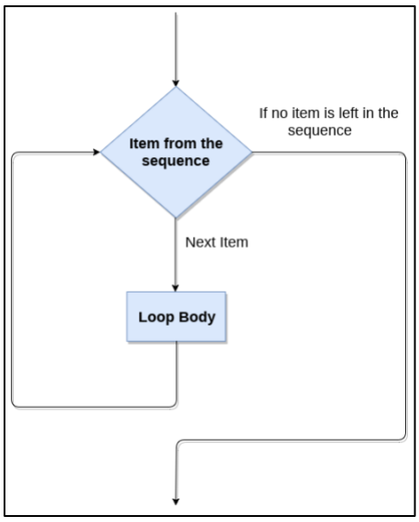
Output

```
1
2
3
4
5
i no longer less than 6
```

when counter i value becomes 6 then, the condition in while becomes False. Hence, the else part is executed.

Loops iterate over a block of code until test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression. The break statement is used in such cases.

For Loop



The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Syntax

for val in sequence:

    Body of for

Here, val is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until val reaches the last item in the sequence. The body of for loop is separated from the rest of the code using indentation. All the statements in the indentation becomes the part of the for loop and the first unintended statement becomes the statement after the for loop.

Eg:

```
    #Looping through a List
fruits = ["apple", "banana", "cherry"]
    for x in fruits:
        print(x)
#Looping through a String
    name= “Python”
    for x in name:
        print(x)
```

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.

Syntax: range (start, stop, step)

Here start is the starting value which is optional (default is 0)

Stop is the end value and is mandatory

Step is also optional which is incrementation from start to end(default is 1)

Note: the start value is always inclusive and end value is always exclusive.

Eg:

```
    range(1,10,1)  returns 1,2,3,4,5,6,7,8,9 but not 10
```

This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go. To force this function to output all the items, use the function list().

Eg:

```
    print(range(10))                    # Output: range(0, 10)
    print(list(range(10)))          # Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
print(list(range(2, 8)))        # Output: [2, 3, 4, 5, 6, 7]
print(list(range(2, 20, 3)))    # Output: [2, 5, 8, 11, 14, 17]
```

range() function can be used in for loops to iterate through a sequence of numbers.

» Step can be optional default is 1

```
Eg: for x in range(0, 6):
        print(x)                    #Output: 0,1,2,3,4,5
```

» Start can be optional default is 0

```
Eg: for x in range(6):
        print(x)                    #Output: 0,1,2,3,4,5
```

» Step can be other than 1

```
Eg: for x in range(0,6,3):
        print(x)                    #Output: 0,3
```

» range() be combined with the len() function to iterate though a sequence using indexing.

```
Eg: # Program to iterate through a list using indexing
    fruits = ['apple','banana','cherry']
    # iterate over the list using index
    for i in range(len(fruits)):
       print(fruits[i])
```

```
Output:
apple
banana
cherry
```

for loop with else

A for loop can have an optional else block as well. else block of code is executed when the for loop is finished but it is not executed when the loop terminates abruptly with an exception inside the loop or use of break statement.

Eg:

```
for x in range(6):
    print(x)
else:
    print("Finally finished!")
```

```
Output:
1
2
3
4
5
Finally finished
```

Here, the for loop executes till range becomes 6. When the for-loop exhausts, it executes the block of code in the else and prints Finally finished.

Nested loops

A nested loop is a loop inside a loop. The "inner loop" will be executed one time for each iteration of the "outer loop":

Eg:

```
#Print each adjective for every fruit:
adj = ["red", "tasty"]
fruits = ["apple", "banana"]
for x in adj:
  for y in fruits:
    print(x, y)
```

```
Output:
red apple
red banana
tasty apple
tasty banana
```

break statement

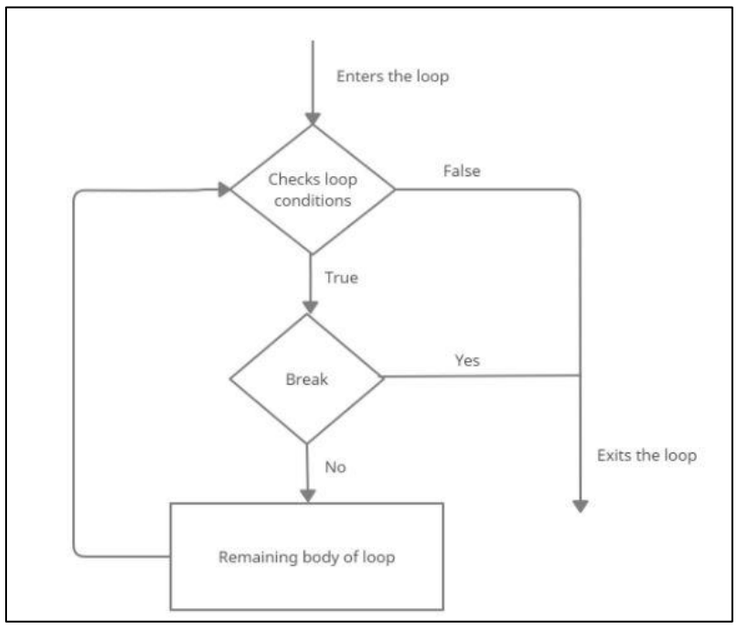
The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop only

Syntax:

```
Break
```

Working of break in for and while loops

```
for var in sequence:
        #body of for loop
        if condition:
                break
        #body of for loop
#Next statement after for loop
While condition:
        #body of while loop
        if condition:
                break
        #body of while loop
#Next statement after for loop
```



Eg:

```
# Use of break statement inside for loop
for val in "string":
   if val == "i":
      break
   print(val)
print ("The end")
```

```
Output:
s
t
r
The end
```

In this program, for iterates through the "string" sequence. When the letter is "i", then loop is terminated. That is why the for loop prints letters up till "i" only, after that, the loop terminates.

Eg:
```
# Use of break statement inside while loop
i = 1
while i < 6:
        print(i)
        if (i == 3):
                break
        i += 1
```

Output:
1
2
3

CONTINUE statement

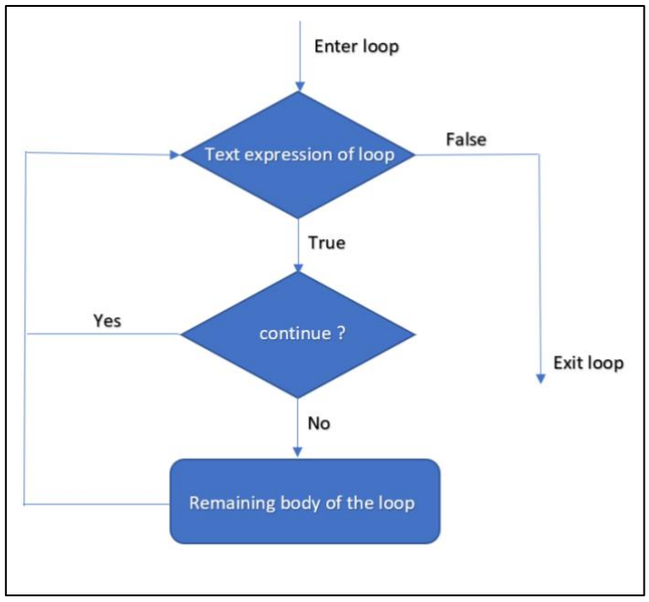
The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

Syntax

```
continue
```

Working of continue in for and while loops

```
for var in sequence:
        #body of for loop
        if condition:
                continue
        #body of for loop
#Next statement after for loop
While condition:
        #body of while loop
        if condition:
                continue
        #body of while loop
#Next statement after for loop
```



Eg:
```
# Program to show the use of continue statement inside loops
for val in "string":
   if val == "i":
      continue
   print(val)
print("The end")
```

Output
s
t
r
n
g

In this example the loop works all values for val except for value of ‘i'. Hence, output has all letters except "i" gets printed.

pass statement

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored. However, nothing happens when pass is executed. It results into no operation (NOP).

Syntax

```
pass
```

Generally, pass is used as a placeholder. When a loop or a function that is not implemented but needs to be implemented further in those situations pass statement is used. Since they cannot have an empty body. The pass statement to construct a body that does nothing.

Eg:
```
# pass is just a placeholder for functionality to be added later.
sequence = {'p', 'a', 's', 's'}
for val in sequence:
        pass
```
pass can also used to define an empty function or class
```
def function(args):
   pass
class example:
        pass
```

Python Functions

In Python, function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular units. As the program grows larger and larger, functions make it more organized and manageable. Furthermore, functions avoid repetition and makes code reusable.

Types of Functions

There are mainly two types of functions.

- Built-in functions - The built-in functions are those functions that are pre-defined in Python.
  Eg: print (), range (), len (), sorted (), max (), min () etc.,
- User-define functions - The user-defined functions are those define by the user to perform the specific task.

Defining the function:

In python function is defined by using def keyword.

Syntax:
```
def function_name(<<parameters>>):
        """docstring"""
        statement(s)
        <<return statement>>
```

- Here def is a key word used to define function
- function_name is a user defined name which follows the same rules of writing identifiers in Python. Parameters (arguments) through which we pass values to a function. They are optional.
- colon (:) is used to mark the end of function header
- Optional documentation string (docstring) to describe what the function does.
- One or more valid python statements that make up the function body.
- Statements must have same indentation level (usually 4 spaces).
- An optional return statement to return a value from the function.

Eg:
```
def greet(name):
        """This function greets good morning"""
        print("Hello, " + name + ". Good morning!")
```

Function Call

Once a function is defined, it can be called from another function or from a program or even the from Python prompt. To call a function simply type the function name with appropriate parameters.

Syntax: function_name(<<parameters>>)

Eg: greet(“Programmer”) # Output: Hello, Programmer. Good morning!

Docstring

The first string after the function header is called the docstring and is short for documentation string. It is used to explain in brief, what a function does. Although optional, documentation is a good programming practice. For writing doc strings single/ double triple quotes are used. Docstring can extend up to multiple lines. In order to access the docstring the __doc__ attribute of the function is used

Syntax: functionName.__doc__

Eg: print(greet.__doc__) # Output: This function greets good morning!

return statement:

In order to return value from a function the return statement is used. It can also be used to exit a function.

Syntax: return [expression]

Here expression which gets evaluated and the value is returned. If there is no expression in the statement then function is terminated.

```
Eg:  def abs_value(num):
         """This function returns the absolute
         value of the entered number"""
          if num >= 0:
              return num
         else:
              return (-1* num)
     print(abs_value(2))              # Output: 2
     print(absolute_value(-4))        # Output: 4
```

Arguments

The arguments are types of information which can be passed into the function. The arguments are specified in the parentheses. While calling a function any number of arguments can be passed, but they must be separate them with a comma. In Python there are several types of arguments which can be passed at the time of function call.

- Required arguments
- Default arguments
- Variable-length (Arbitrary) arguments
- Keyword arguments
- **Required arguments:**

  By default, a Python function must be called with the correct number of arguments as defined in the function. Meaning that if a function expects 2 arguments, then while calling that function with 2 arguments must be provided, not more, and not less otherwise error is thrown.

```
  Eg:  def  add(a,b):
           “”” adds two numbers and return the result”””
           return (a+b)
       print(add(20,30))              # Output: 50
       print(add(20,30,40))           # returns TypeError
```

```
print(add(10))                    #returns TypeError
```

» **Default Arguments:**

Python allows us to initialize the arguments at the function definition. If the value of any of the arguments is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

```
Eg:   def   square(num,exp=2):
          return (num**exp)
      print(num(5,3))                 # Output: 125
      print(num(10))                  # Output: 100
```

» **Variable-length (Arbitrary) Arguments:**

Python allows to define a function which takes variable-length arguments (i.e., not fixed number of arguments). Python provides us the flexibility to offer the comma-separated values which are internally treated as tuples at the function call. By using the variable-length arguments, any number of arguments can be passed to the function. However, at the function definition, define the variable-length argument using the *(Asterix)

```
Syntax:       def functionName( *names):
Eg:   def  display(*names):
          for value in names:
              print(value, end=" ")
      display("ram","shyam")          # Output: ram shyam
      display("sairam")               # Output: sairam
```

» **Keyword arguments:**

Python allows to call a function with the keyword arguments. This kind of function call will enable pass the arguments in the random order. The name of the arguments is treated as the keywords and matched in the function calling and definition. If the same match is found, the values of the arguments are copied in the function definition.

```
Eg:   def   greet(name,message):
          print("Hai ", name, message)
    greet(name = "Python", message=" Programming")    # Output: Hai Python Programming
    greet(message=" Programming",name="Python")       # Output: Hai Python Programming
```

Recursion

Recursion is a common mathematical and programming concept. In Python, a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions. Recursion is a process of a function calling itself continuously until a condition is met.

```
def   function_name(<<params>>):
      #statements
      ========
      if condition :
          return          # termination of recursion
      else:
          function_name(<<params>>)
```

Eg: # Program to find the factorial of a given number using recursion

```
def fact(num):
    """Recursive function to find the factorial of an integer"""
    if num == 1:
        return 1
    else:
        return (num * fact(num-1))
f = fact(4)                    # function call
print("The factorial of", num, "is", f)
```

In this example, fact() is a recursive functions as it calls itself. When a call is made to this function with a positive integer, it will recursively call itself by decreasing the number. Each function call multiples the number with the factorial of number 1 until the number is equal to one.

These recursive calls involved in this are as follows.

```
fact(4)                    # 1st call with 4
4 * fact(3)                # 2nd call with 3
4 * 3 * fact(2)            # 3rd call with 2
4 * 3 * 2 * fact(1)        # 4th call with 1
4 * 3 * 2 * 1              # return from 4th call as number=1
4 * 3 * 2                  # return from 3rd call
4 * 6                      # return from 2nd call
24                         # return from 1st call
```

Recursion ends when the number reduces to 1. This is called the base condition. Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

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.

Disadvantages

- Sometimes 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.

Anonymous (Lambda) Functions

Anonymous functions are not declared in the standard manner by using the def keyword. Python provides lambda keyword to create anonymous functions. Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.

An anonymous function cannot be a direct call to print because lambda requires an expression

Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

Syntax: lambda [arg1 [,arg2,.....argn]] : expression

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Eg:
```
# Program to show the use of lambda functions
double = lambda x: x * 2
print(double(5))                    # Output: 10
```

In this program, lambda x: x * 2 is the lambda function. Here x is the argument and x * 2 is the expression that gets evaluated and returned. This function has no name (so the name anonymous). It returns a function object which is assigned to the identifier double. Using the object, it can be called like a normal function.

```
double = lambda x: x * 2
```

nearly the same as

```
def double(x):
    return x * 2
```

Eg:
```
sum = lambda arg1, arg2: arg1 + arg2;
# calling lambda function using sum
print ("Value of total: ", sum (10, 20))        # Output: 30
print ("Value of total: ", sum (20, 20))        # Output: 40
```

Advantages:

- Lambda functions are used when a nameless function is required for a short period of time.
- In Python it is used as an argument to a higher-order function (a function that takes in other functions as arguments).
- Lambda functions are used along with built-in functions like filter(), map() etc.
- filter() function filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

  Eg:
  ```
  # Program to filter out only the even items from a list
  values= [1, 5, 4, 6, 8, 11, 3, 12]
  even= list(filter(lambda x: (x%2 == 0) ,num))
  print(even)              # Output: [4, 6, 8, 12]
  ```

- map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter. The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

  Eg:
  ```
  # Program to double each item in a list using map()
  num= [1, 5, 4, 6, 8, 11, 3, 12]
  doubles= list(map(lambda x: x * 2 ,num))
  print(new_list)              # Output: [2, 10, 8, 12, 16, 22, 6, 24]
  ```

Scope and Lifetime of variables

The scope defines the availability and accessibility of the python object. To access the particular variable in the code, the scope must be defined as it cannot be accessed from anywhere in the program. The particular coding

region where variables are visible is known as scope. The lifetime of a variable is the period throughout which the variable exits in the memory of your Python program.

» **Local Scope:**

The Variables which are defined in the function belongs to the local scope of that function, and can only be used inside that function. These variables are defined in the function body and are not visible from outside. Hence, they have a local scope. The lifetime of these variables inside is as long as the function executes. They are destroyed when function is terminated. Hence, functions do not remember the value of a variable from its previous calls.

» **Global Scope:**

A variable created in the main body of the Python code is a global variable and belongs to the global scope. The global scope means the variable is accessible from anywhere in the program. The global variables are declared outside the functions as their scope is not limited to any function. The lifetime of the global variable is equal to the runtime of the program.

Eg:

```
x = 300                    #global
def test ():
        y=100              #local
        print ("Inside test function")
        print("X:",x)
        print("Y:",y)
# end of function
test ()
print ("Outside of test function")
print("X:",x)
print("Y:",y)              # error undefined y
```

```
Output
Inside test function
X: 300
y: 100
Inside test function
X: 300
NameError: name 'y' is not defined
```

Global Keyword:

Python provides global keyword in order to make a variable which is in local scope as global variable. i.e., the global keyword makes the variable global. When global keyword is used, the variable belongs to the global scope

Eg:

```
def test():
        global x
        x = 300
test()
print(x)                   # Prints 300
```

ITERATORS

An iterator in Python is an object that contains a countable number of elements that can be iterated upon. In simpler words, we can say that Iterators are objects that allow you to traverse through all the elements of a collection and return one element at a time.

Iterator consists the methods __iter__() and __next__().

Iterator vs Iterable

Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable containers which you can get an iterator from.

All these objects have a iter() method which is used to get an iterator:

Example:

Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)


print(next(myit))
print(next(myit))
print(next(myit))
```

Looping Through an Iterator

We can also use a for loop to iterate through an iterable object:

Example

Iterate the values of a tuple:

```
mytuple = ("apple", "banana", "cherry")
for x in mytuple:
  print(x)
```

Create an Iterator

To create an object/class as an iterator you have to implement the methods __iter__() and __next__() to your object.

As you have learned in the Python Classes/Objects chapter, all classes have a function called __init__(), which allows you to do some initializing when the object is being created.

The __iter__() method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

The __next__() method also allows you to do operations, and must return the next item in the sequence.

Example

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.):

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self
  def __next__(self):
    x = self.a
    self.a += 1
    return x
myclass = MyNumbers()
myiter = iter(myclass)
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

```
print(next(myiter))
print(next(myiter))
```

StopIteration

The example above would continue forever if you had enough next() statements, or if it was used in a for loop.

To prevent the iteration to go on forever, we can use the StopIteration statement.

In the __next__() method, we can add a terminating condition to raise an error if the iteration is done a specified number of times:

Example

Stop after 20 iterations:

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self
  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration
myclass = MyNumbers()
myiter = iter(myclass)
for x in myiter:
  print(x)
```

Generators

Python provides a generator to create your own iterator function. A generator is a special type of function which does not return a single value, instead, it returns an iterator object with a sequence of values. In a generator function, a yield statement is used rather than a return statement.

All the overhead in building iterators are automatically handled by generators in Python.

Example: Generator Function

```
def mygenerator():
    print('First item')
    yield 10
    print('Second item')
    yield 20
    print('Last item')
    yield 30
```

The iterator for the above function can be created as follows:

```
>>> gen = mygenerator()
>>> next(gen)
```

```
First item
10
>>> next(gen)
Second item
20
>>> next(gen)
Last item
30
```

Differences between Generator function and a Normal function

- Generator function contains one or more yield statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, StopIteration is raised automatically on further calls.

Using for Loop with Generator Function

The generator function can also use the for loop.

Example: Use For Loop with Generator Function

```
def get_sequence_upto(x):
    for i in range(x):
        yield i
```

The above generator function get_sequence_upto() can be called as below.

Example: Calling Generator Function

```
>>> seq = get_sequence_upto(5)
>>> next(seq)
0
>>> next(seq)
1
>>> next(seq)
2
>>> next(seq)
3
>>> next(seq)
4
```

Example: Write a program to print the table of the given number using the generator.

```
def table(n):
    for i in range(1,11):
        yield n*i
          i = i+1
for i in table(15):
    print(i)
```

Generator Expression

Python also provides a generator expression, which is a shorter way of defining simple generator functions. The generator expression is an anonymous generator function. The following is a generator expression for the square_of_sequence() function.

Example: Generator Expression

```
>>> squres = (x*x for x in range(5))
>>> print(next(squre))
0
>>> print(next(squre))
1
>>> print(next(squre))
4
>>> print(next(squre))
9
>>> print(next(squre))
16
```

In the above example, (x*x for x in range(5)) is a generator expression. The first part of an expression is the yield value and the second part is the for loop with the collection.

Note: The generator expression can also be passed in a function. It should be passed without parentheses, as shown below.

Example: Passing Generator Function

```
>>> import math
>>> sum(x*x for x in range(5))
30
```

Example: Python program to generate Fibonacci Numbers using generator

```
def fib(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a + b
# Create a generator object
x = fib(5)
# Iterating over the generator object using next
print(next(x))
print(next(x))
print(next(x))
print(next(x))
print(next(x))
 # Iterating over the generator object using for in loop.
print("\nUsing for in loop")
for i in fib(5):
    print(i)
```