Chapter 2: Variables and Operators

Essence of 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. The declaration 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 that variable.

The variable x is set to 30. *x = x + 1* will increase the value of this variable by 1.

```
>>> x = 30
>>> x = x + 1
>>> print(x)
31
```

The type of a variable can change during the execution of a script.

```
>>> x = 30
>>> x = 35
>>> x = "hello"
>>> x = 5.6
```

Variable nomenclature and properties

Variable as a storage unit.

Since variables hold reference to data for manipulation, popular database operations like Insert, Update, and Delete should also be possible on variable.

id() function:

This is used to determine the id of a variable. It is value based and not variable name based.

```
>>>x=5
>>>id(x)
140566819039352
>>>y=6
>>>id(y)
140566819039328
>>>z=5
>>>id(z)
140566819039352
```

Here two different variables x and z have same id because they refer to same value.

Variable Nomenclature

1. Variable name must start with [_a-z, A-Z] then followed by [_a-z, A-Z, 0-9]
2. Variable names are case sensitive
3. Keywords cannot be used as Python Variable names

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

['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try', 'while','with', 'yield']

Example

```
_a=3 #valid
Apple = "apple" #valid
0time = 0 #in-invalid
```

Python Variable properties

1. Implicitly defined
Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

2. Strongly typed
Python is strongly typed as the interpreter keeps track of all variables types. Compiler here cannot tell which type a variable refers to.

Type() function

```
type(object) -> the object's type
>>>x="name"
>>>type(x)
<type 'str'>
```

Multiple assignment

```
Ex: >>>a=b=c=1
>>>a,b,c= 1,2,"name
```

Python default variable '_'

'_' is a variable auto defined by Python and it stores last calculation. It is like ANS button on calculator.

```
>>> 20 + 30
50
>>> 50 + _
100
```

Operators

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

1. Arithmetic operators
2. Comparison operators
3. Assignment operators
4. Logical operators
5. Bitwise operators
6. Membership operators
7. Identity operators

Arithmetic Operators

Arithmetic operators are used for performing basic arithmetic operations. The following table shows the arithmetic operators supported by python.

Operator	Operations	Description
+	Addition	Adds values on either side of 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 the right hand operand
%	Modulus	Divides left hand operand by right hand operand and returns remainder`
**	Exponent	Performs exponential calculation on operands
//	Floor division	The division of Operands where the result is the quotient in which the digits after the decimal point are removed

Comparison Operator

These operators compare the values on either sides of them and decide the relation among them. They are also called as relational operators. The following table shows the comparison operators supported by python.

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

Assignment Operators

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

supported by python. The following table shows the assignment operators supported by python.

Operator	Description
=	Assigns values from right side operand 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 right operand by the left operand and assign the result to left operand
%=	It takes modulus using two operands and assign the result to left operand
**=	Performs exponential calculation on operands and assign values to the left
//=	It performs floor division on operators and assign value to the left operand

Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. The following table shows the bitwise operators supported by python.

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	It is unary and ahs the effect of 'flipping' bits.
	complement		
<<	Binary	Left	The left operands value is moved left by the number of bits
	Shift		specified by the right operand.
>>	Binary	Right	The left operands value is moved right by the number of bits
	Shift		specified by the right operand.

Logical Operators

Table shows the various logical operators supported by python language.

Operator	Operation	Description
And	Logical And	If both the operands are true then condition becomes true
Or	Logical OR	If any of the operand is true then condition becomes true
not	Logical NOT	Used to reverse the logical state of its operand

Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, and tuples. There are two membership operators supported by python which are described in the following table.

Operator	Description
In	Evaluates to true if the variable on either side of the operator point to the sameobject and false otherwise
not in	Evaluates to true if it does not finds a variable in the specific sequence and falseotherwise.

Identity Operator

Identity operators compare the memory locations of two objects. There are two identity operators as shown in the following table.

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

Operator Precedence

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

Operator	Description
**	Exponentiation
~,+,-	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

Chapter 3: Rules of Python programming

Comments in Python

Comments are very important while writing a program. It describes what the source code hasdone. Comments are for programmers for better understanding of a program. In python, weuse the hash (#) symbol to start writing a comment. All the character after the #and up to theend of the physical line is part of the comment.

Example program

```
>>>#this is demo of comment
>>>print("hello")
```

For multiline comments use triple quotes, either ''' or """. The statements within the start and end of triple quotes are considered as comment.

Indentation in Python

Most of the programming languages like c, c++ use braces {} to define a block of code. Python uses indentation. A code block starts with indentation and ends with the first unintended 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.

Example program

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

Multi-line statement

Instructions that a python interpreter can execute are called statements. In python, end of a statement is marked by a newline character, but we can make statement extend over multiple lines with the line continuation character (\).

```
grand_total = first_item +\
second_item+\
third_item
```

Keywords in Python

These are keywords reserved by the programming language and prevent the user or the programmer from using it as it as an identifier in a program. There are 33 keywords in python.
This number may vary with different versions. To retrieve the keywords in python the

following code can be given at the prompt.

```
>>> import keyword
>>> print(keyword.kwlist)
```

The following list in table shows the python keywords.

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		

Basic input/output methods

Displaying the Output

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

Example program

```
>>>a=2
>>>print("the value of a is:",a)
the value of a is 2
>>>
```

By default a space is added after the text and before the value of variable a. The Syntax of print function is given below print *objects, sep=' ', end='\n', file=sys.stdout,flush=False

Here objects are the value to be printed. 'sep' is the separator used between the values. Spaces character is the default separator. 'end' is printed after printing all the values.
The default value of end is a newline. 'file' is the object where the values are printed and its default value is sys.stdout (screen). Flush determines whether the Output stream needs to be flushed for any waiting Output.

Reading the Input

Python provides two built-in functions to read from standard input (keyboard). These functions are raw_input and input.

Example program

```
str = input("enter your name:")
print("your name is: ",str)
```

Output

```
enter your name: abc
```

your name is: abc

import* and *dir()

Import function

When the program grows bigger or when there are segments of code that is frequently used, it can be stored in different modules. A module is a file containing python definition and statements. Python modules have a file name and end with expression .py. Definition inside a module can be imported to another module or the interactive interpreter in python. We use the import keyword to do this.

Example program

```
>>>import math
>>>math.pi

3.141592653589793
```

Chapter 4: Control flow and Functions

Decision making with *if-elif-else*

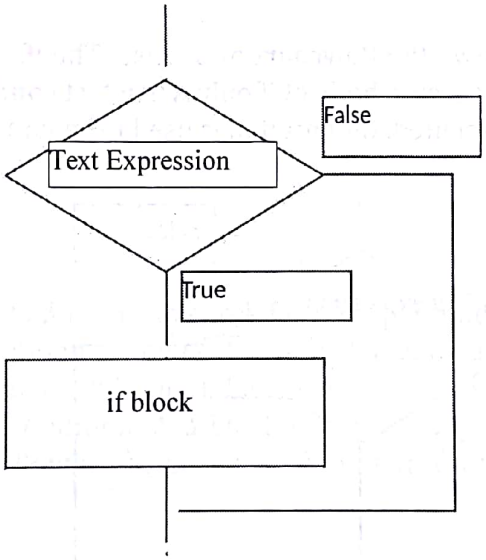
Decision making is required when we want to execute a code only if a certain condition is satisfied. The if...elif....else statement is used in python for decision making.

if statement

Syntax

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

The following figure shows the flowchart of the *if* statement.



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 un-indented 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")
```

Output1

enter the number: 0 Zero
this is always printed

Output2

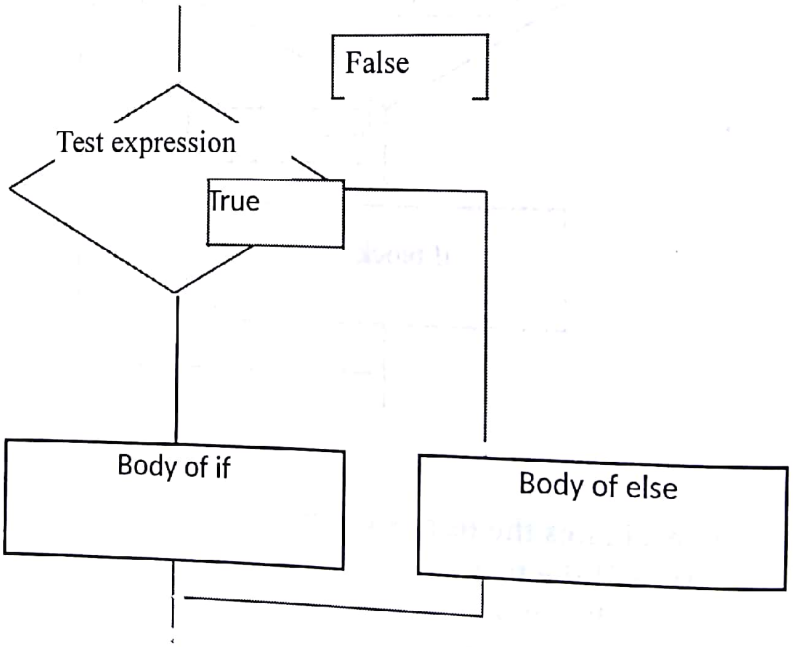
enter the number:1
this is always printed

if....else statement

Syntax

```
if test expression:
        body of if
else:
        body of else
```

The following figure shows the flowchart of if....else. The if...else statement evaluates the 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.



Example program

```
num = int(input("enter a number:"))
if num>=0:
        print("positive integer")
else:
        print("negative integer")
```

Output1

enter a number: 5
positive integer

Output2

enter a number: -2
negative integer

if...elif...else statement

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 conditions. 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. A if block can have only one else block. But it can have multiple elif blocks. Following fig shows the flow chart for if...elif...else statement.

Example program

```
num = int(input("enter a number:"))
if num>o:
        print("positive integer") elif num == 0:
        print("zero")
else:
        print("negative integer")
```

Output1

```
enter a number:5
positive integer
```

Output2

```
enter a number: 0
zero
```

Output3

```
enter a number:-7
negative number
```

Nested if statement

We can have if...elif...else statements 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 num>=0:
        if num == 0:
                print("zero")
else:
        print("positive integer")
else:
        print("negative integer")
```

Output1

```
enter a number:5
positive integer
```

Output2

```
enter a number: 0
zero
```

Output3

```
enter a number:-7
negative number
```

for Loops and range( )

Generally, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on. There will be situations when we need to execute a block of code several number of times. 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.

for loop

The for loop in python is used to iterate over a sequence or other objects that can be iterated.

Iterating over a sequence is called traversal. Below figure shows the flow chart of for loop.

Syntax

```
for item in sequence:
    body of for
```

Here item is a 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.

Example program

```
num = [2,3,4,5]
sum = 0
for item in num:
    sum = sum+item
print("the sum is",sum)
```

Output

```
the sum is 14
```

range()function

We can use range() function in for loop to iterate through a sequence of number. It can be combined with the len() function to iterate through a sequence using indexing. len() function is used to find the length of a string or number of elements in a list, tuple, set etc.

Example program

```
flowers = ['rose','lotus','lily']
for i in range(len(flowers)):
        print("current flower: ",flowers[i])

<Output>
current flower: rose
current flower: lotus
current flower: lily
```

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 . We can also define the start ,stop and step_size as range(start, stop, step_size). The default value of step_size is 1, if not provided. This function does not store the values in memory. It keeps track of the start, stop, step_size and generates the next number.

Example program

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

<Output>
number = 2
number = 4
number = 6
number = 8
```

while loop

The while loop in python is used to iterate over a block of code as long as the test expression is true. We generally use this loop when we don't know the number of times to iterate in advance. Below figure shows the flow chart of a while loop.

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 condition is true. After first iteration the test expression is checked again. This process is continued until the test expression evaluate to false. In python, the body of the while loop is determined through indentation. Body starts with indentation and the first un-indented line shows the end.

Example program

```
n = int(input("enter the limit:")) sum = 0
i=1
while(i<=n): sum= sum+i i = i+1
        print("the sum of first",n,"natural numbers is",sum)

<Output>
enter the limit:5
the sum of first 5 natural numbers is 15
```

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:
        statment(s)
        statemnet(s)
```

Syntax for nested while loop

```
while expression:
    while expression:
        statment(s)
        statemnet(s)
```

Example program

```
import math

n = int(input("enter a limit:"))
for i in range(1,n):
        k=int(math.sqrt(n))
        for j in range(2,K+1):
                if i%j==0:break else:
                print(i)
```

Output

```
enter the limit:12
1 2 3 5 7 11
```

The limit entered by the user is stored in n. To find whether a number is prime number, the logic used si to divide that number from 2 to square root of that number. If the remainder of this division is zero at any time, that number is skipped and moved to next number. A complete division shows that the number is not prime number. If the remainder is not zero at any time, it shows that it is a prime number.

The outer for loop starts from 1 to the input entered by the user. Initially i=1. A variable k is used to store the half of the number. The inner for loop is used to find whether the number is completely divisible by any number between 2 and half of that number (k). If it is completely divisible by any number between 2 and k, the number is not a prime, else the number is considered prime. The same steps are repeated until the limit entered by the user is reached.

Control Statements

Control statements change the execution from normal sequence. 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 and continue statements are used in these cases. Python supports following three control statements.

1. break
2. continue
3. pass

break statement

The break statement terminates the loop containing it. Control of the program is passed to the statement immediately after the body of the loop. If it is nested loop, break will terminate the innermost loop. It can be used for both for and while loops. Following figure shows the flow chart of break statement

Example Program

```
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 continue 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 loop. Following figure shows the flow chart of continue statement.

Example program

```
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. Suppose we have a loop or a function that is not implemented yet, but want to implement it in the future. The function or loop cannot have an empty body. The interpreter will not allow this. So, we use the pass statement to construct a body that does nothing.

Example

```
for val in sequence:
      pass
```

Introduction to Functions

A group of related statements to perform a specific task is known as a function. Python provides two types of functions

- Built-in functions
- User-defined functions

Function Definition

The following specifies simple rules for defining a function.

Function block or function header begins with a keyword def followed by the function name and parentheses ( ).
Any input parameters or arguments should be placed within these parameters. We can also define parameters inside these parentheses.
The first string after function header is called the docstring and is short for documentation string. It is used t explain in brief, what a function does. Although optional, Documentation is good programming practice.
The code block within every function starts with a colon (:) and is indented.
The return statement [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return none.

```
<Syntax>

def functionname(parameters):
        "function_docstring"
        function_suite
        return [expression]

<Example program>

def sum(a,b):
        sum = a+b
        return sum
```

Function Calling

After defining a function, we can call the function from another function or directly from python prompt. The order of the parameters specified in the function definition should be preserved in function call also.

```
<Example program>

a = int(input("enter first number:"))
b = int(input("enter second number:"))
s = sum(a,b)
print("sum of",a,"and",b,"is", s)
```

```
<Output>

enter first number: 4
enter second number: 3
sum of 4 and 3 is 7
```

All the parameters in python are passed by reference. It means if we change a parameter which refers to within a function, the change also reflects back in the calling function. Parameters defined inside a function have local scope. The scope of a variable determines the portion of the program where we can access a particular identifier. They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous call.

```
<Example program>

def value_change(a):
        a=10
        print("value inside function:",a)
        return
a= int(input("enter a number:")
value_change(a)
print("value outside function:",a)

<Output>

enter a number: 2
value inside function: 10
value outside function: 2
```

Function Arguments

We can call function by any of the following 4 arguments

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required Arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition. Consider the following example.

```
<Example program>

def value_change(a):
        a=10
        print("value inside function:",a)
        return
```

```
a= int(input("enter a number:")
value_change()
print("value outside function:",a)

<Output>

enter a number: 4

Traceback (most recent call last): File "main.py", line 7, in <module>
value_change()
TypeError: value_change() take exactly 1 argument (0 given)
```

The above example contains a function which requires 1 parameter. In the main program code, the function is called without passing the parameter. Hence it resulted in an error.

Keyword Arguments

When we use keyword arguments in a function call, the caller identifies the argument by parameter name, this allows us to skip arguments or place them out of order because the python interpreter is able to use keywords provided to match the values with parameters

```
<Example program>

def studentinfo(rollno,name,course):
        print("roll no: ",rollno)
        print("name: ",name)
        print("course: ",course)
        return

studentinfo(course="UG",rollno=50,name="jack")

<Output>

roll no: 50
name: Jack
course: UG
```

Default Arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example shows how default arguments are invoked.

```
<Example program>

def studentinfo(rollno,name,course="UG"):
        print("roll no: ",rollno)
        print("name: ",name)
        print("course: ",course)
        return
```

```
studentinfo(course="UG",rollno=50,name="jack")
studentinfo(rollno=51,name="Tom")

<Output>

roll no: 50
name: Jack
course: UG

roll no: 51
name: Tom
course: UG
```

In the above example the first function call to studentinfo passes the three parameters. In case of second function call to studentinfo, the parameter course is omitted. Hence it takes the default value "UG" given to course in the function.

Variable-Length Arguments

In some cases we may need to process a function for more arguments than specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments. An asterisk (*) is placed before the variable name that holds the value of all non-keyword variable arguments. The tuple remains empty if no additional arguments are specified during the function call. The following shows the Syntax of a function with variable-length arguments.

```
<Syntax>

def functionname([formal_srguments,]):
        "function_docstring"
        function_suite
        return [expression]

<Example program>

def variablelengthfunction(*drgument):
        print("result:",)
        for i in argument:
        print(i)
        return

variablelengthfunction(10)

variablelengthfunction(10,30,50,80)

<Output>

Result: 10
Result: 10
```

```
30
20
80
```

Anonymous Functions (Lambda Functions)

In python, anonymous function is a function that is defined without a name. While normal functions are defined using the def keyword, in python anonymous functions are defined using lambda keyword. Hence, anonymous functions are also called as lambda functions. The following are the characteristics of lambda functions.

Lambda functions can also take any number of arguments but return only one value in the form of an expression.

It cannot contain multiple expressions.

It cannot have commands.

A lambda function cannot be a direct call to print because lambda requires an expression.

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

Lambda functions are not equivalent to inline functions in c or c++.

```
<Syntax>

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

Example Program

square = lambda x: x*x;
n = int(input("enter a number: "))
print("square of",n,"is",square(n))

<Output>

enter a number: 2 square of 2 is 4
```

Uses of Lambda function

We use lambda function when we require a nameless function for a short period of time. In python we generally use it 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.

Functions with more than one return value

Python has a strong mechanism of returning more than one value at a time. This is a very flexible when the function needs to return more than one value. Instead of writing separate functions for returning individual values, we can return all values within same function. The following shows an example for function returning more than one value.

```
<Example program>

def calc(a,b):
        sum = a+b diff = a-b
        return sum,diff
a = int(input("enter first number:"))
b = int(input("enter second number:")) s,d=calc(a,b)
print("sum=",s)
print("Difference=",d)

<Output>

enter first number: 10
enter second number: 5
sum = 15
Difference = 5
```

Chapter 5: Python data types – String

Strings in python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Strings can be taken using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the string and ending at -1. The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. The % operator is used for string formatting.

Example program

```
Str = 'welcome to python programming'

Print(str)              #prints complete string
Print(str[0])           #prints  first character of the
                        #string
Print(str[11:17])       #prints characters starting from
                        # 11th  to 17th
Print(str[11:])         #prints string starting
                        #from 11th  character
Print(str*2)            #prints string two times
Print(str + "session")   #prints concatenated srting
```

Output

```
Welcome to python programming
W
Python
Python programming
Welcome to python programming welcome to python Programming
Welcome to python programming session
```

Escape Characters

An escape character is a character that gets interpreted when placed in single and double quotes. They are represented using backslash notation. These characters are non-printable. The following table shows the most commonly used escape characters and their description.

Escape characters	Description
\a	Bell or alert
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage return
\s	Space
\t	Tab
\v	Vertical tab

String Formatting Operator

This operator is unique to strings and is similar to the formatting operations of the printf() function of c programming language. The following table shows various format symbols and their conversion.

Format symbols	Conversion
%c	Character
%s	String
%i	Signed decimal integer
%d	signed decimal integer
%u	Unsigned decimal integer
%o	Octal integer
%x	Hexadecimal integer(lowercase letters)
%X	Hexadecimal integer(uppercase)
%e	Exponential notation with lowercase letter 'e'
%E	Exponential notation with uppercase letter 'E'
%f	Floating-point real number

String Formatting Functions

1) len(string)

Example program

```
S='Learning python is fun!'
Print("Length of ", s, "is", len(s))
```

Output

```
Length of Learning python is fun! is 23
```

2) lower()

Example program

```
S='Learning python is fun!'
Print(s.lower())
```

Output

```
learning python is fun!
```

3) upper()

Example program

```
S='Learning python is fun!'
Print(s.upper())
```

Output

```
LEARNING PYTHON IS FUN!
```

4) swapcase()

Example program

```
S='LEARning PYTHON is
fun!' Print(s.swapcase())
```

Output

```
learning python IS FUN!
```

5) capitalize()

Example program

```
S='learning python is fun!'
Print(s.capitalize())
```

Output

```
Learning python is fun!
```

6) title()

Example program

```
S='learning python is fun!'
Print(s.title())
```

Output

```
Learning Python Is Fun!
```

7) lstrip()

Example program

```
S=' learning python is fun!'
Print(s.lstrip())
S='*******learning python is fun!'
Print(s.lstrip('*')
```

Output

```
learning python is fun!
learning python is fun!
```

8) rstrip()

Example program

```
S='learning python is fun!      '
Printf(s.rstrip())
S='learning python is fun!*****'
```

```
Printf(s.rstrip('*'))
```

Output

```
learning python is fun!
learning python is fun!
```

9) strip()

Example program

```
S=' learning python is fun! '
Print(s.strip())
S='******learning python is fun!******'
Printf(s.strip())
```

Output

```
learning python is fun!
learning python is fun!
```

10) max(str)

Example program

```
S='learning python is fun!'
Print('Maximum character is :", max(s))
```

Output

```
Maximum character is : y
```

11) min(str)

Example program

```
S='learning-python-is-fun!'
Print('Minimum character is :",min(s))
```

Output

```
Minimum character is : -
```

12) replace(olb,new[,max])

Example program

```
s="this is vary new. This is good".
Print(s.replace('is', 'was'))
Print(s.replace('is', 'was', 2))
```

Output

```
Thwas was very new. Thwas was good
Thwas was very new. This is good
```

13) center(width,fillchar)

Example program

```
s="This is python programming"
print(s.center(30, '*')
print(s.center(30))
```

Output

```
**This is python programming**

This is python programming
```

14) ljust(width[,fillchar])

Example program

```
S="this is python programming"
Print(s.ljust(30,'*'))
Print(s.ljust(30))
```

Output

```
This is python programming****

This is python programming
```

15) rjust(width[,fillchar])

Example program

```
S="this is python programming"
Print(s.rjust(30,'*'))
Print(s.rjust(30))
```

Output

```
****This is python programming

    This is python programming
```

16) zfill(width)

Example program

```
S="this is python programming"
Print(s.zfill(30))
```

Output

```
0000This is python programming
```

17) count(str,beg=0,end=len(string)) Example program

```
S="This is python programming"
Print(s.count('i',0,10))
Print(s.count('i',0,25))
```

Output

```
2
3
```

18) find(str,beg=0,end=len(string)) Example program

```
S="This is python programming"
Print(s.find('thon',0,25))
Print(s.find('thy'))
```

Output

```
10
```

```
-1
```

19) rfind(str,beg=0,end=len(string) Example

program

```
S="This is python programing"
Print(s.rfind('thon',0,25))
Print(s,rfind('thy')
```

Output

```
10

-1
```

20) index(str,beg=0,end=len(sring) Example

program

```
S="this is python programming"
Print(s.index('thon',0,25))
Print(s.index('thy'))
```

Output

```
10
Traceback (most recent call last):
File "main.py", line 4, in <module>
  Print s.index('thy')
ValueError: substring not found
```

21) rindex(str,beg=0,end=len(string) Example

program

```
S="This is python programming"
Print(s.rindex('thon',0,25))
Print(s.rindex('thy'))
```

Output

```
10
Traceback (most recent last):
File "main.py", line 4, in <module>
  Print s.index('thy')
ValueError: substring not found
```

22) startswith(suffix,beg=0,end=len(string) Example

program

```
s="python programming is fun"
print(s.startswith('is',10,21))
print(s.startswith('is',19,25))
```

Output

```
False
True
```

23) endswith(suffix,beg=0,end=len(string) Example program

```
S="python programming is fun"
Print(s.endswith('is',10,25))
S="python programming is fun"
Print(s.endswith('is',0,25))
```

Output

```
True
False
```

24) isdecimal()

Example program

```
S=u"This is python 1234"
Print(s.isdecimal())
S=u"123456"
Print(s.isdecimal())
```

Output

```
False
True
```

25) isalpha()

Example program

```
S="This is python1234"
Print(s.isalpha())
S="python"
Print(s.isalpha())
```

Output

```
False
True
```

26) isalnum()

Example program

```
S="***python1234"
Print(s.isalnum())
S="python1234"
Print(s.isalnum())
```

Output

```
False
True
```

27) isdigit()

Example program

```
s="***python1234"
print(s.isdigit())
```

```
s="123456"
print(s.isdigit())
```

Output

```
False
True
```

28) islower()

Example program

```
S="Python Programming"
Print(s.islower())
S="python Programming"
Print(s.islower())
```

Output

```
False
True
```

29) isupper()

Example program

```
S="Pyhton Programming"
Print(s.isupper())
S="PYTHON PrograMING"
Print(s.isupper())
```

Output

```
False
True
```

30) isnumeric()

Example program

```
S=u"Python Programming1234"
Print(s.isnumeric())
S=u"12345"
Print(s.isnumeric())
```

Output

```
False
True
```

31) isspace()

Example program

```
S="Python Programming"
Print(s.isspace())
S=" "
Print(s.isspace())
```

Output

```
False
```

```
True
```

32) istitle()

Example program

```
S="python Programming Is fun"
Print(s.istitle())
S="Python Programming Is Fun"
Print(s.istitle()
```

Output

```
False
True
```

33) expandtabs(tabsize=8)

Example program

```
S="Python\tProgramming\tis\tfun"
Print(s.expandtabs())
S="Python\tprogramming\tis\tfun"
Print(s.expandtabs(10))
```

Output

```
Python    Programming    is    fun
Python    Programming    is    fun
```

34) join(seq)

Example program

```
S="-"
Seq=("Python", "Programming")
Print(s.join(seq))
S="*"
Seq=("Python", "Programming")
Print(s.join(seq))
```

Output

```
Python-Programming
Python*Programming
```

35) split(str="",num=string.count(str)) Example program

```
S="Python Programming is fun"
Print(s.split(' '))
S="Python*Programming*is*fun"
Print(s.split('*'))
S="Python*Programming*is*fun"
Print(s.split('*',2))
```

Output

```
['Python', 'Programming', 'is', 'fun']
```

```
['Python', 'Programming', 'is', 'fun']
['Python', 'Programming', 'is*fun']
```

36) splitlines(num=striung.count('\n') Example

program

```
S="Python\nProgramming\nis\nfun"
Print(s.splitlines())
Print(s.splitlines(0)
Print(s,splitlines(1))
```

Output

```
['Python', 'Programming', 'is', 'fun']
['Python', 'Programming', 'is', 'fun']
['Python\n', 'Programming\n', 'is\n', 'fun']
```

Chapter 6: Python data types – List, Tuples

List

So far we have seen the primitive data types (**number,strings**) or one value data type. The next data types contain sequence of primitive values type or numeric and string values.

The free style combination that can be unordered and redundant is **list**.

Ordered and non-redundant sequence type is called **Set**.

Read only list, is called **tuple**. Once it is declared it is final, no new values can come in or go out.

List, Tuple, Sets and String follow a default indexing of 0,1,2,3... and -1,-2,-3... If we choose our own index key for every value in our list, the result is a key:value pair sequence, called **dictionary**.

List Definition

A list data type can hold multiple occurrences of numeric or string data type. The values are enclosed within square bracket and separated by comma [, ,]

Eg. list = [1,500, 2,20,2,"cat", "c", 2, 500]

List Properties

1) List is mutable or changeable i.e, it can grow in size and shrink

2) List has unsorted values

3) List can have redundant values

4) List has forward indexing starting with 0 and backward indexing starting with -1

List Operators

Concatenation operator '+' can be used to concatenate two given list operands.

Syntax: list + list

Example

```
>    x = [1,2,3,4] + [2,3,5]
```

> print x

[1,2,3,4,2,3,]

Repetition operator '*' can be used to repeat the list integer times.

Example

> x =[1,2,3] * 2

> print x

[1,2,3,1,2,3]

Membership operator 'in' checks if the variable value is in the list if yes it returns True Otherwise it returns False.

Membership operator 'not in' checks if the variable value is in the list if yes it returns False Otherwise it returns True

Example

>>>list = [6,4,1,4]

>>>5 in list

False

>>>4 in list

True

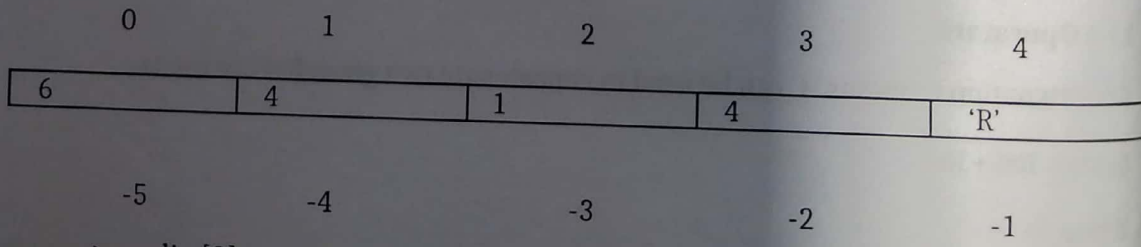
>>>4 not in list

False

List

Indexing list

= [6,4,1,4,'R']



> list[0]
> 6

```
>   list[-1]
>   'R'
>   list[2:4]

>   [1,4]
```

' : ' operator is used to fetch a section of list the value of left is equal to the lower index bound and the value on right is upper index bound + 1

List Functions:

To **create a list** simple type comma separated values within square bracket and assigns it to some variable name.

E.g. x = [6,4,1,4]

Assume, list = ["BR", 6,4,1,4]

Classification according to uses	List function names and explanation	Syntax and Example
To add values	insert(): To insert value at a particular index	insert(index,value) >>>list.insert(2,3) ["BR",6,3,4,1,4]
	append(): To insert value at the end index -1	append(value) >>>list.append('R') ["BR",6,4,1,4,'R']
	extend(): To insert one or more values at the end of a list	extend(list) >>>list2 = [4,5] >>>list.extend(list2) ["BR",6,4,1,4,4,5]
To remove values	remove():To remove 1st occurrence of any chosen value	remove(value) >>>list.remove(4) ["BR",6,1,4]
	pop() : To remove a value at given index. If index is not defined its default is -1	pop(index) >>>list.pop() ["BR",6,4,1] >>>list.pop(2) ["BR",6,1,4]
To arrange values	sort() : To sort the values of a list number –ve 0 +ve ascending followed by strings Alphabetically A-Z then a-z	sort() >>>list=[9,0,'c','ca','C',-9] >>>list.sort

		[-9,0,'C','c','ca']
	reverse() : To reverse index of all the values of list	reverse() >>>list.reverse() [4,1,4,6,"BR"]
To count values	count() :Returns number of occurrences of given value	count(value) >>>list.count(4) 2
To locate a value index	index(v) : Returns index of a given value (first occurrence)	>>>list.index('4') 1

To convert other data types to list	list() : To convert other data types to list	>>>list(tuple(3,4)) [3,4] >>>x= list(otherdatatype) >>>type(x) list

Built-in Python functions with list and Syntax	Uses
len(list)	Returns the length of list
max(list)	Returns the maximum value
min(list)	Returns the minimum value
cmp(list1,list2) only in version 2.7.x	Returns 1 if list1 > list2 0 if list1==list2 -1 if list1<list2
del(list)	Deletes list object and values

List and For

Loop for item in

list:

statements

Here item is a newly introduced variable but list must be defined

Tuples

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

Example program

```
First_tuple = ('abcd',147,2.43,'Tom',74.9)

Small_tuple = (111,'Tom')

Print(First_tuple)

Print(First_tuple[0])

Print(First_tuple[1:3])

Print(First_tuple[2:])

Print(Small_tuple*2)

Print(First_tuple + Small_tuple)
```

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')
```

The following code is invalid with tuple whereas this is possible with lists

```
First_list = ['abcd',147,2.43,'Tom',74.9]

First_tuple = ('abcd',147,2.43,'Tom',74.9)

Tuple[2] = 100 #invalid Syntax with tuple

List[2] = 100          #valid Syntax with list
```

To delete an entire tuple we can use the del statement. It is not possible to remove individual items from a tuple. Example del tuple

Built-in Tuple Functions

1) **len(list)** – Gives the total length of the tuple.

Example program

```
Tuple1 =('abcd',147,2.43,'Tom',74.9)
Print(len(Tuple1))
```

Output

4

2) **max(list)** – Returns item from the tuple with maximum value.

Example program

```
Tuple1 = (1200, 147, 2.43, 1.12) Tuple2 =
(213, 100, 289)
Print("Maximum value in: ", Tuple1 , "is", max(Tuple1))
Print("Maximum value in: ", Tuple2 , "is", max(Tuple2))
```

Output

Maximum value in: (1200, 147, 2.43, 1.12) is 1200 Maximum value in:

(213, 100, 289) is 289

3) **min(list)** – Return item from the tuple with minimum value.

Example program

```
Tuple1 = (1200, 147, 2.43, 1.12) Tuple2 =
(213, 100, 289)
Print("Manimum value in: ", Tuple1 , "is", min(Tuple1))
Print("Manimum value in: ", Tuple2 , "is", min(Tuple2))
```

Output

Manimum value in: (1200, 147, 2.43, 1.12) is 1.12 Manimum value in:

(213, 100, 289) is 100

4) **tuple(seq) –** Return a list into a tuple

Example program

```
list = ['abcd', 147, 2.43, 'Tom']
                print("Tuple:", tuple(list))
```

Chapter 7: Python data types – Dictionary and Set

Dictionary

Dictionary is an unordered collection of key-value pairs. It is generally used when we have a huge amount of data. We must know the key to retrieve the value. In python, dictionaries are defined within braces {} with each item being a pair in the form key:value. 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 enclosed by curly braces ({}) and values can be assigned and accessed using square braces ([])

Example program

```
Dict = {}

Dict['one'] = "This is one" Dict[2] =

"This is two"

Tinydict = {'name':'jhon','code':6758:'dept':'sales'}

print(Dict['one'])                    #prints value      for 'one' key

print(dict[2])                        #prints value      for 2 key

print(Tinydict)                       #prints complete dictionary

print(Tinydict.keys())                #prints    all    keys

Print(Tinydict.values())              #prints    all    values
```

Output

```
This is one
This is two
{'dept':'sales','code':'6758,'name':'jhon'}
{'dept', 'code', 'name'}
{'sales', '6758', 'jhon'}
```

We can update a dictionary by adding a new key-value pair or modifying an existing entry.

Example program

```
#demo of updating and adding new values to dictionary
```

```
Dict1 = {'Name':''tom','Age':'20,'Height':160,}
print(Dict1)
Dict1['age']=25                    #updating existing value in key-
value pair
print("dictionary after update:",dict1)
Dict1['weight']=60                 #adding new key-value pair
print("dictionary afte adding new key-value pair:",dict1)
```

Output

```
{'Age':20,'Name':'Tom','Height':160}
Dictionary after update:
{'age':20,'Age':20,'Name':'Tom','Height':160}
Dictionary after adding new kwy-value pair:
{'age':25,'Age':20.'Name':'tom','Weight':60,'Height':'160 '}
```

We can delete the entire dictionary elements or individual elements in a dictionary. We can use del statement to delete the dictionary completely. To remove entire elements of a dictionary, we can use the clear () method

Example program

```
#demo of deleting dictionary
Dict1 = { 'Age':20,'Name':'Tom','Height':160}
print(Dict1)
del Dict1['Age']                   #deleting key-value pair'Age':20
print("dictionary after deletion: ",Dict1)
Dict1.clear()                      #clearing entire dictionary
print(Dict1)
```

Output

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

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 Functions

1) **Len(dict) –** Gives the length of the dictionary.

Example program

```
#demo of len(dict)
Dict1 = {'Age':20,'Name':'Tom','Height':160} print(Dict1)
print("length of dictionary=",len(Dict1))
```

Output

```
{'Age':20,'Name':'Tom','Height':160}  length  of
dictionary=3
```

2) **Str(dict) –** Produces a printable string representation of the dictionary.

Example program

```
#demo of str(dict)
Dist1 = {'Age':20,'Name':'Tom','Height':160}
print (Dict1)
print("representation of dictionary=",str(Dictt1))
```

Output

```
{'Age':20,'Name':'Tom','Height':160}
representation of dictionary={'Age':20,'Name':'Tom','Height':160}
```

3) **type(variable)** – The method type() returns the type of the passes variable. If passed variable is dictionary then it would return a dictionary type. This function can be applied to any variable type like numbers, string, list, tuple etc

Example program

```
#demo of type(variable)
        Dict1 = {'Age':20,'Name':'Tom','Height':160}
print(dict1)
print("type(variable)=",type(Dict1))
S='abcde'
print("type(variable)=",type(S))
listr1 = [1,'a','23','Tom']
print("type(variable)=",type(list1))
```

Output

```
{'Age':20,'Name':'Tom','Height':160}
Type(variable)=<type 'dict'>
Type(variable)=<type 'str'>
Type(variable)=<type 'list'>
```

Built-in Dictionary Methods

1) **dict.clear()** – removes all elements of dictionary dict.

Example program

```
#demo of dict.clear()
        Dict1 = {'Age':20,'Name':'Tom','Height':160}
print(Dict1)
Dict1.clear()
print Dict1
```

Output

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

2) **dict.copy()** – Returns a copy of the dictionary dict().

Example program

```
#demo of dict.copy()
        Dict1 = {'Age':20,'Name':'Tom','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()** – Return a list of keys in dictionary dict.

Example program

```
#demo of dict.keys()

Dict1 = {'Age':20,'Name':'Tom','Height':160} print(dict1)

print("keys in dictionary:",dict1.keys())
```

Output

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

Keys in dictionary : ['Age','Name','Height']
```

4) **dict.values()** – This method returns list all values svailable in a dictionary.

Example program

```
#demo of dict.values()

Dict1 = {'Age':20,'Name':'Tom','Height':160}

print(Dict1)

print("values in dictionary:",Dict1.values())
```

Output

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

Values in dictionary:[20,'Tom',160]
```

5) **dict.items()** – returns a list of dictionary dict's(key,value) tuple pairs.

Example program

```
#demo of dict.items()
Dict1 = {'Age':20,'Name':'Tom','Height':160}

print(Dict1)

print("items in dictionary:",Dict1.items())
```

Output

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

Itmes 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 program

```
#demo of dict1.update()
Dict1 = {'Age':20,'Name':'Tom','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','Height':160,'weight':60}

7) **dict.has_key(key)** – Return True, if the key in the dictionary dict, else False is returned.

Example program

```
#demo of dict1.has_key(key)
Dict1 = {'Age':20,'Name':'Tom','Height':160}
print(dict1)
print("Dict1.has_key(key):",Dict1.has_key('Age'))
print("Dict1.has_key(key):",Dict1.has_key('phone'))
```

Output

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

Dict1.has_key(key) : True

Dict1.has_key(key) : False

8) **dict.get(key,default=None)** –Returns the value corresponding to the key specified and if key specified is not in the dictionary , it returns the default value

Example program

```
#demo of dict.get(key,default='Name')

Dict1 = {'Age':20,'Name':'Tom','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
```

9) **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 program

```
#demo of dict1.set(key,default='Name')

Dict1={'Age':20,'Name':'Tom','Height':160}
print(Dict1)
print("Dict1.setdefault('Age')

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

Output

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

Dict1.setdefault('Age') : 20

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

10) **dict.fromkeys(seq,[val])** – Creates a new dictionary from sequence seq and values from val.

Example program

```
#demo of dict.fromkeys)seq,[val])

List = ['Name', 'Age', 'Height']

Dict = Dict.fromkeys(list)

print ("new dictionary:",dict)
```

Output

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

Sets

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. 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.

Example program

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

print(s1)

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

print(s2)

s3 = {1, 2.4,'apple','tom',3)

print(s3)
```

Output

```
set([1,2,3])

set([1,2,3])

set([1,3,2.4,'apple','tom'])
```

Built-in Set Functions

1) len(set) - Returns the length or total number of items in a set.

Example program

```
set1 = {1,2,3}

print(len(set1))
```

Output

```
3
```

2) max(set) - Returns item form a set with maximum value.

Example program

```
set1 = {1,200,300,100}

print("the maximum value in",set1,"is",max(set1))
```

Output

```
the maximum value in {1,200,300,100} is 300
```

3) min(set) - Returns item form a set with minimum value.

Example program

```
set1 = {1,200,300,100}

print("the minimum value in :",set1,"is",min(set1))
```

Output

```
the minimum value in {1,200,300,100} is 1
```

4) sum(set) - Returns the sum of all items in the set.

Example program

```
set1 = {147,2.43}

print("the sum of elements in set",set1,"is",sum(set1)
```

Output

```
the sum of elements in set ([147,2.43]) is 149.43
```

5) sorted(set) -Returns a new sorted set. The set does not sort itself.

Example program

```
set1 = {2,3,6,4,5} set2 =

sorted(set1)

print("set before sorting:",set1) print("set

after sorting:",set2)
```

Output

```
set before sorting: ([2,3,6,4,5]) set after

sorting: ([2,3,4,5,6])
```

6) enumerate(set) -Returns an enumerate object. It contains the index and value of all the items of set pair.

Example program

```
set1 = {213,100,289,40,23,1,1000}
print("enumerate(set):",enumerate(set))
```

Output

```
enumerate(set): <eumerate object at 0x7f0a73573690>
```

7) any(set) - Returns true, if the set contains at leat one item, false otherwise.

Example program

```
set1 = set()
set2 = {1,2,3,4}
print("any(set):",any(set1))
print("any(set):",any(set2))
```

Output

```
any(set):False
any(set):True
```

8) all(set) - Returns True, if all the elements are true or the set is empty.

Example program

```
set1 = set()
set2 = {1,2,3,4}
print("any(set):",all(set1))
print("any(set):",all(set2))
```

Output

```
any(set):True
any(set):True
```

Built-in Set Methods

1) set.add(obj) - Adds an element obj to a set.

Example program

```
set1 = {1,2,3,4}
```

```
print("set before addition:",set1) set1.add(5)
print("set after addition:",set1)
```

Output

```
set before addition: set([1,2,3,4]) set after
addition: set([1,2,3,4,5])
```

2) set.remove(obj) - Removes an element obj from the set. Raises KeyError if the set is empty.

Example program

```
set1 = {1,2,3,4}
print("set before deletion:",set1)
set1.remove(4)
print("set after deletion:",set1)
```

Output

```
set before deletion: set([1,2,3,4]) set after
deletion: set([1,2,3])
```

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 program

```
set1 = {1,2,3,4}
print("set before discard:",set1)
set1.discard(4)
print("set after discard:",set1)
set1.discard(6)
print("set after discard:",set1)
```

Output

```
set before discard: set([1,2,3,4]) set after
discard: set([1,2,3])
```

set after discard: set([1,2,3])

4) set,pop()- Removes and returns an arbitrary set elements. Raises KeyError if the set is empty.

Example program

```
set1 = {1,2,3,4}

print("set before popping:",set1) set1.pop()

print("set after popping:",set1)
```

Output

set before popping: set([2,3,1,4]) set after

popping: set([2,3,6])

5) set1.union(set2) -Returns the union of two sets as a new set.

Example program

```
set1={3,8,2,6}

print("set1:",set1)

set2={4,2,1,9}

print("set2:",set2)

set3=set1.union(set2)

print("union:",set3)
```

Output

set1: set([8,2,3,6])

set2: set([9,2,4,1])

union: set([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 program

```
set1={3,8,2,6}

print("set1:",set1)
```

```
set2={4,2,1,9}
print("set2:",set2)
set1.update(set2) print("update
method:",set1)
```

Output

```
set1: set([8,2,3,6]) set2:
set([9,2,4,1])
update method: set([1,2,3,4,6,8,9])
```

7) set1.intersection(set2) - Returns the intersection of two sets as a new set.

Example program

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

Output

```
set1: set([8,2,3,6])
set2: set([9,2,4,1])
intersection: set([2])
```

8) set1.intersection_update() - Update the set with the intersection of itself and another. The result will be stored in set1.

Example program

```
set1={3,8,2,6}
print("set1:",set1)
set2={4,2,1,9}
print("set2:",set2)
```

```
set1.intersection_update(set2)
print("intersection_update:",set1)
```

Output

```
set1: set([8,2,3,6]) set2:
set([9,2,4,1])
intersection_update: set([8,2,3,6])
```

9) set1.differnce(set2) - Returns the difference of two or more sets into new set

Example program

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

Output

```
set1: set([8,2,3,6])
set2: set([9,2,4,1])
difference: set([3,6,8])
```

10) set1.difference_update(set2) - Remove all elements of another set set2 from set1 and result is stored in set1

Example program

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

Output

set1: set([8,2,3,6])

set2: set([9,2,4,1])

difference_update: set([8,3,6])

11) set1.symmetric_differnce(ste2) - Return the symmetric difference of two sets as a new set.

Example program

```
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: set([8,2,3,6]) set2:

set([9,2,4,1])

symmetric_difference: set([1,3,4,6,8,9])

12) set1.isdisjoint(set2) - Return true if two sets have a null intersection

Example program

```
set1={3,8,2,6}
print("set1:",set1)
set2={4,7,1,9}
print("set2:",set2)
print("result is:",set1.isdisjoint(set2))
```

Output

set1 = set([3,8,2,6]) set2 =

set([4,7,1,9])

result is: True.

13) set1.issubset(set2) - Return True if set1 is a subset of set2

Example program

```
set1={3,8}
set2={3,8,1}
print("result is:",set1.issubset(set2))
```

Output

result is: True.

14) set1.issuperset(set2) - Returns True, if set1 is a super set of set2

Example program

```
set1={3,8,1}
set2={3,8}
print("result is:",set1.issubset(set2))
```

Output

result is: True.