PYTHON BASICS

Python is most widely used **general purpose high level programming language** like Java, C, C++ etc. Python was developed by **Guido van Rossum** in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands. Python is derived from many other languages, including **ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell** and other scripting languages. Rossum was a fan of a comedy series from late seventies. Python is named after a TV Comedy Show called '**Monty Python's Flying Circus'** and not after Python-the snake**. Python 3.7.0** is the latest version of Python.

FEATURES OF PYTHON

- **General purpose programming language** - As a general-purpose programming language python is designed to be used for widest variety of application domains (a general-purpose language). ...It can be used for developing both desktop and web applications, complex scientific and numeric applications, data analysis and visualization. Conversely, a domain-specific programming language is one designed to be used within a specific application domain. For eg: COBOL (Business applications), FORTRAN(Complex mathematical computations).
- **High level language** - High-level language is any programming language that enables development of a program in a much more user-friendly

programming context and is generally independent of the computer's hardware architecture. Like JAVA, C ,C++ Python is a Highlevel programming language.

- **Interpreted** - Python runs on an interpreter system, meaning that code can be executed as soon as it is written. You do not need to compile your program before executing it.
- **Interactive** - We can actually sit at a Python prompt and interact with the interpreter directly to write your programs and It allows interactive testing and debugging of snippets of code.
- **Portable** - Python can run on a wide variety of hardware platforms like Windows, Linux, MAC OS and has the same interface on all platforms. You can move Python programs from one platform to another, and run it without any changes.
- **Multiple Programming Paradigms**- Python also supports several programming paradigms. It supports object oriented and Functional programming. Python can be treated in a procedural way, an object-oriented way or a functional way.
- **Easy to Learn** - Python has a simple syntax similar to the English language. This makes python easier to learn. Python has syntax that allows developers to write programs with fewer lines than some other programming languages. Most of other high level languages like JAVA, C, C++ has so many syntactical constructs like Punctuation marks, semicolon, braces etc to indicate the ending of a statement or to identify block of

code. But in Python, It has fewer syntactical constructions than other languages.

For eg:

In **JAVA**

Suppose we need to assign a name **"BOB"** to a variable **name** .In JAVA we have to first specify the type of the variable as **String**. After that we assign value **BOB** to variable **name** and put a semicolon at the end to indicate ending of a line.

String name="BOB";

In order to print it out we use **System.Out.Println("name");** and put a semi colon at the end.

But

In **Python**

We write **name="BOB"**

In order to print it out we use simply **print(name**)

We do not need to specify the type and put semicolon to indicate the ending of a line. This syntax makes Python code more easier to read and Learn.

- **Dynamic Typed Language** – As specified above, in python we don't have to specify the type when declaring a variable. It skip the headache of type casting and declaring types when declaring a variable.

  In JAVA,

  int x=1;

  x=(int)x/2;

In order to store integer values to a variable **x** first we have to declare its type as **int**. it means that x always store integer values .In the first line we assign value 1 to integer variable x.

If we take **x=x/2**; the value of x becomes ½. x can never equal 0.5. So first we have to cast the result into type integer using (int) function. Now the result becomes 0.

In python,

x=1

x=x/2

In this case, Python itself take care of type management we don't need to worry about it.

If we write x=1 at this point type of x is int because we assign integer value to x. if we write x=x/2 now x equals 0.5 and the type of x at this point is float. We don't need to explicitly type cast the result into float. According to the type of values we store to a variable its type is decided by the interpreter at runtime. We don't need to worry about it.

- **Databases** - Python provides interfaces to all major commercial databases like POSTGRES, MY SQL, SQLITE.
- **GUI Programming**- Python supports GUI applications. Python provides Tk GUI library to develop user interface in python based application.

Setting up of Environment

In order to use python first it must be installed on our computer, Follow these steps

1).Go to the Python website www.python.org and click the Download menu choice.

2). Next , Download the Python 3.7.0 installer.

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

First Python Program

Let us execute the programs in different modes of programming.

- **Interactive Mode Programming**.

  Python provides Interactive Shell to execute code immediately and produce output instantly. To test a short amount of code in python sometimes it is quickest and easiest not to write the code in a file. This is made possible because Python can be run as a command line itself.

  Type the following text at the Python prompt and press Enter –

  >>> **print ("Hello, Python!")**

  This produces the following result –

  >>>**Hello, Python!**

- **Script Mode Programming**

  Using **Script Mode**, we can write our Python code in a separate file of any editor in our Operating System. Let us write a simple Python program in a script. Python files have the extension .**py**. Type the following source code in **a test.py** file –

print ("Hello, Python!")

Now open Command prompt and execute it by :

>>> **python test.py**

This produces the following result –

>>>**Hello, Python!**

- **USING IDLE**

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

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

>>>**print("Hello, World!")**

Or

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

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

print("Hello, World!")

Go to Run > Run module (Shortcut: F5) and we can see the output.

IDENTIFIERS

- Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.

Rules for writing identifiers Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like myClass, var_1 and print_this_to_screen, all are valid example.

- An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
- Keywords cannot be used as identifiers.

```
>>> global = 1

File "<interactive input>", line 1
  global = 1
         ^
SyntaxError: invalid syntax
```

- We cannot use special symbols like !, @, #, $, % etc. in our identifier.

```
>>> a@ = 0
  File "<interactive input>", line 1
    a@ = 0
     ^
SyntaxError: invalid syntax
```

- Identifier can be of any length.
- Python is a case-sensitive language. This means, Variable and variable are not the same. Always name identifiers that make sense.

Python Keywords

Keywords are the reserved words in Python. We cannot use a keyword as variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language. In Python, keywords are case sensitive. All the keywords except True, False and None are in lowercase and they must be written as it is. The list of all the keywords are given below.

True	False	None	and	as
Asset	Def	Class	continue	break
Else	Finally	Elif	del	except
Global	For	If	from	import
Raise	Try	or	return	pass
nonlocal	In	not	is	lambda

Lines and Indentation

Python does not use braces ({}) to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced. The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example –

```
if (True):
    print ("True")
else:
    print ("False")
```

Multi-Line Statements

Statements in Python typically end with a new line. Python, however, allows the use of the **line continuation character (\)** to denote that the line should continue. For example −

```
total = item_one + \
    item_two + \
    item_three
```

The statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example −

```
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Quotation in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string. The triple quotes are used to span the string across multiple lines. For example, all the following are legal −

word = 'word'

sentence = "This is a sentence."

paragraph = """This is a paragraph. It is

made up of multiple lines and sentences."""

Comments in Python

Python supports two types of comments:

1) **Single lined comment**:

In case user wants to specify a single line comment, then comment must start with **#**

eg: **# This is single line comment**.

2) **Multi lined Comment**:

Multi lined comment can be given inside triple quotes.

eg: **''' This**

Is

Multipline comment'''

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on a single line given that no statement starts a new code block

For eg:

If(a>b);print(“a is greater than b”)

Variables

Variables are nothing but reserved memory locations to store values. It means that when you create a variable, you reserve some space in the memory. 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 the variables, you can store integers, decimals or characters in these variables.

Assigning Values to 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.

```
c = 100                 # An integer assignment
m  = 1000.0             # A floating point
name   = "John"         # A string
print (c)
print (m)
print (name)
```

Output: 100

1000.0

John

Multiple Assignment

Python allows us to assign 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 the three variables are assigned to the same memory location. We can also assign multiple objects to multiple variables.

For example −

```
a, b, c = 1, 2, "john"
```

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

Standard Data Types

The data stored in memory can be of many types.

Python provides following data types −

- **Numbers**
- **String**
- **List**

- **Tuple**
- **Dictionary**
- **Set**

Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example −

```
var1 = 1
```

We can also delete the reference to a number object by using the **del** statement. The syntax of the **del** statement is −

```
del var1[,var2[,var3[....,varN]]]]
```

You can delete a single object or multiple objects by using the **del** statement.

For example −

```
del var
```

Python supports three different numerical types −

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

A complex number consists of an ordered pair of real floating-point numbers denoted by **x + yj,** where x and y are real numbers and j is the imaginary unit.

Python Strings

Strings in Python are identified as **a contiguous set of characters** represented in the quotation marks. Python allows either pair of single or double quotes.

Subsets of strings can be taken using the **slice operator** ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 to the end.

The plus (+) sign is the string **concatenation operator** and the asterisk (*) is the **repetition operator.**

```
str = 'Hello World!'

print (str)          # Prints complete string

print (str[0])       # Prints first character of the string

print (str[2:5])     # Prints characters starting from 3rd to 5th

print (str[2:])      # Prints string starting from 3rd character

print (str * 2)      # Prints string two times

print (str + "TEST") # Prints concatenated string

print (str [ -1])      # Prints Last character of the string
```

OUTPUT:

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
!
```

Python Lists

Lists are the most versatile of Python's compound data types. **A list contains items separated by commas and enclosed within square brackets ([]).** To some extent, lists are similar to **arrays** in C. **One of the differences between them is that all the items belonging to a list can be of different data type**.

The values stored in a list can be accessed using the **slice operator ([ ] and [:])** with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list **concatenation operator**, and the asterisk (*) is the **repetition operator**.

For example −

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tinylist = [123, 'john']

print (list)        # Prints complete list
```

```
print (list[0])      # Prints first element of the list

print (list[1:3])    # Prints elements starting from 2nd till 3rd

print (list[2:])     # Prints elements starting from 3rd element

print (tinylist * 2)  # Prints list two times

print (list + tinylist) # Prints concatenated lists
```

OUTPUT:

```
['abcd', 786, 2.23, 'john', 70.200000000000003]
abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, **tuples are enclosed within parenthesis.**

The main difference between lists and tuples are – Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as read-only lists.

For example −

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )

tinytuple = (123, 'john')

print (tuple)          # Prints complete tuple

print (tuple[0])       # Prints first element of the tuple

print (tuple[1:3])     # Prints elements starting from 2nd till 3rd

print (tuple[2:])      # Prints elements starting from 3rd element

print (tinytuple * 2)  # Prints tuple two times

print (tuple + tinytuple) # Prints concatenated tuple
```

OUTPUT:

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists −

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2  )
```

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2  ]

tuple[2] = 1000    # Invalid syntax with tuple

list[2] = 1000     # Valid syntax with l
```

Python Dictionary

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

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

For example −

```
dict = {}

dict['one'] = "This is one"

dict[2]     = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print (dict['one'])       # Prints value for 'one' key

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

print (tinydict)         # Prints complete dictionary
```

```
print (tinydict.keys())          # Prints all the keys

print (tinydict.values())        # Prints all the values
```

OUTPUT–

```
This is one
This is two
{'name': 'john', 'dept': 'sales', 'code': 6734}
dict_keys(['name', 'dept', 'code'])
dict_values(['john', 'sales', 6734])
```

Dictionaries have no concept of order among the elements. It is incorrect to say that the elements are "out of order"; **they are simply unordered**.

Set

A set is an **unordered collection of items**. Every element is **unique (no duplicates) and must be immutable (which cannot be changed)**. However, **the set itself is mutable**. We can add or remove items from it. Sets can be used to perform mathematical set operations like **union, intersection, symmetric difference** etc.

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

For eg:

my_set = {1, 2, 3}

print(my_set) **Output:{1,2,3}**

my_set = {1.0, "Hello", (1, 2, 3)}

print(my_set) **Output:{1.0,”Hello”,(1,2,3)}**

*** Set do not have duplicates**

my_set = {1,2,3,4,3,2}

print(my_set) **Output:{1,2,2,4}**

***Set cannot have mutable items**

my_set = {1, 2

, [3, 4]} **Output: Type Error: unhashable type: 'list'**

***Using set() Function**

my_set = set([1,2,3,2])

print(my_set) Output: {1, 2, 3}

*** Creating an empty set**

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

For eg:

a = set()

*We cannot access or change an element of set using indexing or slicing. Set does not support it. We can add single element using the **add()** method and multiple elements using the **update()** method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

For eg:

my_set = {1,3}

print(my_set) **Output:{1,3}**

my_set.add(2)

print(my_set) **Output:{1,2,3}**

my_set.update([2,3,4])

print(my_set) **Output:{1,2,3,4}**

my_set.update(((4,5), 1,6,8}))

print(my_set) **Output: {1, 2, 3, 4, (4,5), 6, 8}**

***A particular item can be removed from set using methods, discard() and remove().**

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

The following example will illustrate this.

```
# initialize my_set

my_set = {1, 3, 4, 5, 6}

print(my_set)                                   Output: {1, 3, 4, 5, 6}

#discard an element

my_set.discard(4)

print(my_set)                                   Output: {1, 3, 5, 6}

# remove an element

my_set.remove(6)

print(my_set)                                   Output: {1, 3, 5}

# discard an element

# not present in my_set

my_set.discard(2)
```

print(my_set) **Output: {1, 3, 5}**

 # remove an element

 # not present in my_set

 # we will get an error.

my_set.remove(2) **Output: Key Error: 2**

Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type-names as a function. There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

r.No.	Function & Description
1	**int(x [,base])** Converts x to an integer. The base specifies the base if x is a string.
2	**float(x)** Converts x to a floating-point number.
3	**complex(real [,imag])** Creates a complex number.

4	**str(x)** Converts object x to a string representation.
5	**repr(x)** Converts object x to an expression string.
6	**eval(str)** Evaluates a string and returns an object.
7	**tuple(s)** Converts s to a tuple.
8	**list(s)** Converts s to a list.
9	**set(s)** Converts s to a set.
10	**dict(d)** Creates a dictionary. d must be a sequence of (key,value) tuples.
11	**frozenset(s)** Converts s to a frozen set.

12	**chr(x)** Converts an integer to a character.
13	**unichr(x)** Conv4erts an integer to a Unicode character.
14	**ord(x)** Converts a single character to its integer value.
15	**hex(x)** Converts an integer to a hexadecimal string.
16	**oct(x)** Converts an integer to an octal string.

OPERATORS

Operators are the constructs, which can manipulate the value of operands.

Types of Operator

Python language supports the following types of operators −

- **Arithmetic Operators**
- **Comparison (Relational) Operators**

- **Assignment Operators**
- **Logical Operators**
- **Bitwise Operators**
- **Membership Operators**
- **Identity Operators**

Python Arithmetic Operators

Assume variable **a** holds the value 10 and variable **b** holds the value 21, then −

• Operator	• Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 31
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -11
* Multiplication	Multiplies values on either side of the operator	a * b = 210
/ Division	Divides left hand operand by right hand operand	b / a = 2.1
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 1
** Exponent	Performs exponential (power) calculation on	a**b =10 to

	operators	the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):	9//2 = 4 and 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0

Python Comparison Operators

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

Assume variable **a** holds the value 10 and variable **b** holds the value 20, then −

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a!= b) is true.
>	If the value of left operand is greater than the value	(a > b) is not

	of right operand, then condition becomes true.	true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Python Assignment Operators

Assume variable **a** holds the value 10 and variable **b** holds the value 20, then −

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c +

		a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Python Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows −

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

Python's built-in function **bin()** can be used to obtain binary representation of an integer number.

The following Bitwise operators are supported by Python language −

Operator	Description	Example
& Binary AND	Operator copies a bit, to the result, if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit, if it exists in either operand.	(a | b) = 61 (means 0011 1101)

^ Binary XOR	It copies the bit, if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operand's value is moved left by the number of bits specified by the right operand.	a << = 240 (means 1111 0000)
>> Binary Right Shift	The left operand's value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

Python Logical Operators

The following logical operators are supported by Python language. Assume variable **a** holds True and variable **b** holds False then −

Operator	Description	Example
and Logical	If both the operands are true then condition	(a and

AND	becomes true.	b) is False.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is True.
not Logical NOT	Used to reverse the logical state of its operand.	Not (a and b) is True.

Python Membership Operators

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below −

Operator	Description	Example
In	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in	x not in y, here

	the specified sequence and false otherwise.	not in results in a 1 if x is not a member of sequence y.

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators as explained below −

Operator	Description	Example
Is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

Python Operators Precedence

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

Sr.No.	Operator & Description
1	****** Exponentiation (raise to the power)
2	**~ + -** Complement, unary plus and minus (method names for the last two are +@ and -@)
3	*** / % //** Multiply, divide, modulo and floor division
4	**+ -** Addition and subtraction
5	**>> <<** Right and left bitwise shift
6	**&** Bitwise 'AND'
7	**^ |** Bitwise exclusive `OR' and regular `OR'
8	**<= < a> >=** Comparison operators
9	**<> == !=**

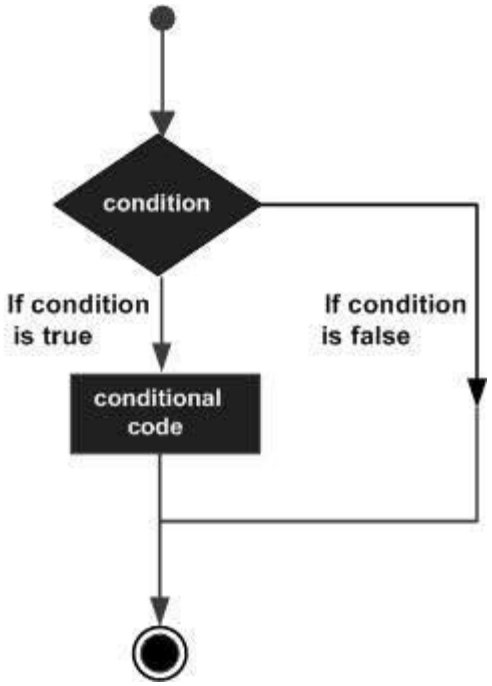
	Equality operators
10	**= %= /= //= -= += *= **=** Assignment operators
11	**is is not** Identity operators
12	**in not in** Membership operators
13	**not or and** Logical operators

DECISION MAKING STATEMENTS

Decision making is about deciding the order of execution of statements based on certain conditions. **Decision making** structures require that the

programmer specifies one or more conditions to be evaluated or tested by the program, along with a **statement** or **statements** to be executed if the condition is determined to be true, and optionally, other **statements** to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages −



Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.

Python programming language provides following types of decision making statements.

1	**if statements** An **if statement** consists of a boolean expression followed by one or more statements.

2	**if...else statements** An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.
3	**if ..elif..else statements** You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
4	**nested if statements** You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

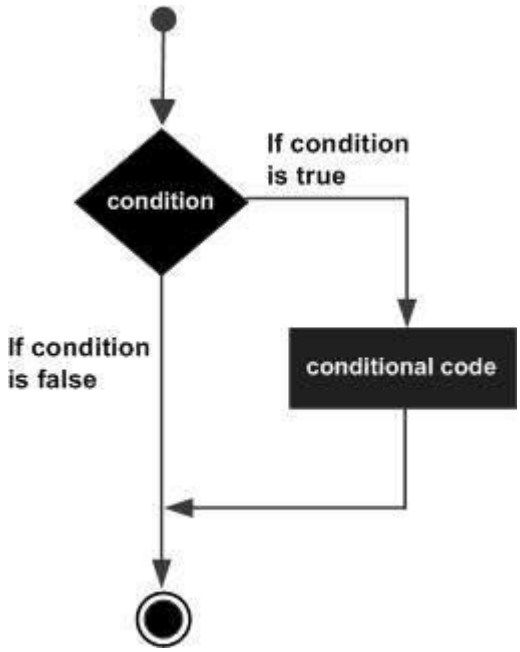
1. IF statement

Syntax

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

If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.

Flow Diagram



For eg:

```
a=int(input("Enter the First Number"))

b=int(input("Enter the Second Number"))

if(a>b):

        print("First Number is Biggest")

print("Hello World")
```

Output: Suppose a=10

b=20

Hello World

2).IF ELSE STATEMENT

An **else** statement can be combined with an **if** statement. An **else** statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

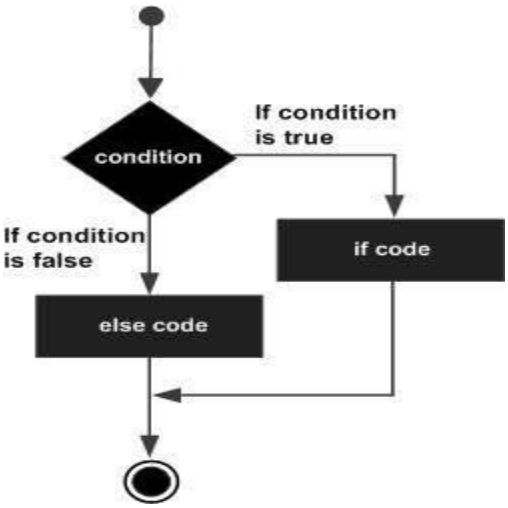
The else statement is an optional statement and there could be at most only one **else** statement following **if**.

Syntax

The syntax of the if...else statement is –

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

Flow Diagram



For eg:

a=int(input(“Enter the First Number”))

b=int(input("Enter the Second Number"))

if(a>b):

print("First Number is Biggest")

else:

print("Second Number is Biggest")

print("Hello World")

Output: Suppose a=10 b=20

Second Number is Biggest

Hello World

3.IF-ELIF-ELSE STATEMENT

The **elif** statement allows you to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE.

Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at most one statement, there can be an arbitrary number of **elif** statements following an **if**.

Syntax

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
```

```
    statement(s)
else:
    statement(s)
```

Core Python does not provide switch or case statements as in other languages, but we can use if..elif...statements to simulate switch case as follows –

For eg:

```
a=int(input("Enter The First Number"))
b=int(input("Enter The Second Number"))
op=input("Enter the operator(+,-,*,/)")
if(op=='+'):
        print(a+b)
elif(op=='-'):
        print(a-b)
elif(op=='*'):
        print(a*b)
elif(op=='/'):
        print(a/b)
else:
        print("No Operation")
```

OUTPUT: Suppose a=10 b=7 op='-'

FLOW DIAGRAM

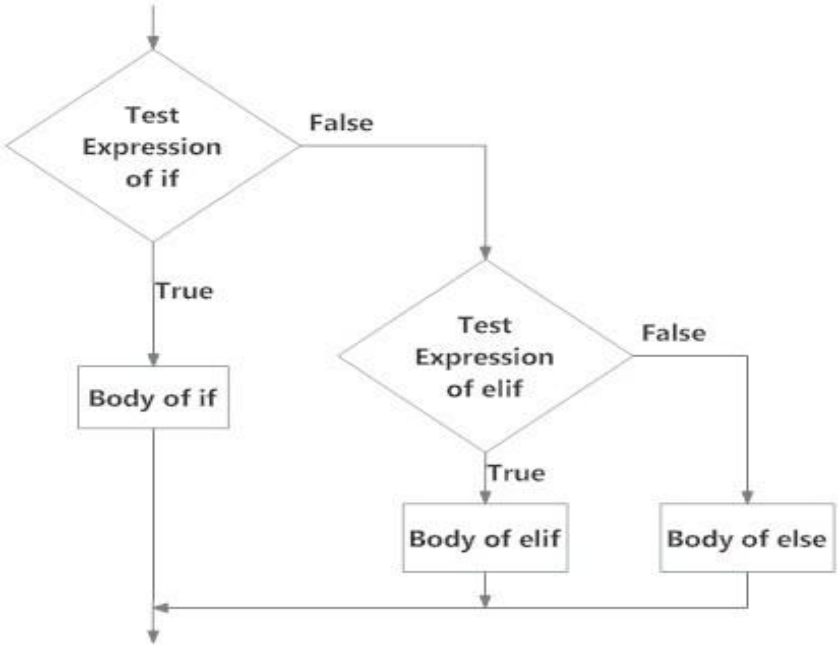


Fig: Operation of if...elif...else statement

4).NESTED IF CONSTRUCT

There may be a situation when you want to check for another condition after a condition resolves to true. In such a situation, you can use the nested **if**construct.

In a nested **if** construct, you can have an **if...elif...else** construct inside another **if...elif...else** construct.

Syntax

The syntax of the nested if...elif...else construct may be –

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
```

```
    statement(s)
  else:
    statement(s)
elif expression4:
    statement(s)
else:
  statement(s)
```

For eg:

```
num=int(input("Enter the number"))
if (num%2==0):
        if (num%3==0):
                print ("Divisible by 3 and 2")
        else:
                print ("divisible by 2 not divisible by 3")
else:
        if (num%3==0):
                print ("divisible by 3 not divisible by 2")
        else:
                print ("not Divisible by 2 not divisible by 3")
```

OUTPUT:

```
Enter the number       8
divisible by 2 not divisible by 3
Enter the number       15
divisible by 3 not divisible by 2
Enter the number       12
 Divisible by 3 and 2
 Enter the number      5
 not Divisible by 2 not divisible by 3
```

LOOPING STATEMENTS

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

A loop statement allows us to execute a statement or group of statements multiple times while a given **condition** is true. It tests the **condition** before executing the **loop body**.

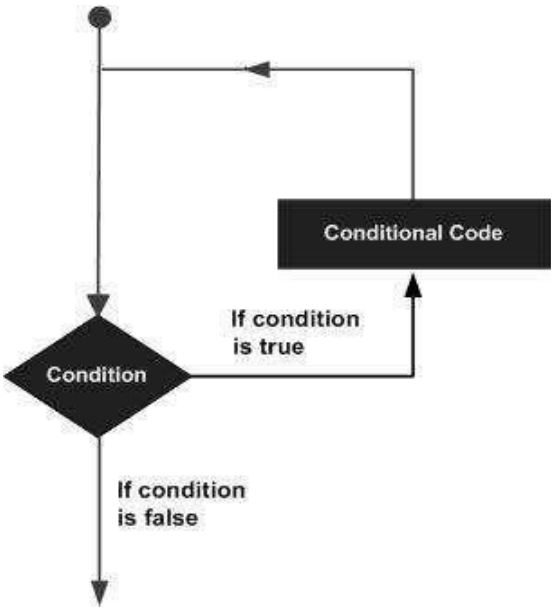
Python programming language provides the following types of loops to handle looping requirements.

while loop :-Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

for loop :-Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

nested loops:- we can use one or more loop inside any another while, or for loop.

The following diagram illustrates a loop statement



WHILE LOOP

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

Syntax

The syntax of a **while** loop in Python programming language is

while expression:

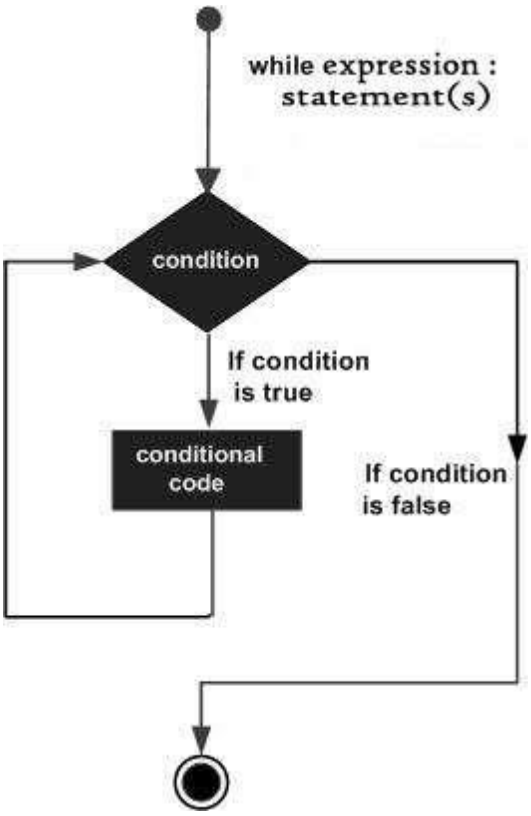
    statement(s)

Here, **statement(s)** may be a single statement or a block of statements with uniform indent. The **condition** may be any expression, and true is any non-zero value. **The loop** iterates while the condition is **true.**

When the condition becomes **false**, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Flow Diagram



For eg:

```
count = 1
sum=0
while (count <=10):
        sum=sum+count

        count = count + 1
print(sum)
```

OUTPUT:55

The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. You must be cautious when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

Using else Statement with Loops

Python supports having an **else** statement associated with a loop statement.

If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.

If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise the else statement gets executed.

For eg:

```
count = 0
while (count < 5):
        print (count, " is less than 5")
        count = count + 1
else:
        print (count, " is not less than 5")
```

OUTPUT:

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
```

5 is not less than 5

For Loop Statements

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

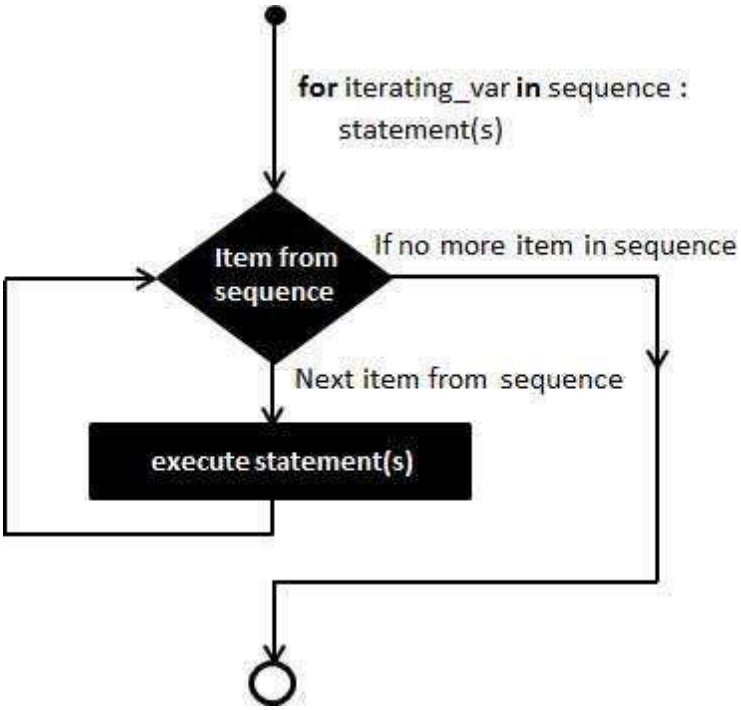
Syntax

for iterating_var in sequence:

    statements(s)

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating_var*, and the statement(s) block is executed until the entire sequence is exhausted

FLOW DIAGRAM



The range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided. This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

The built-in function range() is the right function to iterate over a sequence of numbers. It generates an iterator of arithmetic progressions.

>>> range(5)

range(0, 5)

>>> list(range(5))

[0, 1, 2, 3, 4]

range() generates an iterator to progress integers starting with 0 upto n-1. To obtain a list object of the sequence, it is typecasted to list(). Now this list can be iterated using the for statement.

>>> **for var in list(range(5)):**

    print (var)

This will produce the following **output.**

0

1

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

For eg:
for letter in 'Python':                  # traversal of a string sequence
        print ('Current Letter :', letter)
print()
fruits = ['banana', 'apple', 'mango']
for fruit in fruits:                     # traversal of List sequence
        print ('Current fruit :', fruit)
print ("Good bye!")

Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n

Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself. Following is a simple example-

```
fruits = ['banana', 'apple', 'mango']

for index in range(len(fruits)):

print ('Current fruit :', fruits[index])

print ("Good bye!")
```

OUTPUT

```
Current fruit : banana

Current fruit : apple

Current fruit : mango

Good bye!
```

Here, we took the assistance of the len() built-in function, which provides the total number of elements in the tuple as well as the range() built-in function to give us the actual sequence to iterate over.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements.

1. **break statement**

Terminates the loop statement and transfers execution to the statement immediately following the loop.

2. **continue statement**

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

3. **pass statement**

The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

Break

It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

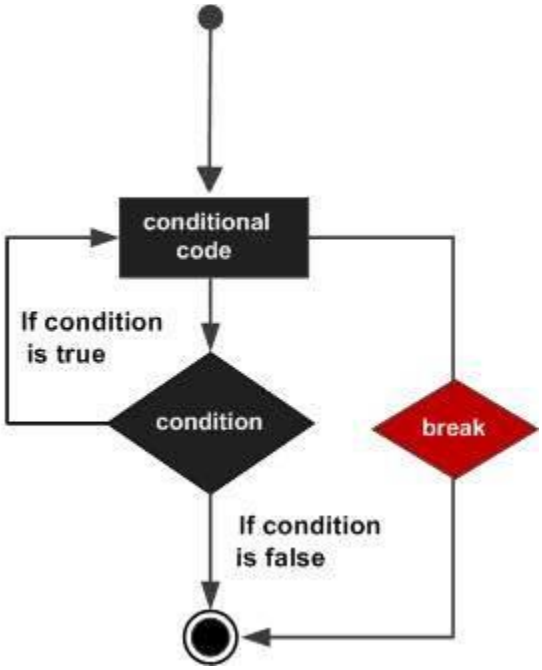
If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a **break** statement in Python is as follows –

```
break
```

Flow Diagram



For eg

```
for letter in 'Python':                                    # First Example
        if (letter == 'h')
                break
        print ('Current Letter :', letter)


var = 10                                                   # Second Example
while var > 0:
        print ('Current variable value :', var)
```

```
            var = var -1
            if var == 5:
                        break

print "Good bye!"
```

When the above code is executed, it produces the following output –

```
Current Letter : P
Current Letter : y
Current Letter : t

Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

CONTINUE

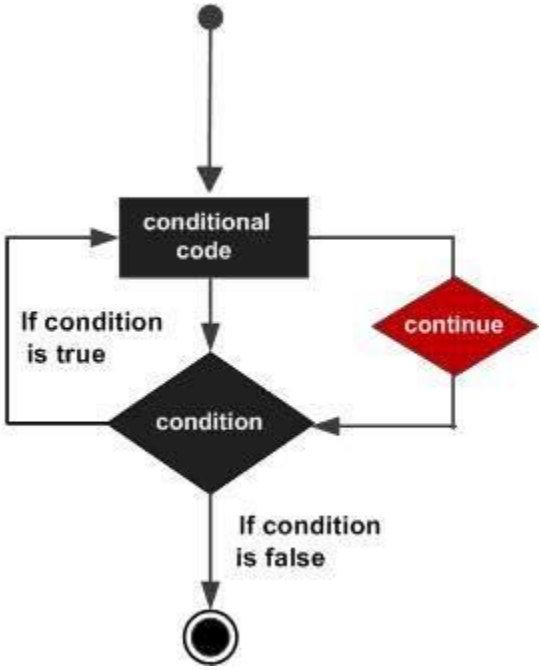
It returns the control to the beginning of a loop. The **continue** statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The **continue** statement can be used in both while and for loops.

Syntax

```
continue
```

Flow Diagram



For eg

```
for letter in 'Python':                    # First Example
        if (letter == 'h'):
                continue
        print ('Current Letter :', letter)


var = 10                                   # Second Example
while (var > 0):
        var = var -1
        if (var == 5):
                continue
print( 'Current variable value :', var)
```

```
print "Good bye!"
```

When the above code is executed, it produces the following output–

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n


Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Good bye!
```

Pass

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored.

However, nothing happens when pass is executed. It results into no operation (NOP).

Syntax

```
pass
```

We generally use it as a placeholder.

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

For eg

sequence = {'p', 'a', 's', 's'} **# pass is just a placeholder for**

for val in sequence: **# functionality to be added later**

pass

We can do the same thing in an empty function or class as well.

```
def function(args):
    pass
```

```
class example:
    pass
```

NESTED LOOPS

Python programming language allows to use one loop inside another loop.

Syntax

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

The syntax for a **nested while loop** statement in Python programming language is as follows −

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

Python Nested For Loop Example1

```
for i in range(1,6):
  for j in range (1,i+1):
      print(i)
   print()
```

Output:

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

For each value of Outer loop the whole inner loop is executed.

For each value of inner loop the Body is executed each time.

Python Nested Loop Example 2

```
for i in range (1,6):
    for j in range (5,i-1,-1):
        print ("*")
  print ()
```

Output:

```
* * * * *
* * * *
* * *
* *
*
```

Python Nested while Loop Example3

Prime Numbers from 2 to 100

```
i=2
while(i<=100):
    j=2
    while(j<i):
        if(i%j==0):
            break
        else:
            j=j+1
    else:
        print(i)
    i=i+1
```

OUTPUT

```
2
3
5
7
```

11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97

Python Nested Loop Example4

Find the elements (in a list) containing letter 'n'

```
n= ["banana","orange","grapes","apple"]
for i in n:
    for j in i:
        if(j=='n'):
            print(i)
            break
```

OUTPUT

banana

orange

FUNCTIONS

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. Furthermore, it avoids repetition and makes code reusable.

Syntax

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

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

There are three types of functions in Python:

- **Built-in functions**, such as help() to ask for help, min() to get the minimum value, print()to print an object to the terminal.
- **User-Defined Functions** (UDFs), which are functions that users create to help them out
- **Anonymous functions**, which are also called **lambda functions** **because** they are **not declared with the standard def keyword**.

*Example of a function

```
def greet(name):

    "This function greets to the person passed in as parameter "

    print("Hello, " + name + ". Good morning!")
```

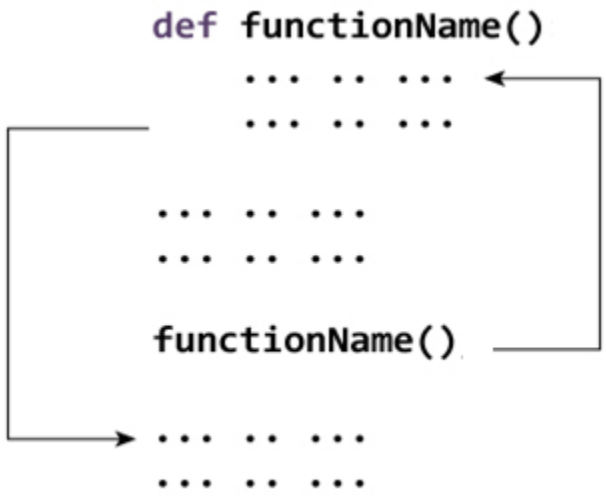
How to call a function in python

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

```
>>> greet('Paul')

Hello, Paul. Good morning!
```

How Function works in Python



Function Argument and Parameter

There can be two types of data passed in the function.

1) The First type of data is the data passed in the function call. This data is called **arguments**

2) The second type of data is the data received in the function definition. This data is called **parameters**

Arguments can be literals, variables and expressions. Parameters must be variable to hold incoming values.

Alternatively, arguments can be called as **actual parameters or actual arguments** and parameters can be called as **formal parameters or formal arguments**.

Function Arguments

You can call a function by using the following types of formal arguments −

Required arguments

Keyword arguments

Default arguments

Variable-length arguments

1 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.

To call the function printme(), you definitely need to pass one argument, otherwise it gives a syntax error as follows −

```
# Function definition is here

def printme( str ):

   "This prints a passed string into this function"

   print (str)

   return;

# Now you can call printme function

printme()
```

When the above code is executed, it produces the following result −

```
Traceback (most recent call last):

   File "test.py", line 11, in <module>

      printme();

TypeError: printme() takes exactly 1 argument (0 given)
```

2 Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. We can also make keyword calls to the printme() function in the following ways –

Ex1:

```
# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print (str)
    return;
# Now we can call printme function
printme( str = "My string")
```

OUTPUT

```
My string
```

Ex2:

```
# Function definition is here
def printinfo( name, age ):
```

```
    "This prints a passed info into this function"
    print ("Name: ", name)
    print( "Age ", age)
    return;
# Now we can call printinfo function
printinfo( age=50, name="miki" )
```

OUTPUT

```
Name:  miki
Age  50
```

3 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 gives an idea on default arguments, it prints default age if it is not passed –

```
# Function definition is here
def printinfo( name, age = 35 ):
    "This prints a passed info into this function"
    print ("Name: ", name)
```

```
    print ("Age ", age)

    return

# Now we can call printinfo function

printinfo( age=50, name="miki" )

printinfo( name="miki" )
```

OUTPUT

```
Name:  miki

Age  50

Name:  miki

Age  35
```

4 Variable-length arguments

You may need to process a function for more arguments than you 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.

Syntax

```
def functionname([formal_args,] *var_args_tuple ):
```

```
    "function_docstring"

    function_suite

    return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all non keyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

Example1

```
# Function definition is here

def printinfo( arg1, *vartuple ):

    "This prints a variable passed arguments"

    print ("Output is: ")

    print (arg1)

    for var in vartuple:

        print var

    return

# Now we can call printinfo function

printinfo( 10 )
```

```
printinfo( 70, 60, 50 )
```

OUTPUT

```
10
70
60
50
```

Example2

```
def varl(*a):
        sum=0
        for i in a:
                sum=sum+i
        print(sum)
        return
varl(10,8)
varl(6,5,4,3,2)
varl(7)
```

OUTPUT

```
18
20
```

7

The return Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows −

```
# Function definition is here

def sum( arg1, arg2 ):

        # Add both the parameters and return them."

        total = arg1 + arg2

        print ("Inside the function : ", total)

        return total;

# Now we can call sum function

total = sum( 10, 20 );

print ("Outside the function : ", total )
```

OUTPUT

Inside the function : 30

Outside the function : 30

Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized.

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python –

Global variables

Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions.

Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

Lifetime of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes. They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

```
def my_func():
        x = 10          #Here x is local variable
        print("Value inside function:",x)
x = 20                  #global variable
my_func()
print("Value outside function:", x)
```

OUTPUT

```
Value inside function: 10
```

```
Value outside function: 20
```

Here, we can see that the value of x is 20 initially. Even though the function my_func()changed the value of x to 10, it did not effect the value outside the function.

This is because the variable x inside the function is different (local to the function) from the one outside. Although they have same names, they are two different variables with different scope.

On the other hand, variables outside of the function are visible from inside. They have a global scope.

We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword **global.**

In Python, **global keyword** allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.

Rules of global Keyword

The **basic rules for global keyword in Python are:**

- **When we create a variable inside a function, it's local by default.**
- **When we define a variable outside of a function, it's global by default. You don't have to use global keyword.**
- **We use global keyword to read and write a global variable inside a function.**
- **Use of global keyword outside a function has no effect**

```
def add():
    global c
    c = c + 2        # increment by 2
    print("Inside add():", c)
c = 0                # global variable
add()
print(“Outside:”,c)
```

OUTPUT

```
Inside add(): 2

Outside: 2
```

Global in Nested Functions

Example

```
def foo():
        x = 20
        def bar():
            global x
            x = 25

        print("Before calling bar: ", x)
```

```
        print("Calling bar now")
        bar()
        print("After calling bar: ", x)
foo()
print("x in main : ", x)
```

Output

```
Before calling bar: 20
Calling bar now
After calling bar: 20
x in main : 25
```

In the above program, we declare global variable inside the nested function bar(). Inside foo() function, x has no effect of global keyword.

Before and after calling bar(), the variable x takes the value of local variable i.e x = 20. Outside of the foo() function, the variable x will take value defined in the bar() function i.e x = 25. This is because we have used global keyword in x to create global variable inside the bar() function (local scope).

If we make any changes inside the bar() function, the changes appears outside the local scope, i.e. foo().

The Anonymous Functions

These functions are called anonymous because they are not declared in the standard manner by using the def keyword. You can use the lambda keyword to create small anonymous functions.

Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions. An anonymous function cannot be a direct call to print because lambda requires an expression

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

Syntax

The syntax of lambda functions contains only a single statement, which is as follows –

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

Example1

```
# Function definition is here
double = lambda x: x * 2
# Now you can call double as a function
```

```
print(double(5))
```

In the above program, `lambda x: x * 2` is the lambda function. Here `x` is the argument and `x * 2` is the expression that gets evaluated and returned.

This function has no name. It returns a function object which is assigned to the identifier `double`. We can now call it as a normal function. The statement

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

is nearly the same as

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

OUTPUT

5

Example2

```
# Function definition is here
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print ("Value of total : ", sum( 10, 20 ))

print ("Value of total : ", sum( 20, 20 ))
```

OUTPUT

Value of total : 30

Value of total : 40

Use of Lambda Function in python

We use lambda functions 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 `map()` , `filter()`, reduce() etc

map Function

The `map()` function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Syntax

map (function, sequence)

Here sequence means list, tuple or strings. If we want to apply some function on each elements in a sequence then we can use map() function.

Example1

Find the square of each elements in a list

* map function without Using lambda function

a=[1,2,3,4]

first we need to define function

```
def square(x):
    return x:x*x
```

Next we will use map function

map(square,a)

In python3 it will return an iterator object and in python2 it will return as a list. In python 3 if we want to get the output as list we should use

list(map(square,a))

***If we want to get the output as tuple we need to use tuple(map(square,a))**

OUTPUT

[1,4,9,16]

In python 2 we need not mention list or tuple function. Directly we can write map(square,a). In python2 it will return output as list.

*map function using lambda function

map(lambda x:x*x,a)

To get the output in list form we should use

list(map(lambda x:x*x,a)

We can use more than one list or tuple in the map function. Suppose we want to add the elements of two lists we can use map function for that. We already have one list a=[1,2,3,4]

b=[1,1,1,1]

while adding two lists or tuples we should remember both the lists or tuples should have same length and same types of elements.

map (lambda x,y:x+y,a,b)

convert output into tuple or list

so list(map(lambda x,y:x+y,a,b))

we can add list and tuple in the same way we will get the output

a=[1,2,3,4]

b=(1,1,1,1)

list(map(lambda x,y:x+y,a,b))

Filter Function

The filter() function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

Syntax

filter (function ,iterables)

This filter function will return the elements from this iterables for which the function will return true.

Here is an example use of filter() function to filter out only even numbers from 1 to 10 numbers

*Without using lambda function

For this we use for loop

```
For i in range(1,11):
        If(i%2==0);
                print(i)
```

OUTPUT

2

4

6

8

10

*Filter function using lambda

```
filter(lambda x:x%2==0,range(1,11))
```

Here we get the filter object as output. In order to get as list we should use

```
list(filter(lambda x:x%2==0,range(1,11)))
```

OUTPUT

[2,4,6,8,10]

In the filter function we can only use 1 iterable.

Reduce Function

Reduce function is used to reduce the iterable into a single element using some function.so output from the reduce function will be a single element. If we want to perform some computation on lists or tuples we use reduce function.

Syntax

reduce(function, iterable)

Example

Sum of all the elements in a list

In python3 reduce function is defined in a module called functools. So for using reduce function we have to import this module first.

Import functools

num=[1,2,3,4]

functools.reduce(lambda x,y:x+y,num)

OUTPUT

10

In the reduce function we can only use 1 iterable.

Python Module

Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or **modules**. Individual modules can then be cobbled together like building blocks to create a larger application.

Modules are used to categorize Python code into smaller parts. A **module is simply a Python file, where classes, functions and variables are defined**. Grouping similar code into a single file makes it easy to access..

Python Module Advantages

Python provides the following advantages for using module:

1**) Reusability**: Module can be used in some other python code. Hence it provides the facility of code reusability.

2) **Categorization:** Similar type of attributes can be placed in one module.

Importing a Module:

There are different ways by which you we can import a module. :

1) Using import statement:

Module contents are made available to the caller with the **import statement.**

Syntax:

import <file_name1, file_name2,... file_name(n) >

Example

```
def add(a,b):
    c=a+b
    print( c)
    return
```

Save the file by the name **addition.py**. To import this file "import" statement is used.

```
import addition
```

Note that this does not make the module contents directly accessible to the caller. Each module has its own private symbol table, which serves as the global symbol table for all objects defined in the module.

The statement import <module_name> only places <module_name> in the caller's symbol table. The objects that are defined in the module remain in the module's private symbol table.

From the caller, objects in the module are only accessible when prefixed with <module_name> via dot notation, as illustrated below.

After the following import statement, addition is placed into the local symbol table. Thus, addition has meaning in the caller's local context: But add remain in the module's private symbol table and are not meaningful in the local context. To

be accessed in the local context, names of objects defined in the module must be prefixed by module name:

addition.add(10,20)

addition.add(30,40)

Output:

>>> 30

70

>>>

NOTE: You can access any function which is inside a module by module name and function name separated by dot. It is also known as period. Whole notation is known as **dot notation.**

2) Python Importing Multiple Modules Example

1) msg.py:

```
def msg_method():
    print ("Today the weather is rainy"  )
    return
```

2) display.py:

```
def display_method():
    print ("The weather is Sunny" )
```

return

3) multiimport.py:

```
import msg, display

msg.msg_method()

display.display_method()
```

Output:

```
>>>

Today the weather is rainy

The weather is Sunny

>>>
```

3) Using from.. import statement:

from..import statement is used to import particular attribute from a module. In case you do not want whole of the module to be imported then you can use from .. import statement.

Syntax:

```
from  <module_name> import <attribute1,attribute2,attribute3,...attributen>
```

Python from.. import Example

1).area.py

```
def circle(r):
    print 3.14*r*r
    return
  def square(l):
    print l*l
    return
  def rectangle(l,b):
    print l*b
    return
  def triangle(b,h):
    print 0.5*b*h
    return
```

2) area1.py

```
from area import square, rectangle
square(10)
rectangle(2,5)
```

Output:

>>>

100

10

>>>

4) To import whole module:

You can import whole of the module using "from …. import *"

Syntax:

from <module_name> import *

Using the above statement all the attributes defined in the module will be imported and hence you can access each attribute.

1) area.py

Same as above example

2) area1.py

```
from area import *

square(10)

rectangle(2,5)

circle(5)

triangle(10,20)
```

Output:

>>>

100

10

78.5

100.0

>>>

Built in Modules in Python:

There are many built in modules in Python. Some of them are as follows:

math, random , threading , collections , os , mailbox , string , time , tkinter etc..

Each module has a number of built in functions which can be used to perform various functions.

1) math:

Using math module , you can use different built in mathematical functions.

Functions:

Function	Description
ceil(n)	**It returns the next integer number of the given number**
sqrt(n)	**It returns the Square root of the given number.**

exp(n)	**It returns the natural logarithm e raised to the given number**
floor(n)	**It returns the previous integer number of the given number.**
log(n,baseto)	**It returns the natural logarithm of the number.**
pow(baseto, exp)	**It returns baseto raised to the exp power.**
sin(n)	**It returns sine of the given radian.**
cos(n)	**It returns cosine of the given radian.**
tan(n)	**It returns tangent of the given radian.**

Python Math Module Example

```
import math

a=4.6

print math.ceil(a)

print math.floor(a)

b=9

print math.sqrt(b)

print math.exp(3.0)

print math.log(2.0)

print math.pow(2.0,3.0)

print math.sin(0)
```

print math.cos(0)

print math.tan(45)

Output:

>>>

5.0

4.0

3.0

20.0855369232

0.69314718056

8.0

0.0

1.0

1.61977519054

>>>

Constants:

The math module provides two constants for mathematical Operations:

Constants	Descriptions

Pi Returns constant ? = 3.14159...

e Returns constant e= 2.71828...

Example

```
import math

print math.pi

print math.e
```

Output:

```
>>>

3.14159265359

2.71828182846
```

2) random:

The random module is used to generate the random numbers. It provides the following two built in functions:

Function	Description
random()	**It returns a random number between 0.0 and 1.0 where 1.0 is exclusive.**
randint(x,y)	**It returns a random number between x and y where both the numbers are inclusive.**

Python Module Example

```
import random

print random.random()

print random.randint(2,8)
```

Output:

```
>>>

0.797473843839

7
```

Packages

We don't usually store all of our files in our computer in the same location. We use a well-organized hierarchy of directories for easier access. Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.

As our application program grows larger in size with a lot of modules, we place **similar modules in one package and different modules in different packages**. This makes a project (program) easy to manage and conceptually clear. Similar, as a directory can contain sub-directories and files, a Python package can have sub-packages and modules.

A package is basically a directory with Python files and a file with the name __init__.py. This means that every directory inside of the Python path, which

contains a file named __init__.py, will be treated as a package by Python. It's possible to put several modules into a Package.

Packages are a way of structuring Python's module namespace by using "dotted module names". A.B stands for a submodule named B in a package named A. Two different packages like P1 and P2 can both have modules with the same name, let's say A, for example. The submodule A of the package P1 and the submodule A of the package P2 can be totally different. A package is imported like a "normal" module.

Steps to create and import Package:

1) Create a directory

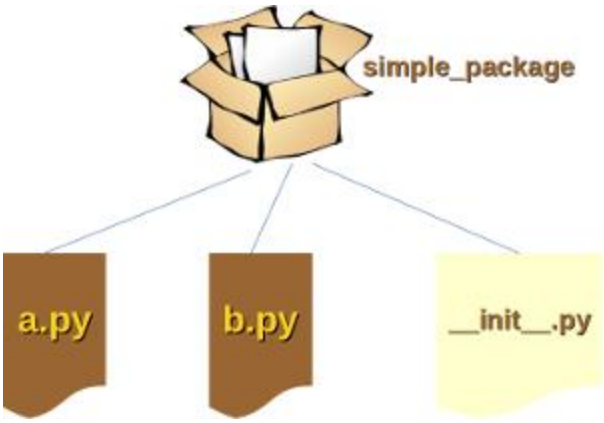
2) Place different modules inside the directory.

3) Create a file __init__.py which specifies attributes in each module.

4) Import the package and use the attributes using package.

A Simple Example

We will demonstrate with a very simple example how to create a package with some Python modules.



First of all, we need a directory. The name of this directory will be the name of the package, which we want to create. We will call our package "simple_package".

1). **Create the directory:**

```
import os

os.mkdir("simple_package")
```

Now we can put into this directory all the Python files which will be the submodules of our module. We create two simple files a.py and b.py just for the sake of filling the package with modules.

2). **Place different modules in package: (Save different modules inside the simple_package package)**

a.py

```
def msg1():
    print ("This is msg1" )
```

b.py

```
def msg2():
   print ("This is msg2" )
```

3).We will also add an empty file with the name __init__.py inside of simple_package directory.

4). Import package and use the attributes:

when we import simple_package from the interactive Python shell, assuming that the directory simple_package is either in the directory from which you call the shell or that it is contained in the search path or environment variable "PYTHONPATH" (from your operating system):

```
>>> import simple_package
>>>
```

We can see that the package simple_package has been loaded but neither the module "a" nor the module "b"! We can import the modules a and b in the following way

```
>>> from simple_package import a, b
>>> a.msg1()
This is msg1
>>> b.msg2()
This is msg2
```

we can't access neither "a" nor "b" by solely importing simple_package. There is a way to automatically load these modules. We can use the file __init__.py for this purpose. All we have to do is add the following lines to the so far empty file __init__.py:

```
import simple_package.a

import simple_package.b
```

It will work now:

```
>>> import simple_package

>>> simple_package.a.msg1()

This is msg1

>>> simple_package.b.msg2()

This is msg2
```

NUMBERS

Number data types store numeric values. They are immutable data types. This means, changing the value of a number data type results in a newly allocated object.

Number objects are created when you assign a value to them.

For example

var1 = 1

var2 = 10

We can also delete the reference to a number object by using the del statement.

The **syntax of the del statement is** −

del var1[,var2[,var3[....,varN]]]]

You can delete a single object or multiple objects by using the del statement.

For example

del var

del var_a, var_b

Python supports different numerical types

1. **int (signed integers):** They are often called just integers or ints. They are positive or negative whole numbers with no decimal point. Integers in Python 3 are of unlimited size. Python 2 has two integer types - int and long. There is no 'long integer' in Python 3 anymore.
2. **float (floating point real values)** : Also called floats, they represent real numbers and are written with a decimal point dividing the integer and the fractional parts. Floats may also be in scientific

notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).

3. **complex (complex numbers)** : are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

Number Type Conversion

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. Sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

Type int(x) to convert x to a plain integer.

Type float(x) to convert x to a floating-point number.

Type complex(x) to convert x to a complex number with real part x and imaginary part zero.

Type complex(x, y) to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions.

Mathematical Functions

Python includes the following functions that perform mathematical calculations.

Sr.No.	Function & Returns (Description)
1	**abs(x)**

	The absolute value of x: the (positive) distance between x and zero.
2	**ceil(x)** The ceiling of x: the smallest integer not less than x.
3	**cmp(x, y)** -1 if x < y, 0 if x == y, or 1 if x > y. **Deprecated** in Python 3. Instead use **return (x>y)-(x<y)**.
4	**exp(x)** The exponential of x: e^x
5	**fabs(x)** The absolute value of x.
6	**floor(x)** The floor of x: the largest integer not greater than x.
7	**log(x)** The natural logarithm of x, for x > 0.
8	**log10(x)** The base-10 logarithm of x for x > 0.
9	**max(x1, x2,...)** The largest of its arguments: the value closest to positive infinity

10	**min(x1, x2,...)** The smallest of its arguments: the value closest to negative infinity.
11	**modf(x)** The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
12	**pow(x, y)** The value of x**y.
13	**round(x [,n])** **x** rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round(0.5) is 1.0 and round(-0.5) is -1.0.
14	**sqrt(x)** The square root of x for x > 0.

Number abs() Method

Description

The abs() method returns the absolute value of x i.e. the positive distance between x and zero.

Syntax

Following is the syntax for abs()

Method

abs(x)

Parameters

x - This is a numeric expression.

Return Value

This method returns the absolute value of x.

Example

```
print ("abs(-45) : ", abs(-45))
print ("abs(100.12) : ", abs(100.12))
```

OUTPUT

```
abs(-45) : 45
abs(100.12) : 100.12
```

Numberceil() Method

Description

The ceil() method returns the ceiling value of x i.e. the smallest integer not less than x.

Syntax

Following is the syntax for the ceil()

method

```
import math
math.ceil( x )
```

Note: This function is not accessible directly, so we need to import math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns the smallest integer not less than x.

Example.

```
import math                    # This will import math module
print ("math.ceil(-45.17) : ", math.ceil(-45.17))
print ("math.ceil(100.12) : ", math.ceil(100.12))
print ("math.ceil(100.72) : ", math.ceil(100.72))
print ("math.ceil(math.pi) : ", math.ceil(math.pi))
```

OUTPUT

```
math.ceil(-45.17) : -45
math.ceil(100.12) : 101
math.ceil(100.72) : 101
math.ceil(math.pi) : 4
```

Number exp() Method

Description

The exp() method returns exponential of x: ex.

Syntax

Following is the syntax for the exp()

Method

```
import math
math.exp( x )
```

Note: This function is not accessible directly. Therefore, we need to import the math module and then we need to call this function using the math static object.

Parameters

X - This is a numeric expression.

Return Value

This method returns exponential of x: ex.

Example

```
import math # This will import math module
print ("math.exp(-45.17) : ", math.exp(-45.17))
print ("math.exp(100.12) : ", math.exp(100.12))
print ("math.exp(100.72) : ", math.exp(100.72))
print ("math.exp(math.pi) : ", math.exp(math.pi))
```

OUTPUT

```
math.exp(-45.17) : 2.4150062132629406e-20
math.exp(100.12) : 3.0308436140742566e+43
math.exp(100.72) : 5.522557130248187e+43
math.exp(math.pi) : 23.140692632779267
```

Number fabs() Method

Description

The fabs() method returns the absolute value of x. Although similar to the abs() function, there are differences between the two functions. They are:- abs() is a built in function whereas fabs() is defined in math module. fabs() function works only on float and integer whereas abs() works with complex number also.

Syntax

Following is the syntax for the fabs()

Method

import math

math.fabs(x)

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric value.

Return Value

This method returns the absolute value of x.

Example

```
import math                    # This will import math module
print ("math.fabs(-45.17) : ", math.fabs(-45.17))
print ("math.fabs(100.12) : ", math.fabs(100.12))
print ("math.fabs(100.72) : ", math.fabs(100.72))
print ("math.fabs(math.pi) : ", math.fabs(math.pi))
```

OUTPUT

```
math.fabs(-45.17) : 45.17
math.fabs(100) : 100.0
math.fabs(100.72) : 100.72
math.fabs(math.pi) : 3.141592653589793
```

Number floor() Method

Description

The floor() method returns the floor of x i.e. the largest integer not greater than x.

Syntax

Following is the syntax for the floor()

method

import math

math.floor(x)

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns the largest integer not greater than x.

Example

```
import math                                # This will import math module
print ("math.floor(-45.17) : ", math.floor(-45.17))
print ("math.floor(100.12) : ", math.floor(100.12))
print ("math.floor(100.72) : ", math.floor(100.72))
print ("math.floor(math.pi) : ", math.floor(math.pi))
```

OUTPUT

```
math.floor(-45.17) : -46
math.floor(100.12) : 100
math.floor(100.72) : 100
math.floor(math.pi) : 3
```

Number log() Method

Description

The log() method returns the natural logarithm of x, for x > 0.

Syntax

Following is the syntax for the log()

method

import math

math.log(x)

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns natural logarithm of x, for x > 0.

Example

```
import math                    # This will import math module
print ("math.log(100.12) : ", math.log(100.12))
print ("math.log(100.72) : ", math.log(100.72))
print ("math.log(math.pi) : ", math.log(math.pi))
```

OUTPUT

```
math.log(100.12) : 4.6063694665635735
math.log(100.72) : 4.612344389736092
math.log(math.pi) : 1.1447298858494002
```

Number log10() Method

Description

The log10() method returns base-10 logarithm of x for x > 0.

Syntax

Following is the syntax for log10()

method

import math

math.log10(x)

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns the base-10 logarithm of x for x > 0.

Example

```
import math              # This will import math module
print ("math.log10(100.12) : ", math.log10(100.12))
print ("math.log10(100.72) : ", math.log10(100.72))
print ("math.log10(119) : ", math.log10(119))
print ("math.log10(math.pi) : ", math.log10(math.pi))
```

OUTPUT

```
resultmath.log10(100.12) : 2.0005208409361854
math.log10(100.72) : 2.003115717099806
math.log10(119) : 2.0755469613925306
math.log10(math.pi) : 0.49714987269413385
```

Number max() Method

Description

The max() method returns the largest of its arguments i.e. the value closest to

positive infinity.

Syntax

Following is the syntax for max()

method

max(x, y, z, ...)

Parameters

x - This is a numeric expression.

y - This is also a numeric expression.

z - This is also a numeric expression.

Return Value

This method returns the largest of its arguments.

Example

```
print ("max(80, 100, 1000) : ", max(80, 100, 1000))
print ("max(-20, 100, 400) : ", max(-20, 100, 400))
print ("max(-80, -20, -10) : ", max(-80, -20, -10))
print ("max(0, 100, -400) : ", max(0, 100, -400))
```

OUTPUT

```
max(80, 100, 1000) : 1000
max(-20, 100, 400) : 400
max(-80, -20, -10) : -10
max(0, 100, -400) : 100
```

Number min() Method

Description

The method min() returns the smallest of its arguments i.e. the value closest to

negative infinity.

Syntax

Following is the syntax for the min() methodmin(x, y, z, ...)

Parameters

x - This is a numeric expression.

y - This is also a numeric expression.

z - This is also a numeric expression.

Return Value

This method returns the smallest of its arguments.

Example

```
print ("min(80, 100, 1000) : ", min(80, 100, 1000))
print ("min(-20, 100, 400) : ", min(-20, 100, 400))
print ("min(-80, -20, -10) : ", min(-80, -20, -10))
print ("min(0, 100, -400) : ", min(0, 100, -400))
```

OUTPUT

```
min(80, 100, 1000) : 80
min(-20, 100, 400) : -20
min(-80, -20, -10) : -80
min(0, 100, -400) : -400
```

Numbermodf() Method

Description

The modf() method returns the fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.

Syntax

Following is the syntax for the modf()

method

import math

math.modf(x)

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns the fractional and integer parts of x in a two-item tuple. Both the parts have the same sign as x. The integer part is returned as a float.

Example

```
import math                          # This will import math module
print ("math.modf(100.12) : ", math.modf(100.12))
print ("math.modf(100.72) : ", math.modf(100.72))
print ("math.modf(119) : ", math.modf(119))
print ("math.modf(math.pi) : ", math.modf(math.pi))
```

OUTPUT

```
math.modf(100.12) : (0.12000000000000455, 100.0)
math.modf(100.72) : (0.7199999999999989, 100.0)
math.modf(119) : (0.0, 119.0)
math.modf(math.pi) : (0.14159265358979312, 3.0)
```

Number pow() Method

Return Value

This method returns the value of xy.

Example

```
import math                    # This will import math module
print ("math.pow(100, 2) : ", math.pow(100, 2))
print ("math.pow(100, -2) : ", math.pow(100, -2))
print ("math.pow(2, 4) : ", math.pow(2, 4))
print ("math.pow(3, 0) : ", math.pow(3, 0))
```

OUTPUT

```
math.pow(100, 2) : 10000.0
math.pow(100, -2) : 0.0001
math.pow(2, 4) : 16.0
math.pow(3, 0) : 1.0
```

Number round() Method

Description

round() is a built-in function in Python. It returns x rounded to n digits from the decimal point.

Syntax

Following is the syntax for the round() methodround(x [, n])

Parameters

x - This is a numeric expression.

n - Represents number of digits from decimal point up to which x is to be rounded

Default is 0.

Return Value

This method returns x rounded to n digits from the decimal point.

Example

```
print ("round(70.23456) : ", round(70.23456))
print ("round(56.659,1) : ", round(56.659,1))
print ("round(80.264, 2) : ", round(80.264, 2))
print ("round(100.000056, 3) : ", round(100.000056, 3))
print ("round(-100.000056, 3) : ", round(-100.000056, 3))
```

OUTPUT

```
round(70.23456) : 70
round(56.659,1) : 56.7
round(80.264, 2) : 80.26
round(100.000056, 3) : 100.0
round(-100.000056, 3) : -100.0
```

Number sqrt() Method

Description

The sqrt() method returns the square root of x for x > 0.

Syntax

Following is the syntax for sqrt()

method

```
import math
math.sqrt( x )
```

Note: This function is not accessible directly, so we need to import the math

module and then we need to call this function using the math static object.

Parameters

x - This is a numeric expression.

Return Value

This method returns square root of x for x > 0.

Example

```
import math                    # This will import math module
print ("math.sqrt(100) : ", math.sqrt(100))
print ("math.sqrt(7) : ", math.sqrt(7))
print ("math.sqrt(math.pi) : ", math.sqrt(math.pi))
```

OUTPUT

```
math.sqrt(100) : 10.0
math.sqrt(7) : 2.6457513110645907
math.sqrt(math.pi) : 1.7724538509055159
```

Random Number Functions

Random numbers are used for games, simulations, testing, security, and privacy applications. Python includes the following functions that are commonly used.

Sr.No.	Function & Description
1	**choice(seq)** A random item from a list, tuple, or string.

2	**randrange ([start,] stop [,step])** A randomly selected element from range(start, stop, step).
3	**random()** A random float r, such that 0 is less than or equal to r and r is less than 1
4	**seed([x])** Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
5	**shuffle(lst)** Randomizes the items of a list in place. Returns None.
6	**uniform(x, y)** A random float r, such that x is less than or equal to r and r is less than y.

Number choice() Method

Description

The choice() method returns a random item from a list, tuple, or string.

Syntax

Following is the syntax for choice()

Method

choice(seq)

Note: This function is not accessible directly, so we need to import the random

module and then we need to call this function using the random static object.

Parameters

seq - This could be a list, tuple, or string...

Return Value

This method returns a random item.

Example

```
import random
print ("random number from range(100) ",random.choice(range(100)))
print ("random element from list [1, 2, 3, 5,9]:",random.choice([1,2,3,5,9]))
print ("random character from string 'Hello World' : ",
random.choice('Hello World'))
```

OUTPUT

```
random number from range(100) : 19
random element from list [1, 2, 3, 5, 9]) : 9
random character from string 'Hello World' : r
```

Number randrange() Method

Description

The randrange() method returns a randomly selected element from range(start, stop,step).

Syntax

Following is the syntax for the randrange() method-

```
randrange ([start,] stop [,step])
```

Note: This function is not accessible directly, so we need to import the random

moduleand then we need to call this function using the random static object.

Parameters

start - Start point of the range. This would be included in the range. Default is 0.

stop - Stop point of the range. This would be excluded from the range.

step - Value with which number is incremented. Default is 1.

Return Value

This method returns a random item from the given range.

Example

```
import random

# randomly select an odd number between 1-100

print ("randrange(1,100, 2) : ", random.randrange(1, 100, 2))

# randomly select a number between 0-99

print ("randrange(100) : ", random.randrange(100))
```

OUTPUT

```
randrange(1,100, 2) : 83

randrange(100) : 93
```

Number random() Method

Description

The random() method returns a random floating point number in the range [0.0, 1.0].

Syntax

Following is the syntax for the random()

method

random ( )

Note: This function is not accessible directly, so we need to import the random

module and then we need to call this function using the random static object.

Parameters

NA

Return Value

This method r**eturns a random float r, such that 0.0 <= r <= 1.0**

Example

```
import random

# First random number

print ("random() : ", random.random())

# Second random number

print ("random() : ", random.random())
```

OUTPUT

```
random() : 0.281954791393

random() : 0.309090465205
```

Number seed() Method

Description

The seed() method initializes the basic random number generator. Call this function before calling any other random module function.

Syntax

Following is the syntax for the seed()

Method

seed ([x], [y])

Note: This function initializes the basic random number generator.

Parameters

x - This is the seed for the next random number. If omitted, then it takes system time to generate the next random number. If x is an int, it is used directly.

Y - This is version number (default is 2). str, byte or byte array object gets converted in int. Version 1 used hash() of x.

Return Value

This method does not return any value.

Example

```
import random
random.seed()
print ("random number with default seed", random.random())
random.seed(10)
print ("random number with int seed", random.random())
random.seed("hello",2)
print ("random number with string seed", random.random())
```

OUTPUT

```
random number with default seed 0.2524977842762465
random number with int seed 0.5714025946899135
random number with string seed 0.3537754404730722
```

Number shuffle() Method

Description

The shuffle() method randomizes the items of a list in place.

Syntax

Following is the syntax for the shuffle()

Method

shuffle (lst,[random])

Note: This function is not accessible directly, so we need to import the shuffle module and then we need to call this function using the random static object.

Parameters

lst - This could be a list or tuple.

random - This is an optional 0 argument function returning float between 0.0 - 1.0. Default is None.

Return Value

This method returns reshuffled list.

Example

```
import random
list = [20, 16, 10, 5];
random.shuffle(list)
print ("Reshuffled list : ", list)
random.shuffle(list)
print ("Reshuffled list : ", list)
```

OUTPUT

```
Reshuffled list : [16, 5, 10, 20]
reshuffled list : [20, 5, 10, 16]
```

Number uniform() Method

Description

The uniform() method returns a random float r, such that x is less than or equal to r and r is less than y.

Syntax

Following is the syntax for the uniform()

method

uniform(x, y)

Note: This function is not accessible directly, so we need to import the uniform module and then we need to call this function using the random static object.

Parameters

x - Sets the lower limit of the random float.

y - Sets the upper limit of the random float.

Return Value

This method returns a floating point number r such that x <=r < y.

Example

The following example shows the usage of the uniform() method.

```
import random
print ("Random Float uniform(5, 10) : ", random.uniform(5, 10))
print ("Random Float uniform(7, 14) : ", random.uniform(7, 14))
```

OUTPUT

```
Random Float uniform(5, 10) : 5.52615217015
Random Float uniform(7, 14) : 12.5326369199
```

Trigonometric Functions

Python includes the following functions that perform trigonometric calculations.

Sr.No.	Function & Description
1	**acos(x)**

	Return the arc cosine of x, in radians.
2	**asin(x)** Return the arc sine of x, in radians.
3	**atan(x)** Return the arc tangent of x, in radians.
4	**atan2(y, x)** Return atan(y / x), in radians.
5	**cos(x)** Return the cosine of x radians.
6	**hypot(x, y)** Return the Euclidean norm, sqrt(x*x + y*y).
7	**sin(x)** Return the sine of x radians.
8	**tan(x)** Return the tangent of x radians.
9	**degrees(x)** Converts angle x from radians to degrees.
10	**radians(x)** Converts angle x from degrees to radians.

Number acos() Method

Description

The acos() method returns the arc cosine of x in radians.

Syntax

Following is the syntax for acos()

Method

acos(x)

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value in the range -1 to 1. If x is greater than 1 then it will generate 'math domain error'.

Return Value

This method returns arc cosine of x, in radians.

Example

```
import math
print ("acos(0.64) : ", math.acos(0.64))
print ("acos(0) : ", math.acos(0))
print ("acos(-1) : ", math.acos(-1))
print ("acos(1) : ", math.acos(1))
```

OUTPUT

```
acos(0.64) : 0.876298061168
acos(0) : 1.57079632679
acos(-1) : 3.14159265359
acos(1) : 0.0
```

Number asin() Method

Description

The asin() method returns the arc sine of x (in radians).

Syntax

Following is the syntax for the asin()

method

asin(x)

Note: This function is not accessible directly, so we need to import the math module andthen we need to call this function usingthe math static object.

Parameters

x - This must be a numeric value in the range -1 to 1. If x is greater than 1 then it will generate 'math domain error'.

Return Value

This method returns arc sine of x, in radians.

Example

```
import math
print ("asin(0.64) : ", math.asin(0.64))
print ("asin(0) : ", math.asin(0))
print ("asin(-1) : ", math.asin(-1))
print ("asin(1) : ", math.asin(1))
```

OUTPUT

```
asin(0.64) : 0.694498265627
asin(0) : 0.0
asin(-1) : -1.57079632679
asin(1) : 1.5707963267
```

Number atan() Method

Description

The atan() method returns the arc tangent of x, in radians.

Syntax

Following is the syntax for atan()

method

atan(x)

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns arc tangent of x, in radians.

Example

```
import math
print ("atan(0.64) : ", math.atan(0.64))
print ("atan(0) : ", math.atan(0))
print ("atan(10) : ", math.atan(10))
print ("atan(-1) : ", math.atan(-1))
print ("atan(1) : ", math.atan(1))
```

OUTPUT

```
atan(0.64) : 0.569313191101
atan(0) : 0.0
atan(10) : 1.4711276743
atan(-1) : -0.785398163397
```

atan(1) : 0.785398163397

Number atan2() Method

Description

The atan2() method returns atan(y / x), in radians.

Syntax

Following is the syntax for atan2()

method

atan2(y, x)

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

y - This must be a numeric value.

x - This must be a numeric value.

Return Value

This method returns atan(y / x), in radians.

Example

```
import math
print ("atan2(-0.50,-0.50) : ", math.atan2(-0.50,-0.50))
print ("atan2(0.50,0.50) : ", math.atan2(0.50,0.50))
print ("atan2(5,5) : ", math.atan2(5,5))
print ("atan2(-10,10) : ", math.atan2(-10,10))
print ("atan2(10,20) : ", math.atan2(10,20))
```

OUTPUT

```
atan2(-0.50,-0.50) : -2.35619449019
atan2(0.50,0.50) : 0.785398163397
```

atan2(5,5) : 0.785398163397

atan2(-10,10) : -0.785398163397

atan2(10,20) : 0.463647609001

Number cos() Method

Description

The cos() method returns the cosine of x radians.

Syntax

Following is the syntax for cos()

method

cos(x)

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns a numeric value between -1 and 1, which represents the cosine of the angle.

Example

```
import math
print ("cos(3) : ", math.cos(3))
print ("cos(-3) : ", math.cos(-3))
print ("cos(0) : ", math.cos(0))
print ("cos(math.pi) : ", math.cos(math.pi))
print ("cos(2*math.pi) : ", math.cos(2*math.pi))
```

OUTPUT

cos(3) : -0.9899924966

cos(-3) : -0.9899924966

cos(0) : 1.0

cos(math.pi) : -1.0

cos(2*math.pi) : 1.0

Number hypot() Method

Description

The method hypot() return the Euclidean norm, sqrt(x*x + y*y). This is length of vector from origin to point (x,y)

Syntax

Following is the syntax for hypot()

method

hypot(x, y)

Note: This function is not accessible directly, so we need to import math module and then we need to call this function using math static object.

Parameters

x - This must be a numeric value.

y - This must be a numeric value.

Return Value

This method returns Euclidean norm, sqrt(x*x + y*y).

Example

```
import math
print ("hypot(3, 2) : ", math.hypot(3, 2))
print ("hypot(-3, 3) : ", math.hypot(-3, 3))
```

```
print ("hypot(0, 2) : ", math.hypot(0, 2))
```

OUTPUT

```
hypot(3, 2) : 3.60555127546
hypot(-3, 3) : 4.24264068712
hypot(0, 2) : 2.0
```

Number sin() Method

Description

The sin() method returns the sine of x, in radians.

Syntax

Following is the syntax for sin()

method

```
sin(x)
```

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns a numeric value between -1 and 1, which represents the sine of the parameter x.

Example

```
import math
print ("sin(3) : ", math.sin(3))
print ("sin(-3) : ", math.sin(-3))
print ("sin(0) : ", math.sin(0))
print ("sin(math.pi) : ", math.sin(math.pi))
```

```
print ("sin(math.pi/2) : ", math.sin(math.pi/2))
```

OUTPUT

```
sin(3) : 0.14112000806
sin(-3) : -0.14112000806
sin(0) : 0.0
sin(math.pi) : 1.22460635382e-16
sin(math.pi/2) : 1
```

Number tan() Method

Description

The tan() method returns the tangent of x radians.

Syntax

Following is the syntax for tan()

method.

```
tan(x)
```

Note: This function is not accessible directly, so we need to import math module and then we need to call this function using math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns a numeric value between -1 and 1, which represents the tangent of the parameter x.

Example

```
import math
print ("(tan(3) : ", math.tan(3))
print ("tan(-3) : ", math.tan(-3))
```

```
print ("tan(0) : ", math.tan(0))
print ("tan(math.pi) : ", math.tan(math.pi))
print ("tan(math.pi/2) : ", math.tan(math.pi/2))
print ("tan(math.pi/4) : ", math.tan(math.pi/4))
```

OUTPUT

```
print ("(tan(3) : ", math.tan(3))
print ("tan(-3) : ", math.tan(-3))
print ("tan(0) : ", math.tan(0))
print ("tan(math.pi) : ", math.tan(math.pi))
print ("tan(math.pi/2) : ", math.tan(math.pi/2))
print ("tan(math.pi/4) : ", math.tan(math.pi/4))
```

Number degrees() Method

Description

The degrees() method converts angle x from radians to degrees..

Syntax

Following is the syntax for degrees()

Method

degrees(x)

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns the degree value of an angle.

Example

```
import math
print ("degrees(3) : ", math.degrees(3))
print ("degrees(-3) : ", math.degrees(-3))
print ("degrees(0) : ", math.degrees(0))
print ("degrees(math.pi) : ", math.degrees(math.pi))
print ("degrees(math.pi/2) : ", math.degrees(math.pi/2))
print ("degrees(math.pi/4) : ", math.degrees(math.pi/4))
```

OUTPUT

```
degrees(3) : 171.88733853924697
degrees(-3) : -171.88733853924697
degrees(0) : 0.0
degrees(math.pi) : 180.0
degrees(math.pi/2) : 90.0
degrees(math.pi/4) : 45.0
```

Number radians() Method

Description

The radians() method converts angle x from degrees to radians.

Syntax

Following is the syntax for radians()

Method

radians(x)

Note: This function is not accessible directly, so we need to import the math module and then we need to call this function using the math static object.

Parameters

x - This must be a numeric value.

Return Value

This method returns radian value of an angle.

Example

```
import math
print ("radians(3) : ", math.radians(3))
print ("radians(-3) : ", math.radians(-3))
print ("radians(0) : ", math.radians(0))
print ("radians(math.pi) : ", math.radians(math.pi))
print ("radians(math.pi/2) : ", math.radians(math.pi/2))
print ("radians(math.pi/4) : ", math.radians(math.pi/4))
```

OUTPUT

```
radians(3) : 0.0523598775598
radians(-3) : -0.0523598775598
radians(0) : 0.0
radians(math.pi) : 0.0548311355616
radians(math.pi/2) : 0.0274155677808
radians(math.pi/4) : 0.0137077838904
```

Mathematical Constants

The module also defines two mathematical constants

Constants	Description
pi	The mathematical constant pi.
e	The mathematical constant e.