PYTHON Unit-1

Python was developed by Guido van Rossum during 1985-1990.

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

It is a general-purpose interpreted, interactive, object oriented, and high-level programming language.

Python source code is available under the GNU General Public License (GPL) and it is now maintained by a core development team at the National Research Institute.

1.1 FEATURES OF PYTHON

a) Simple and easy-to-learn:- Python is a simple language with few keywords, simple structure and its syntax is also clearly defined. This makes Python a beginner's language

b) Interpreted and Interactive :- Python is processed at runtime by the interpreter. We need not compile the program before executing it. The Python prompt interact with the interpreter to interpret the programs that we have written Python has an option namely interactive mode which allows interactive testing and debugging of code.

c) Object-Oriented:- Python supports Object Oriented Programming (OOP) concepts that encapsulate code within objects All concepts in OOP's like data hiding, operator overloading, inheritance etc. can be well written in Python. It supports functional as well as structured programming

d) Portable :- Python can run on a wide variety of hardware and software platforms and has the same interface on all platforms. All variants of Windows, Unix, Linux and Macintosh are to name a few.

e) Scalable:- Python provides a better structure and support for large programs thanshell scripting. It can be used as a scripting language or can be compiled to bytecode (intermediate code that is platform independent) for building large applications.

f) Extendable:- we can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient. It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java

g) Dynamic :- Python provides very high-level dynamic data types and supports dynamic type checking. It also supports automatic garbage collection.

h) GUI Programming and Databases :- Python supports GUI applications that can be created and ported to many libraries and windows systems, such as Windows Microsoft Foundation Classes (MFC), Macintosh, and the X Window system of Unix. Python also provides interfaces to all major commercial databases.

i) Broad Standard Library :- Python's library is portable and cross platform compatible on UNIX, Linux, Windows and Macintosh. This helps in the support and development of a wide range of applications from simple text processing to browsers and complex games.

IDENTIFIERS

A Python identifier is a name used to identify a variable, function, class, module or any other object. Python is case sensitive and hence uppercase and lowercase letters are considered distinct. The following are the rules for naming an identifier in Python.

a) 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 (_). For example Total and total is different.

b) Reserved keywords cannot be used as an identifier.

c) Identifiers cannot begin with a digit. For example 2more, 3times etc. are invalid identifiers.

d) Special symbols like @, !, #, $, % etc. cannot be used in an identifier For example sum@, #total are invalid identifiers.

e) Identifier can be of any length.

Some examples of valid identifiers are totai, max_mark, count2, Student etc. Here are some naming conventions for Python identifiers.

• Class names start with an uppercase letter All other identifiers start with a lowercase ietter. Eg Person

Starting an identifier with a single leading underscore indicates that the identifier is private. Eg: _sum

. Starting an identifier with two leading underscores indicates a strongly private identifier Eg: __sum

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

1.4 RESERVED KEYWORDS

These are keywords reserved by the programming language and prevent the user or the programmer from using it as an identifier in a program. There are 33 keywords in Python 3.3. This number may vary with different versions. To retrieve the keywords in Python the following code can be given at the prompt. All keywords except True, False and None are in lowercase.

>>> import keyword

>>> print (keyword.kwlist)

["False", "None', 'True','continue', 'def, 'del, 'global',"from",not', 'or','yield']

1.5 VARIABLES

Variables are reserved memory locations to store values. Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables Python variables do not need explicit declaration to reserve memory space. happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables. The operand to the left of the operator is the name of the variable and the operand to the right of the operator is the value stored in the variable.

Example Program

a=100

b=1000.0

name= "John"

print (a)

print (b)

print (name)

Here, 100, 1000.0 and "John" are the values assigned to a, b, and name variables. respectively. This produces the following result.

100

1000.0

John

Python allows you to asign a single value to several variables simultaneously. For example

a=b=c=1

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example

a, b, c= 1, 2, "Tom"

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "Tom" is assigned to the variable c

1.6 COMMENTS IN PYTHON

Comments are very important while writing a program It describes what the source code has done. Comments are for programmers for better understanding of a program. In Python, we use the hash (#) symbol to start writing a comment. A hash sign (0) that is not inside a string literal begins a comment All characters after the # and up to the end of the physical line are part of the comment. It extends up to the newline character Python interpreter ignores comment.

Example Program

>>>#This is dena of comment

>>>#Display Hello

>>>print("Hello")

This produces the following result: Hello

For multiline comments one way is to use (#) symbol at the beginning of each line. The following example shows a multiline comment. Another way of doing this is to use triple quotes, either ''' or """.

Example 1

```
>>>#This is a very long sentence.
>>>#and it ends after
>>>#Three lines
```

Example 2

```
>>>"""This is a very long sentence
>>>and it ends after
>>>Three lines """
```

Both Example 1 and Example 2 given above produces the same result

1.7 INDENTATION IN PYTHON

Most of the programming languages like C, C++ and Java use braces { } to define a block of code. Python uses indentation. A code block (body of a function, loop etc) starts with indentation and ends with the first unindented line. The amount of indentation can be decided by the programmer, but it must be consistent throughout the block Generally four whitespaces are used for indentation and is preferred over tabspace. For example

```
if True:
    print ("Correct")
else:
    print ("Wrong")
```

1.8 MULTI-LINE STATEMENTS

Instructions that a Python interpreter can execute are called statements. For example, a=1 is an assignment statement. In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character(\).

For example:

```
grand_ total= first_item + \
                    second_item+\
                     third item
```

Statements contained within the [ ], { }. or () brackets do not need to use the line continuation character.

For example

```
months=[ 'May', 'June', 'July', 'August', 'September','October', 'November', 'December']
```

We could also put multiple statements in a single line using semicolons,

```
a = 1; b = 2; c= 3
```

1.9 MULTIPLE STATEMENT GROUP (SUITE)

A group of individual statements, which make a single code block is called a suite in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite Header lines begin the statement (with the keyword) and terminate with a colon () and are followed by one or more lines which make up the suite. For example

```
if expression :
    Suite
elif expression :
     suite
else :
     suite
```

1.10 QUOTES IN PYTHON

Python accepts single ('), double (") and triple (' ' 'or" " ") quotes to denote string literals. The same type of quote should be used to start and end the string. The triple quotes are used to span the string across multiple lines. For example, all the following are legal.

word=' single word'

sentence = "This is a short sentence."

paragraph="""" This is a long paragraph. It consists of several lines and sentences """"

1.11 INPUT, OUTPUT AND IMPORT FUNCTIONS

1.11.1 Displaying the Output

The function used to print output on a screen is the print statement where it can pass zero or more expressions separated by commas. The print function converts the expressions we pass into a string and writes the result to standard output.

Example 1:

>>>print("Learning Python is fun and enjoy it.") output:

Learning Python is fun and enjoy it.

Example 2:

>>>a=2

>>>print ("The value of a is",a)

output :

The value of a is 2

By default a It a space is added after the text and before the value of variable a.

syntax of print function :

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

Here, objects are the values to be printed .Sep is the separator used between the values. Space character is the default separator. end is printed after printing all the values. The default value for end is the new line. file is the object where the values are printed and its default value is sys atdout (screen) .Flush determines whether the output stream needs to be flushed for any waiting output. A True value forcibly flushes the stream .

Example

>>>print (1,2,3,4)

>>>print (1,2,3,4, sep='+')

>>> print (1,2,3,4, sep='+',end='%')

The output will be as follows

1 2 3 4

1+2+3+4

1+2+3+4%

1.11.2 Reading the Input

raw_input function

The raw_input([prompt]) function reads one line from standard input and returns it as a string (removing the trailing newline). This prompts you to enter any string and it would display same string on the screen.The raw_input function is not supported by Python 3.

Example

str= raw_input ("Enter your name:");

print ("Your name is : ", str)

Output:

Enter your name: Collin Mark

Your name is : Collin Mark

input function

The input([prompt]) function is equivalent to raw_input, except that it assumes the input is a valid Python expression and returns the evaluated result .

Example 1

n = input ("Enter a number: ");

print ("The number is: , n)

Output:

Enter a number: 5

The number is 5

Example 2

n= input ("Enter an expression:");

print ("The result is: ", n)

Output:

Enter an expression: 5*2

The result is: 10.

1.11.3 Import function

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. We use the import keyword to do this. For example, we can import the math module by typing in import math

>>> import math

>>> math.pi

3.141592653589793

1.12 OPERATORS

Operators are the constructs which can manipulate the value of operands. Consider the expression a = b + c. Here a, b and care called the operands and +, are called the operators. There are different types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

1.12.1 **Arithmetic Operators**

Arithmetic operators are used for performing basic arithmetic operations.

Table 1.2 Arithmetic Operators in Python

Operator	Operation	Description
+	Addition	Adds values on either side of the operator.
-	Subtraction	Subtracts right hand operand from left hand operand.
*	Multiplication	Multiplies values on either side of the operator.
/	Division	Divides left hand operand by right hand operand.
%	Modulus	Divides left hand operand by right hand operand and returns remainder .
**	Exponent	Performs exponential (power) calculation on operators.
//	Floor Division	The division of operands where the result is the quotient in which the digits after the decimal point are removed.

Example Program

a, b, c = 10, 5, 2

```
print ("Sum=, (a+b))
print ("Difference=", (a-b))
print ("Product=, (a*b))
print("Quotient=", (a/b))
print("Remainder=", (b%c))
print ("Exponent=", (b**2))
 print ("Floor Division=", (b//c))
```

Output

Sum =15

Difference =5

Product = 50

Quotient= 2

Remainder= 1

Exponent =25

Floor Division =2

1.12.2 **Comparison Operators**

These operators compare the values on either sides of them and decide the relation among them. They are also called relational operators.

Table 1.3 Comparison (Relational Operators) in Python

Operator	Description
==	If the values of two operands are equal, then the condition becomes true.
!=	If values of two operands are not equal, then condition becomes true.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.

Example Program

```
a, b=10,5
print ("a==b is", (a==b))
print ("a!=b is", (a!=b))
print ("a>b is", (a>b))
print ("a<b is", (a<b))
print ("a>=b is", (a>=b))
print ("a<=b is", (a<=b))
```

Output

a==b is False

a!=b in True

a>b is True

a<b is False

a>=b is True

a<=b is False

1.12.3 **Assignment Operators**

Python provides various assignment operators. Various shorthand operators for addition, subtraction, multiplication, division, modulus, exponent and floor division are also supported by Python.

Table 1.4 Assignment Operators

Operator	Description
=	Assigns values from right side operands to left side operand.
+=	It adds right operand to the left operand and assign the result to left operand.
-=	It subtracts right operand from the left operand and assign the result to left operand.
*=	It multiplies right operand with the left operand and assign the result to left operand.
/=	It divides left operand with the right operand and assigns the result to left operand.
%=	It takes modulus using two operands and assign the result to left operand.
**=	Performs exponential (power) calculation on operators and assign value to the left operand.
//=	It performs floor division on operators and assign value to the left operand.

The assignment operator is used to assign values or values of expressions to a varia Example: a = b+ c.

For example c+=a is equivalent to c=c+a. Similarly c-=a is equivalent c=c-a,

c*=a is equivalent to c=c*a,

c/=a is equivalent to c=c/a,

C%=a is equivalent to c=c%a

Example Program

```
a,b=10,5
a+=b
print (a)
a,b=10,5
a-=b
print (a)
a,b=10, 5
a*=b
print (a)
a,b=10,5
a/=b
print (a)
b, c=5,2
b%=c
print (b)
b, c=5,2
b**=c
print (b)
b, c=5,2
b//=c
print (b)
```

Output

```
15
5
50
2
1
25
2
```

1.12.4 Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation.

Table 1.5 Bitwise Operators

Operator	Operation	Description
&	Binary AND	Operator copies a bit to the result if it exists in both operands.
|	Binary OR	It copies a bit if it exists in either operand.
^	Binary XOR	It copies the bit if it is set in one operand but not both.
~	Binary Ones Complement	It is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift	The left operand's value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift	The left operand's value is moved right by the number of bits specified by the right operand.

Let a =60 (0011 1100 in binary) and b= 2 (0000 0010 in binary)

Example Program

```
a,b=60,2
print (a&b)
print (alb).
print (a*b)
print (-a)
```

print (a>>b)

print (a<<b)

Output

0

62

62

-61

15

240

1.12.5 Logical Operators

Logical operators are used to combine conditional statements:

Table 1.6 Logical Operators

Operator	Operation	Description
and	Logical AND	If both the operands are true then condition becomes true.
or	Logical OR	If any of the operands are true then condition becomes true.
not	Logical NOT	Used to reverse the logical state of its operand.

Example Program:

a, b, c, d=10,5,2,1

print ((a>b) and (c>d)).

print ((a>b) or (d>c))

print (not (a>b))

Output:

True

True

False

1.12.6 Identity Operators

Identity operators compare the memory locations of two objects.

Table 1.8 Identity Operators

Operator	Description
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

Example Program :

a, b, c=10, 10,5

print (a is b)

print (a is c)

print (a is not b)

Output:

True

False

False

1.12.7 Membership Operators

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

Example Program :

S='abcde'

print ('a' in s)

print ('f' in s)

print('f'not in s)

Output:

True

False

True

Operator Precedence

The following Table lists all operators from highest precedence to lowest.

Operator associativity determines the order of evaluation, when they are of the same precedence, and are not grouped by paranthesis.

An operator may be left-associative or right-associative.

In left-associative, the operator falling on the left side will be evaluated first, while in right associative, operator falling on the right will be evaluated first.

In Python, '='and '**' are right-associative while all other operators are left-associative.

Table 1.9 Operator Precedence

Operator	Description
**	Exponentiation (raise to the power)
~, +, -	Complement, unary plus and minus
*, /, %, //	Multiply, divide, modulo and floor division
+, -	Addition and subtraction
>>, <<	Right and left bitwise shift
&	Bitwise 'AND'
^, |	Bitwise exclusive 'OR' and regular 'OR'
<=, < >, >=	Comparison operators
<>, ==, !=	Equality operators
=, %=, /=, //=, -=, +=, *=, **=	Assignment operators
is, is not	Identity operators
in, not in	Membership operators
not, or, and	Logical operators

DECISION MAKING

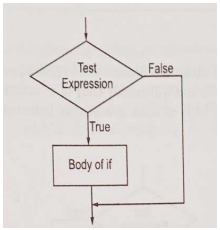
Decision making is required when we want to execute a code only if a certain condition is satisfied. The if-elif-else statement is used in Python for decision making.

if statement

Syntax:

if test expression:

    statement(s)



Here, 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 ends with the first unindented line.

Python interprets non-zero values as True. None and 0 are interpreted as False.

Example Program:

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

if num == 0:

    print("Zero")

print ("This is always printed")
```

Output:

Enter a number: 0

Zero This is always printed

if...else statement

Syntax:

if test expression:

    Body of if

else:

    Body of else

The if..else statemen 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.

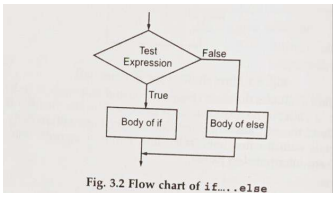


Fig. 3.2 Flow chart of if…..else

Example Program:

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

Output :

Enter a number: 5

Positive Number or Zero

if...elif...else statement

Syntax:

```
if test expression:
        Body of if
elif test expression:
      LBody 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 it 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. An if block can have only one else block. But it can have multiple elif blocks.

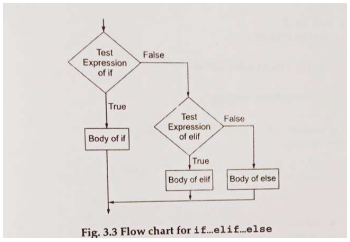


Fig. 3.3 Flow chart for if…elif…else

Example Program:

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

Output :

Enter a number: 5

Positive Number

Nested if statement

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming. Indentation is the only way to identify the level of nesting.

Example Program:

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

Output :

Enter a number: 5

Positive Number

LOOPS

Python provides various control structures that allow for repeated execution. A loop statement allows us to execute a statement or group of statements multiple times.

3.2.1 for loop

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

Syntax:-

```
for item in sequence:
    Body of for
```

Here, item is the variable that takes the value of the item inside the sequence of each iteration. The sequence can be list, tuple, string, set etc. Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

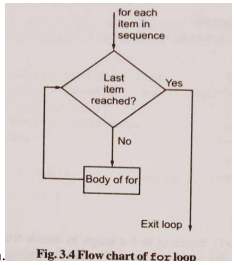


Fig. 3.4 Flow chart of for loop

Example Program 1

```
#Program to find the sum of all numbers stored in a list
numbers= [2,4,6,8,10]
 #variable to store the sum
sum = 0
#iterate over the list
for item in numbers:
    sum=sum+ item
# print the sum
print ("The sum is", sum)
```

Output :

The sum is 30

range() Function

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

Syntax: range(start, stop, step)

start :-Optional. An integer number specifying at which position to start. Default is 0

stop :-Required. An integer number specifying at which position to stop (not included).

step :-Optional. An integer number specifying the incrementation. Default is 1

Example:-

```
for num in range (2, 10, 2):
    print ("Number =", num)
```

Output

Number =2

Number= 4

Number= 6

Number= 8

enumerate() function

The enumerate() function takes a collection (e.g. a tuple) and returns it as an enumerate object.

The enumerate() function adds a counter as the key of the enumerate object.

Syntax :enumerate(iterable, start)

iterable: An iterable object

start : A Number. Defining the start number of the enumerate object. Default 0.

Example:-

```
flowers=["rose", "lotus", jasmine", "sun flower"] print (list (emunerate (flowers)))
for index, item in enumerate (flowers):
      print (index, item)
```

Output:

[(0.' rose'), (1,' lotus'), (2. 'jasmine'), (3, 'sun flower')]

0 rose

1 lotus

2 jasmine

3 sun flower

for loop with else statement

Python supports to have an else statement associated with a loop statement. If the elsestatement is used with a for loop, the else statement is executed when the loop has finish iterating the list. A break statement can be used to stop a for loop. In this case, the else part is ignored. Hence, a for loop's else part runs if no break occurs.

Example Program

```
#Program to find whether an item is present in the list
list=[10, 12, 13, 34,27,98]
 num= int (input("Enter the number to be searched in the list: "))
for item in range (len (list)):
       if list[item]== num:
          print ("Item found at: , (item+1))
          break
       else:
        print ("Item not found in the list)
```

Output 1

Enter the number to be searched in the list 34 Item found at 4

Output 2

Enter the number to be searched in the list:99 Item not found in the list

while loop

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is True. We generally use this loop when we don't know the number of times to iterate in advance.

Syntax:

while test expression:

    Body of while

In while loop, test expression is checked first. The body of the loop is entered only if the test expression evaluates to True. After one iteration, the test expression is checked again. This process is continued until the test expression evaluates to False. In Python, the body of the while loop is determined through indentation. Body starts with indentation and the first unindented line shows the end. When the condition is tested and the result is False, the loop body will be skipped and the first statement after the while loop will be executed.

Example Program

```
# Program to find the sum of first N natural numbers
 n=int (input ("Enter the limit: "))
sum=0
i=1
while i<=n:
    sum=sum+i
    i=i+1
print ("Sum of first", n. "natural numbers is", sum)
```

Output:

Enter the limit: 5

Sum of first 5 natural numbers is 15

while loop with else statement

If the else statement is used with a while loop, the else statement is executed when the condition becomes False while loop can be terminated with a break statement. In such case, the else part is ignored. Hence, a while loop's else part runs if no break occurs and the condition is False

Example Program:

```
count=1
while count <=3:
    print ("Python Programming)
    count= count+1
else:
    print ("Exit")
print ("End of Program")
```

Output :

Python Programming

Python Programming Python Programming

Exit

End of Program

NESTED LOOPS

Sometimes we need to place a loop inside another loop. This is called nested loop. We can have nested loops for both while and for Syntax for nested for loop:-

```
for iterating_variable in sequence:
    for iterating_variable in sequence:
        statements (s)
    statements (s)
```

Syntax for nested while loop :-

```
while expression:
    while expression:
        statement (s)
    statement (s)
```

TYPES OF LOOPS

There are different types of loops depending on the position at which condition is checked

Infinite Loop:- A loop becomes infinite loop if a condition never becomes False. This results in a loop that never ends. Such a loop is called an infinite loop. We can create an infinite loop using while statement. If the condition of while loop is always True, we get an infinite loop.

Example Frogram:-

```
count=1
while count==1:
    n=input ("Enter a Number :")
print ("Numbers", n)
```

This will continue running unless you give CTRL+C to exit from the loop.

Loops with condition at the top:-

This is a normal while loop without break statements. The condition of the while loop is at the top and the loop terminates when this condition is False.

Loop with condition in the middle:-

This kind of loop can be implemented using an infinite loop along with a conditional break in between the body of the loop.

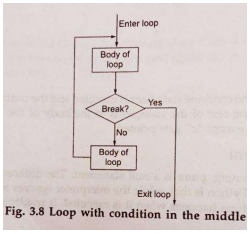
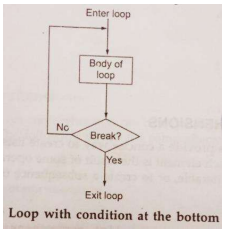


Fig. 3.8 Loop with condition in the middle

Loop with condition at the bottom:-

This kind of loop ensures that the body of the loop is executed least once. It can be implemented using an infinite loop along with a conditional break at the end. This is similar to the do..while loop in C.



Loop with condition at the bottom

CONTROL STATEMENTS

Control statements change the execution from normal sequence. The break and continue statements are used to terminate the current iteration or even the whole loop without checking test expression. Python supports the following three control statements.

1. break

2. continue

3. pass

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 it is inside a nested loop (loop inside another loop), break will terminate the innermost loop. It can be used with both for and while loops.

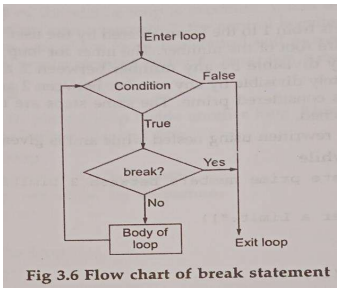


Fig 3.6 Flow chart of break statement

Example :-

```
for i in range (2,10,2):
    if i==6:
        break
       print (i)
print("End of Program")
```

Output :-

```
2
4
End of Program
```

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. Continue returns the control to the beginning of the loop. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop. The continue statement can be used in both while and for loops.

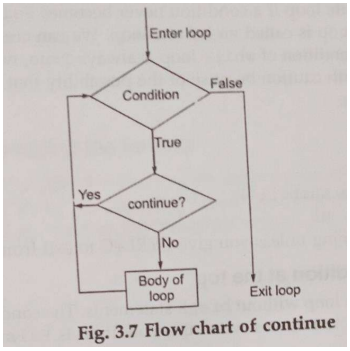


Fig. 3.7 Flow chart of continue

Example :-

```
for letter in 'abcd' :
    if letter =='c' :
        continue
     print (letter)
```

Output:-

a

b

d

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. But nothing happens when it is executed. It results in no operation

It is used as a placeholder. We use the pass statement to construct a body that does nothing.

Example:-

for val in sequence:

    pass

STANDARD DATA TYPES

Python has six basic data types which are as follows:

1. Numeric

2. String

3. List

4. Tuple

5. Dictionary

6. Boolean

Numeric

Numeric data can be broadly divided into integers and real numbers (ie, fractional numbers) Integers can themselves be positive or negative. Unlike many other programming languages, Python does not have any upper bound on the size of integers The real numbers or fractional numbers are called floating point numbers on programming languages. Such floating point numbers contain a decimal and a fractional part.

String

Single quotes or double quotes are used to represent strings. A string in Python can be a series or a sequence of alphabets, numerals and special characters. The first character of a string has an index 0

There are many operations that can be performed on a string There are several operators such as slice operator ([]) and [:]), concatenation operator (+), repetition operator (*), etc. Slicing is used to take out a subset of the string, concatenation is used to combine two or more than two strings and repetition is used to repeat the same string several times.

Example :

>>> sample_string= "Hello"

>>> print sample_string

Output:

'Hello'

>>>print sample_string + "World"

Output:

'HelloWorld

>>> print sample_string * 3

Output:

'Hello Hello Hello'

Python also provides slice operators ([] and [:]) to extract substring from the string. In Python, the indexing of the characters starts from 0, therefore, the index value of the first character is 0.

Syntax:-

sample_string[start :end <: step>] step is optional

>>>sample_string="Hello"

>>>sample_string[1]   # display 'e'

>>>sample_string[0:2] # display 'He'

>>>sample_string ="HelloWorld"

>>>sample_string[1:8:2]

display all the alternate characters between index 1 to 8 ie, 1,3,5,7 .Output will be 'elWr'

List:-

A list can contain the same type of items. Alternatively, a list can also contain different types of items. A list is an ordered and indexable sequence. To declare a list in Python, we need to separate the items using commas and enclose them within square brackets. I.e a list can contain different types of items.

Similar to the string data type, the list also has plus (+), asterisk (*) and slicing [:] operators forconcatenation, repetition and sub-list, respectively.

Example :

>>>first=[1, "two", 3.0,"four"]

>>>second= ["five", 6]

>>>first

Output:

[1, "two", 3.0,"four"]

>>>first + second

Output:

[1, "two", 3.0,"four"," five",6]

>>>second * 3

["five", 6,"five", 6,"five", 6]

>>>first[0:2]

Output:

[1, "two"]

Tuple:-

Similar to a list, a tuple is also used to store sequence of items. Like a list, a tuple consists of items separated by commas. However, tuples are enclosed within parentheses rather than within square brackets.

Example :

>>>third=(7, "eight",9, 10.0)

>>>third

Output:

(7. 'eight', 9, 10.0)

Lists and tuples have the following differences:

• In lists, items are enclosed within square brackets [], whereas in tuples, items are enclosed with parentheses ()

• Lists are mutable whereas Tuples are immutable. Tuples are read only lists. Once the items the tuple cannot be modified.

Dictionary:-

It is the same as the hash table type .The order of elements in a dictionary is undefined But, we can iterate over the following:

1 The keys

2 The values

3. The items (key-value pairs) in a dictionary

A Python dictionary is an unordered collection of key-value pairs. When we have the large amount of data: the dictionary data type is used. Keys and values can be of any type in a dictionary. Items in dictionary are enclosed in the curly-braces {} and separated by the comma (,). A colon (:) is used to separate key from value. A key inside the square bracket [] is used for accessing the dictionary items.

Example :

```
>>> dict1={ 1:"first line", "second":2}
>>> dict1[3]="third line"

>>> dict1
```

Output:

```
{1: first line", "second": 2, 3: third line'}
>> dict1.keys()
```

Output:

```
[1, 'second'. 3]
>>> dict1.values()
```

Output:

```
['First line', 2, 'third line']
```

Boolean:

sometimes we need to store the data in the form of 'Yes' or 'No'. In terms of programming language, Yes is similar to True and No is similar to False.

This True and False data is known as Boolean Data and the data types which stores this Boolean data are known as Boolean Data Types

Example :

```
>>> a = True
>>> type (a)

<type 'bool'>
>>> x = False
>>> type (x)

<type 'bool'>
```

type () Function :- is a built-in function which returns the datatype of any arbitrary object. The object is passed as an argument to the type() function. Type() function can take anything as an argument and returns its datatype, such as integers, strings, dictionaries, lists, classes, modules, tuples, functions, etc.

Sets:

An unordered collection of data known as Set. A Set does not contain any duplicate values or elements. Union, Intersection, Difference and Symmetric Difference are some operations which are performed on sets.

Union: Union operation performed on two sets returns all the elements from both the sets. It is performed by using & operator.

Intersection: Intersection operation performed on two sets returns all the element which are common or in both the sets. It is performed by | using operator

RTrformed on two sets set1 and set2 returns the elements which on set1 but not in set2. It is performed by using - operator

Symmetric Difference: Symmetric Difference operation performed on two sets returns the element which are present in either set1 or set2 but not in both. It is performed by using ^ operator.

Example

set1=set([1,2,8,5,4])

set2=set([1, 9,3,2,5])

>>> print set1

Output:

set([8,1,2,4,5])

>>> print set2

Output:

set([1,2,3,5,9])

>>>intersection=set1|set2

>>>print intersection

Output:

set([1,2,5])

>>>union=set1&set2

>>>print union

Output:

set([1,2,3,4,5,8,9])

>>>difference=set1- set2

>>>print difference

Output:

set([8,4])

>>>symm_diff=set1 ^ set2

>>>print symm_diff

Output:

set([3,4,8,9])

STATEMENT AND EXPRESSION

Statement:-

A statement can be an instruction that can be interpreted by the Python interpreter. A statement is interpreted by the interpreter and after execution displays some result (if there is a need for displaying it) For eg., print statement produces some result to display but it does not happen in case of assignment statement A program can contain many statements in sequence. If there are multiple statements, the result is displayed after every statement

.

Let us look at an example of the assignment statement

>>> massage="Hello world"

Due to the assignment operator (=), the message variable stores the Hello world string.

>>> print 1

>>>x=2

>>>print x

Expression:-

An expression is a combination of variables, operators, values and a reserve keyword. Whenever we type an expression in the command line, the interpreter evaluates it and produces the result.

LIST

List is an ordered sequence of items. It is one of the most used data type in Python and is very flexible. All the items in a list do not need to be of the same type. Items separated by commas are enclosed within brackets [] .To some extent, lists are similar to arrays in C. Orie difference between them is that all the items belonging to a list can be of different data type. The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indices starting at 0 in the beginning of the list and ending with -1. The plus (+) sign is the list concatenation operator and the asterisk (*) is the repetition operator

Example :-

```
first_list=['abcd", 147, 2.43,"Tom', 74.9]
small_list= [111 ,"Tom"]
print (first_list)
print (first_list [0])
print (first_list [1:3])
 print (first_list [2:])
print (small_list * 2)
print (first_list + small_list)
```

Output:-

```
['abcd', 147, 2.43. Tom', 74.9]
abcd
[147, 2.43]
[2.43, Tom', 74.9]
[111, "Tom", 111, "Tom"]
[['abcd', 147, 2.43, 'Tom', 74.9, 111, 'Tom']
```

We can update lists by using the slice on the left hand side of the assignment operator

Updates can be done on single or multiple elements in a list.

Example :-

```
list=['abcd", 147, 2.43, 'Tom', 74.9]
 print("Item at position 2=, list[2])
list[2]=500
print("Item at position 2=, list [2])
print("Item at Position 0 and 1 is, list[0], list [1])
list [0]=20, list [1]='apple'
print("Item at Position 0 and 1 is, list [0], list [1])
```

Output:-

```
Item at position 2=2.43
Item at position 2=500
 Item at Position 0 and 1 is abcd 147
```

Item at Position 0 and 1 is 20 apple

Built-in List Methods:-

1. **list.append(obj)** - This method appends an object obj passed to the existing list.

Example :-

list= ['abcd', 147, 2.43,'Tom']

print("Old List before Append:", list)

list.append (100)

print("New List after Append:", list)

Output:-

old List before Append: ['abed', 147, 2 43,"Tom"]

New List after Append: ['abed', 147, 2.43,'Tom', 100)

2. list.count(obj) - Returns how many times the object obj appears in a list

Example :-

list =['abcd', 147,2 .43,'Tom', 147,200,147]

print("The number of times", 147," appears in", list, "=",list.count (147))

Output:-

The number of times 147 appears in ['abcd', 147, 2.43,' Tom', 147,200,147]=3

3. list.remove(obj) - Removes object obj from the list.

Example :-

list1= ['abcd', 147, 2.43, 'Tom']

list1.remove("Tom")

print (list1)

Output:-

['abcd', 147, 2.43]

4. list.index(obj) - Returns index of the object obj if found, otherwise raise an exception indicating that value does not exist.

Example :-

list1= ['abcd', 147, 2.43, 'Tom']

print (list1.index (2.43))

Output:-

2

5. list.extend(seq) - Appends the contents in a sequence seq passed to a list.

Example :-

list1=['abcd', 147, 2.43, 'Tom']

list2= ['def', 100]

list1.extend (list2)

print (list1)

Output:-

['abcd', 147, 2.43, 'Tom', 'def', 100]

6. list.reverse() - Reverses objects in a list.

Example:-

list1 = ['abcd', 147, 2.43,' Tom']

list1.reverse()

print (list1)

Output:-

['Tom', 2.43, 147, 'abcd']

7. list.insert(index,obj) - Returns a list with object obj inserted at the given index.

Example :-

list1=['abcd', 147, 2.43,' Tom']

print ("List before insertion:",list1)

list1.insert (2,222)

print("List after insertion:", list1)

Output:

List before insertion:['abcd',149, 2.43,'Tom']

List after insertions :['abcd', 147, 222, 2.43, 'Tom']

8. list.sort([Key=None, Reverse= False])- Sorts the items in a list and returns the list. If a function is provided, it will compare using the function provided

Example:-

list1= [840, 147, 2.43 ,100]

print ("List before sorting: list1)

list1.sort ()

print ("List after sorting in ascending order ,list1)

Output:-

List before sorting: [840, 147, 2.43, 100]

List after sorting in ascending order: [2.43 100,147,840]

The following example illustrates how to sort the list in descending order.

Example2:

list1=[890, 147, 2.43,103]

print("List before sorting:",list1)

list1.sort (reverse=True)

print("List after sorting in descending order:",list1)

Output:-

List before sorting. [840, 147, 2:43, 100]

List after sorting in descending order :[840, 147, 100, 2.43]

9. list.pop(index))-Removes or returns the last object obj from a list. We can even pop out any item in a list with the index specified

Example :

```
list1= ['abcd', 147, 2.43,'Tom']
print ("List before poping", list1)
list1.pop(-1)
print ("List after poping:",list1)
item=list1.pop(-3)
print("Popped item:", item)
print("List after poping:", list1)
```

Output:-

List before poping :['abcd', 147, 2.43, "Tom"]

List after poping: ['abcd', 147, 2.43]

poped item= abcd

List after poping: [147, 2.43]

10. list.clear() -Removes all items from a list. Example :-

```
list1= ['abcd', 147, 2.43, 'Tom']
print("List before clearing:", list1)
list1.clear()
print("List after clearing:",list1)
```

Output:-

List before clearing: ['abcd', 147, 2.43, Tom'] List after clearing: [ ]

11. list.copy() - Returns a copy of the list

Example :-

```
list1= ['abcd', 147, 2.43, Tom']
print("List before clearing:",list1)
list2=list1.copy()
list1.clear()
print ("List after clearing:,list1)
print ("Copy of the list:",list2)
```

Output:-

List before clearing: ["abcd', 147, 2.43, Tom"]

List after clearing: [ ]

Copy of the list: ['abcd', 147, 2.43. Tom']

DICTIONARY

Dictionary is an unordered collection of key value pairs. It is generally used when we have a huge amount of data. Key and value can be of any type Keys are usually numbers or strings Values, on the other hand, can be any arbitrary Python object Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays"

Dictionaries are enclosed by curly braces {} and values can be assigned and accessed

using square braces [ ].

Properties of Dictionary Keys

1. More than one entry per key is not allowed ie, no duplicate key is allowed. When duplicate keys are encountered during assignment, the last assignment is taken.

2. Keys are immutable. This means keys can be numbers, strings or tuple. But it does not permit mutable objects like lists.

Built-in Dictionary Methods

1. dict.clear()- Removes all elements of dictionary dict.

Example:-

```
dict1={' Name': 'Tom', 'Age':20,'Height ':160}
print(dict1)
dict1.clear()
print (dict1)
```

Output:-

```
Age: 20,' Name':'Tom's, 'Height' :160
{ }
```

2. dict.copy() - Returns a copy of the dictionary dict.

Example :-

```
dict1={' Name': 'Tom', 'Age': 20, 'Height': 160}
print (dict1)
dict2=dict1.copy ()
print (dict2)
```

Output:-

```
{"Age":20 ,"Name": "Tom", "Height: 160}
{"Age": 20,"Name": "Tom", "Height: 160}
```

3. dict.keys() - Returns a list of keys in dictionary dict.

Example :-

```
dict1= {"Name":"Tom", "Age":20, "Height:160}
print (dict1)
print("keys in Dictionary: ",dict1.keys())
print("Keys in sorted order, sorted( dict1.keys ()))
```

Output:-

```
{"Age": 20, "Name":"Tom"," Height ":160}
 Keys in Dictionary {'Age', 'Name,'Height'}
{"Age","Height","Name"}
```

The values of the keys will be displayed in a random order. In order to retrieve the keys in sorted order, we can use the sorted function. But for using the sorted function, all the keys should of the same type.

4. dict.values() - This method returns list of all values available in a dictionary.

Example :-

```
dict1={"Name":" Tom", "Age":20, "Height":160}
print (dict1)
print("Values in Dictionary:",dict1.values())
print ("Values in sorted order:", sorted (dict1.values()))
```

Output:

{Age: 20, 'Name': Tom', 'Height': 160}

Values in Dictionary: {20, 'Tom', 160}

Values in sorted order: {20, 160,'Tom'}

The values will be displayed in a random order. In order to retrieve the values in sorted order, we can use the sorted function. But for using the sorted function, all the values should of the same type.

5. dict.items() - Returns a list of dictionary dict's(key,value) tuple pairs.

Example :-

dict1= {'Name':'Tom', 'Age':20, 'Height':160}

print (dict1)

print ("Items in Dictionary:", dict1.items ())

Output:-

{'Age': 20, 'Name': 'Tom', 'Height: 160}

Items in Dictionary: [('Age, 20),("Name", "Tom"), ('Height, 160)]

6. dict1.update(dict2) - The dictionary dict2's key-value pair will be updated in dictionary dict1.

Example:-

dict1={'Name':'Tom', 'Age' :20, 'Height' :160)

print (dict1)

dict2={'Weight': 60}

print (dict2)

dict1.update (dict2)

print("Dict1 updated Dict2 :",dict1)

Output:-

{'Age': 20, 'Name':' Tom', 'Height': 160}

{'Weight': 60}

Dict1 updated Dict2: {'Age': 20, 'Name': ' Tom','Weight': 60,'Height': 160}

7. dict.get(key, default-None) -

Returns the value corresponding to the key specified and if the key specified is not in the dictionary, it returns the default value.

Example :-

dict1= {'Name': 'Tom', 'Age':20, 'Height':160}

print (dict1)

print ("Dict1.get ('Age') :",dict1.get('Age'))

print ("Dict1_get('Phone'):",dict1.get ('Phone', 0))

Output :-

{'Age': 20, 'Name': 'Tom', 'Height': 160}

Dict1.get (key) : 20

Dict1.get (key) : 0

8. dict.setdefault(key,default-None) -

Similar to dict.get(key,default=None) but will set the key with the value passed and if key is not in the dictionary, it will set with the default value.

Example :-

```
dict1= {'Name':'Tom', 'Age':20, 'Height':160}

print (dict1)

print("Dict1.setdefault ('Age') :", dict1.setdefault ('Age'))


print(" Dict1 setdefault ('Phone') :",dict1.setdefault ('Phone', 0))
```

Output

```
{'Age': 20, 'Name': 'Tom', 'Height': 160}


Dict1 setdefault ('Age') : 20

 Dict1.setdefault ('Phone') : 0
```

9. dict.fromkeys(seq,[val])-

Creates a new dictionary from sequence seq and values from val.

Example :-

```
 list= ['Name', 'Age', 'Height']

dict= dict.fromkeys (list)

print ("New Dictionary:", dict)
```

Output:-

New Dictionary: {'Age': None, 'Name': None, Height': None}

SET:-

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces {}. It can have any number of items and they may be of different types (integer, float, tuple, string etc.). Items in a set are not ordered. Since they are unordered we cannot access or change an element of set using indexing or slicing. We can perform set operations like union, intersection, difference on two sets. Set have unique values. They eliminate duplicates. The slicing operator [] does not work with sets. An empty set is created by the function set().

Example:-

```
s1={1,2,3}

print (s1)

s2={1,2,3,2,1,2}

print (s2)

s3={1, 2.4, 'apple', 'Tom', 3}

print (s3)


#s4={1,2, [3,4]} #sets cannot have mutable items #print s4 # Hence not permitted

s5=set ([1,2,3,4])

print (5)
```

Output:-

```
{1, 2, 3}

 {1, 2, 3}

{1, 3, 2.4, 'apple, Tom'}
```

{1, 2, 3, 4}

Built-in Set Methods

1. set.add(obj) -

Adds an element obj to a set.

Example :-

```
set1={3,8,2,6}
 print ("Set before addition:", set1)
set1.add(9)
print ("Set after addition:", set1)
```

Output:-

Set before addition: {8, 2, 3, 6}

Set after addition: {8, 9, 2, 3, 6}

2. set.remove(obj) -

Removes an element obj from the set. Raises KeyError if the set is empty

Example :-

```
set1={3,8,2,6}
print ("Set before deletion:", set1)
set1.remove (8)
print ("Set after deletion:", set1)
```

Output:-

Set before deletion: {8, 2, 3, 6}

Set after deletion: {2, 3, 6}

3. set.discard(obj)-

Removes an element obj from the set. Nothing happens if the element to be deleted is not in the set.

Example:-

```
set1={3,8,2,6}
print ("Set before discard:",set1)
set1.discard (8)
print ("Set after discard:", set1)
# Element is present
set1.discard (9)
print ("Set after discard:",set1)
#Element is not present
```

Output:-

Set before discard: {8, 2, 3, 6)

Set after discard: {2, 3, 6}

Set after diacard: {2, 3, 6}

4.set.pop()-

Removes and returns an arbitrary set element. Raises KeyError if the set is empty.

Example :-

```
set1={3,8,2,6}
print("Set before pop:",set1)
set1.pop()
print("Set after poping:", set1)
```

Output:-

Set before pop: {8, 2, 3, 6}

Set after poping: {2, 3,6}

5. set1.union(set2) -

Returns the union of two sets as a new set.

Example :-

```
set1={3,8,2,6}
print ("Set1:", set1)
set2={4,2,1,9}
print("Set 2:",set2)
set3=set1.union (set2)
 print("Union:", set3)
```

Output:-

Set1: {8, 2, 3, 6}

Set2:{9, 2, 4, 1}

Union: {1, 2, 3, 4, 6, 8, 9}

6. set1.update(set2) -

Update a set with the union of itself and others. The result will be stored in set1.

Example:-

```
set1={3,8,2,6}
print ("Set1:",set1)
set2={4,2,1,9}
print ("Set 2:", set 2)
set1.update (set2)
print ("Update Method:", set1)
```

Output:-

Set1: {8, 2, 3, 6}

Set 2: {9, 2, 4, 1}

Update Method: {1,2,3, 4, 6, 8, 9}

7. set1.intersection(set2) -

Returns the intersection of two sets as a new

Example :-

```
set1={3,8,2,6}
print ("Set1:",set1)
 set2={4,2,1,9}
print ("Set 2:",set2)
set3 =set1.intersection (set2)
print("Intersection:", set 3)
```

Output:-

Set1: {8, 2, 3, 6}

Set2:{9, 2, 4, 1}

Intersection: {2}

8. set1.intersection_update()-

Update the set with the intersection of itself and another. The result will be stored in set1.

Example :-

```
set1={3,8,2,6}
print ("Set 1:", set1)
 set2={4,2,1,9}
print ("Set2:", set2)
set1.intersection_update (set2)
print ("Intersection:", set1)
```

Output:-

Set1:{8, 2, 3, 6}

Set2: {9, 2, 4, 1}

Intersection update: {2}

9. set1.difference(set2) -

Returns the difference of two or more sets into a new set.

Example :-

```
set1={3,8,2, 6}
print("Set1:", set1)
set2={4,2,1,9}
print ("Set 2:", set 2)
 set3= set1.difference (set2)
print ("Difference:", set3)
```

Output:-

Set1: {8, 2, 3, 6}

Set 2:{9, 2, 4, 1}

Difference: {8, 3, 6}

10. set1.difference_update(set2) -

Remove all elements of another set set2 from set1 and the result is stored in set1.

Example:-

```
set1={3,8,2,6}
print("Set1:", set1)
set2={4,2,1,9}
print ("Set2:", set2)
set1.difference_update (set2)
 print ("Difference Update:", set1)
```

Output:-

Set1: {8, 2, 3, 6}

Set2:{9, 2, 4, 1}

Difference Update: {8, 3, 6}

11. set1.symmetric_difference(set2)-

Return the symmetric difference of two sets as a new set.

Example :-

```
set1={3,8,2,6}
print("Set1:", set1)
set2={4,2,1,9}
print("Set2:", set2)
set3 =set1.symmetric_difference (set2)
print ("Symmetric Difference ", set3)
```

Output:-

Set1: {8, 2, 3, 6}

Set 2: {9, 2, 4, 1}

Symmetric Difference: {1, 3, 4, 6, 8, 9}

12. set1.symmetric _difference_update(set2) - Update a set with the symmetric difference of itself and another.

Example :-

```
set1={3,8,2,6}
print ("Set1:",set1)
set2={4,2,1,9}
print("Set2:", set2)
set1.symmetric_difference_update (set2)
print ("Symmetric Difference Update:", set1)
```

Output:-

Set1: {8, 2, 3, 6}

Set 2: {9, 2, 4, 11}

Symmetric Difference Update: {1, 3, 4, 6, 8, 9}

13. set1.isdisjoint(set2) -

Returns True if two sets have a null intersection.

Example:-

```
set1={3,8,2,6}
print ("Set 1:",set1)
set2={4,7,1,9}
print("Set 2:", set 2)
print("Result of set1 isdisjoint (set2):", set1.isdisjoint(set2))
```

Output:-

Set1: {8, 2, 3, 6}

Set 2: {9, 7, 4, 1}

Result of set1.isdiajoint (set 2): True

14. set1.issubset(set2)

Returns True if set1 is a subset of set2.

Example :-

```
set1={3,8}
print ("Set 1:", set1)
set2={3,8,4,7,1,9}
print ("Set2:",set2)
print("Result of seti.issubset (set2):", set1.issubset (set2))
```

Output:-

Set1:{8, 3}

Set2: {1, 3, 4, 7, 8, 9}

Result of seti.issubset (set2): True

15. set1.issuperset(set2) -

Returns True, if set1 is a super set of set2.

Example :-

```
set1={3,8,4,6}
print ("Set 1:",set1)
set2={3,8}
print ("Set 2:", set2)
print ("Result of seti.issuperset (set 2):", set1.issuperset (set2))
```

Output:-

Set 1: {8, 3, 4, 6}

Set 2: {8, 3}

Result of set1.issuperset (set 2): True

Mutable And Immutable objects:-

Objects in Python:

In Python, everything is treated as an object. Every object has these three attributes:

Identity – This refers to the address that the object refers to in the computer's memory.

Type – This refers to the kind of object that is created. For example- integer, list, string etc.

Value – This refers to the value stored by the object. For example – List=[1,2,3] would hold the numbers 1,2 and 3

While ID and Type cannot be changed once it's created, values can be changed for Mutable objects.

Mutable and immutable:

An object whose internal state can be changed is mutable. On the other hand, immutable doesn't allow any change in the object once it has been created.

Objects of built-in type that are mutable are:

- Lists
- Sets
- Dictionaries
- User-Defined Classes (It purely depends upon the user to define the characteristics)

Objects of built-in type that are immutable are:

- Numbers (Integer, Rational, Float, Decimal, Complex & Booleans)
- Strings

- Tuples
- Frozen Sets
- User-Defined Classes (It purely depends upon the user to define the characteristics)

Object mutability is one of the characteristics that makes Python a dynamically typed language.

Mutable Object	Immutable Object
State of the object can be modified after it is created.	State of the object can't be modified once it is created.
They are not thread safe.	They are thread safe
Mutable classes are not final.	It is important to make the class final before creating an immutable object.

Example of mutable object list:-

```
cities = ['Delhi', 'Mumbai', 'Kolkata']
for city in cities:
        print(city, end=', ')
cities.append('Chennai')
for city in cities:
    print(city, end=', ')
```

Output:-

Delhi, Mumbai, Kolkata

Delhi, Mumbai, Kolkata, Chennai

Example of immutable object tuple:-

```
foo = (1, 2)
foo[0] = 3
```

Output:-

TypeError: 'tuple' object does not support item assignment