PYTHON PROGRAMMING

UNIT – I **FUNDAMENTALS OF PYTHON**

Introduction to Python – Advantages of Python programming – Variables and Datatypes – Comments – I/O function –Operators – Selection control structures – Looping control structures –Functions: Declaration – Types of arguments – Anonymous functions: Lambda.

Introduction

Fundamentals of Python

Python is a general-purpose, interpreted, interactive, object-oriented, open source, robust and high-level programming language. Python is easy to learn yet powerful and versatile scripting language. It was created by Guido van Rossum during 1985- 1990.It is ideal for scripting and rapid application development.Its design makes it very readable. Python makes the development and debugging fast because there is no compilation step included in python development and edit-test-debug cycle is very fast.Python is useful for accomplishingreal-world tasks—the sorts of things developers do day in and day out.It's commonly used in a variety of domains, as a tool for scripting other componentsand implementing standalone programs.It is widely used in web development, scientific and mathematical computing. Python is most widely used in following domains

- Game programming and multimedia
- Image processing
- Natural language analysis
- Artificial intelligence
- Document processing and generation
- Data visualization
- Data mining

Also, it uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.The syntax (words and structure) is extremely simple to read and follow.

Origins

Work on python began in the late 1980s. The implementation of Python was started in the December 1989 by Guido Van Rossum at CWI in Netherland. It was released for public distribution in early 1991. Like all other programming languages like C, C++, Lisp and Java, Python was developed from research background. Python is a successor of interpreted language ABC, capable of exception handling and interfacing with the Amoeba operating system.

- Python version 1.0 was released in January 1994. The major new features included in this release were the functional programming tools lambda, map, filter and reduce.
- In October 2000, Python 2.0 was introduced. This release included list comprehensions, a full garbage collector and it was supporting unicode.

- Python 3.0 was released on 3 December 2008. It was designed to rectify fundamental flaw of the language.

Features

Python provide lot of features for the programmers.

Object-Oriented and Functional

Python is an object-oriented language, from the ground up. It follow object and class concept. Its class model supportsadvanced notions such as polymorphism, operator overloading, and multiple inheritance; Like C++, Python supports procedure-oriented programming as well as object-oriented programming.In addition to its original procedural (statement-based) and object-oriented (classbased)paradigms, python borrows functional programming concepts —a set that by most measures includes generators, comprehensions, closures,maps, decorators, anonymous function lambdas, and first-class function objects.

Free and Open Source

Python is completely free to use and distribute. As with other open source software,such as Tcl, Perl, Linux, and Apache, you can fetch the entire Python system's sourcecode for free on the Internet. There are no restrictions on copying it, embedding it inyour systems, or shipping it with your products. This is called FLOSS(Free/Libre and Open Source Software).

Portable

The standard implementation of Python is written in portable ANSI C, and it compilesand runs on virtually every major platform currently in use. For example, Python programsrun today on everything from PDAs to supercomputers. Python can run equally on different platforms such as Windows, Linux, Unix and Macintosh etc.

Relatively Easy to Use and Learn

Compared to alternative languages, python programming is extremely simple to learn. It offers an easy to understand syntax, simple setup, and has many practical applications in web development. To run a Python program, you simply type it and
run it. There are no intermediate compile and link steps, like there are for languagessuch as C or C++. Python executes programs immediately, which makes for an interactive
programming experience and rapid turnaround after program changes

Interpreted

Python is an interpreted language i.e. interpreter executes the code line by line at a time. When you use an interpreted language like Python, there is no separate compilation and execution steps. You just run the program from the source code. This makesdebugging easy and thus suitable for beginners. Internally, Python converts the source code into an intermediate form

called bytecodes and then translates this into the native language of your specific computer and then runs it. You just run your programs and you never have to worry about linking and loading with libraries, etc.

Extensible

If needed, Python code can be written in other languages like C++. This makes Python an extensible language, meaning that it can be extended to other languages. Python extensions can be written in C and C++ for the standard implementation of python in C. The Java language implementation of python is called Jython. Finally, there is Ironpython, the C# implementation for the .NET.

Powerful

From a features perspective, Python is something of a hybrid. Its toolset places it betweentraditional scripting languages (such as Tcl, Scheme, and Perl) and systems developmentlanguages (such as C, C++, and Java).Python is useful for large-scale development projects. The following features make Python language more powerful.

- Dynamic typing
- Automatic memory management
- Programming-in-the-large support
- Built-in object types
- Built-in tools
- Library utilities

Advantages of Python Programming

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

Variables

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

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

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

Example

```
counter = 0
miles = 100.0
name = 'abc'
counter = counter + 1
```

Assignment does not copy a value; it just attaches a name to the object that contains the data. The name is a reference to a thing rather than the thing itself.

The following are the examples of assignment

```
>>> a = 7
>>>print(a)
7
>>> b = a
>>>print(b)
7
```

The above code performs the following steps

1. Assign the value 7 to the name a. This creates an object box containing the integer value 7.
2. Print the value of a.
3. Assign a to b, making b also stick to the object box containing 7.
4. Print the value of b

Keywords

Keywords are the reserved words in Python which convey a special meaning to the interpreter. Each keyword have a special meaning and a specific operation. These keywords cannot be used as variable name, function name or any other identifier.

Here's a list of all keywords in Python Programming

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

The above keywords may get altered in different versions of Python. Some extra might get added or some might be removed. List of keywords in current version is obtained by typing the following in the prompt.

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

Data Types

Variables can hold values of different data types. Python is a dynamically typed language hence we need not define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.

Python enables us to check the type of the variable used in the program. Python provides us the **type()** function which returns the type of the variable passed.

Consider the following example to define the values of different data types and checking its type.

```
>>>a=25
>>>b="Welcome"
>>>c = 40.3
>>>type(a)
  <class 'int'>
>>>type(b)
  <class 'str'>
>>>type(c);
  <class 'float'>
```

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

1. Boolean
2. Numbers
3. Strings
4. Lists
5. Tuples
6. Dictionaries
7. Sets

Boolean

Booleans are either true or false. Python has two constants, cleverly named True and False, which can be used to assign boolean values directly. Expressions can also evaluate to a boolean value. In certain places, Python expects an expression to evaluate to a boolean value. These places are called boolean contexts.

```
>>>x=True
>>>type(x)
<class 'bool'>
>>>size = 1
>>>size< 0
False
>>>size = 0
>>>size< 0
False
>>>size = -1
>>>size< 0
True
```

Booleans can be treated as numbers. True is 1; False is 0.

```
>>> True + True
 2
>>> True - False
1
>>> True * False
0
>>> True / False
Traceback (most recent call last):
 File "<stdin>", line 1, in<module>
ZeroDivisionError: division by zero
```

Numbers

Number stores numeric values. Python creates Number objects when a number is assigned to a variable. Python has three distinct numeric types

- Integers
  Integers represent negative and positive integers without fractional parts (Integers like 10, 2, 29,-45,-3 etc.).
- Floating point numbers
  Floating point numbers represents negative and positive numbers with fractional parts(float is used to store floating point numbers like 1.9, 9.902, -56.3 etc.).
- Complex numbers
  Mathematically, a complex number (generally used in engineering) is a number of the form A+Bi where i is the imaginary number. Complex numbers have a real and imaginary part. Python supports complex numbers either by specifying the number in (real + imag**J**) or (real + imag**j**) form or using a built-in method complex(x, y).

```
>>>a=123

>>>type(a)

<class 'int'>

>>>b=23.12

>>>type(b)

<class 'float'>

>>>c=-56

>>>type(c)

<class 'int'>
>>>x=complex(1,2)
>>>type(x)
<class 'complex'>
```

Input and Output Statement

Input Statement

The input() function allows user input. The syntax for input() is

input(*prompt*)

where prompt is the string we wish to display on the screen. It is optional.

```
>>>num=input('Enter a number: ')
```

Enter a number:10

>>>num

'10'

Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use int() or float() functions.

Taking multiple inputs from user in Python

Developer often wants a user to enter multiple values or inputs in one line. Python user can take multiple values or inputs in one line by two methods.

- Using split() method
- Using List comprehension

Using split() method :

This function helps in getting a multiple inputs from user . It breaks the given input by the specified separator. If separator is not provided then any white space is a separator. Generally, user use a split() method to split a Python string but one can used it in taking multiple input.

Syntax :

```
input().split(separator, maxsplit)
```

```
x, y =input("Enter a two value: ").split()
print("Number of boys: ", x)
print("Number of girls: ", y)
print()

# taking three inputs at a time
x, y, z =input("Enter a three value: ").split()
print("Total number of students: ", x)
print("Number of boys is : ", y)
print("Number of girls is : ", z)
print()

# taking two inputs at a time
a, b =input("Enter a two value: ").split()
print("First number is {} and second number is {}".format(a, b))
print()

# taking multiple inputs at a time
# and type casting using list() function
x =list(map(int, input("Enter a multiple value: ").split()))
print("List of students: ", x)
```

Using List comprehension

List comprehension is an elegant way to define and create list in Python. We can create lists just like mathematical statements in one line only. It is also used in getting multiple inputs from a user.

```
x, y =[int(x) forxininput("Enter two value: ").split()]
print("First Number is: ", x)
```

```
print("Second Number is: ", y)
print()

# taking three input at a time
x, y, z =[int(x) forxininput("Enter three value: ").split()]
print("First Number is: ", x)
print("Second Number is: ", y)
print("Third Number is: ", z)
print()

# taking two inputs at a time
x, y =[int(x) forxininput("Enter two value: ").split()]
print("First number is {} and second number is {}".format(x, y))
print()

x =[int(x) forxininput("Enter multiple value: ").split()]
print("Number of list is: ", x)
```

Reading multiple integer values

Method1

```
x,y,z = map(int,input("Enter three values: ").split(",") )
sum=x+y+z
print(sum)
```

Method2

```
x,y,z = [int(x) for x in input("Enter three values").split()]
sum=x+y+z
print(sum)
```

Output Statement

The print() function used to output data to the standard output device.

```
print('This sentence is output to the screen')
# Output: This sentence is output to the screen
a = 5
print('The value of a is', a)
# Output: The value of a is 5
```

Use of Separator in print

It's possible to redefine the separator between values by assigning an arbitrary string to the keyword parameter "sep", i.e. an empty string or a smiley

```
>>>print("a","b")

a b

>>>print("a","b",sep="")

ab

>>>print(192,168,178,42,sep=".")

192.168.178.42
```

```
>>>print("a","b",sep=":-)")
a:-)b
```

Use of end in print

By default, python's print() function ends with a newline. This function comes with a parameter called 'end.' The default value of this parameter is '\n,' i.e., the new line character. You can end a print statement with any character or string using this parameter.

```
fori in range(4):
        print(i)
```

Output

```
0
1
2
3
```

```
fori in range(4):
print(i, end=" ")
```

Output

```
0 1 2 3
```

Comments in Python

Comments are lines that exist in computer programs that are ignored by compilers and interpreters. Including comments in programs makes code more readable for humans as it provides some information or explanation about what each part of a program is doing.

In Python, there are two ways to ways to include comments.The first is to include comments that detail or indicate what a section of code – or snippet – does.The second makes use of multi-line comments or paragraphs that serve as documentation for others reading your code.

Single-line comments

Single-line comments are created simply by beginning a line with the hash (#) character, and they are automatically terminated by the end of line.

For example:

```
#Thisis the comment in Python
```

Comments that span multiple lines are used to explain things in more detail are created by adding a delimiter (“””) on each end of the comment.

“”” This would be a multiline commentin Python that includes several lines and describesthecode,or anything you want it to…
“””

Operators

Operators are special symbols which can manipulate the value of operands. Operators can manipulate individual items and returns a result.

Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

Python language supports the following types of operators.

- Arithmetic Operators
- Relational Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication and division etc. Assume variable a holds 10 and variable b holds 20, then

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10
* Multiplication	Multiplies values on either side of the operator	a * b = 200

/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to 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

Relational Operators

Relational operators compare the values of two operands. It either returns True or False according to the condition. Assume variable a holds 10 and variable b holds 20, then

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) = False
!=	If values of two operands are not equal, then condition becomes true.	(a != b) = True.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) = True. This is similar to != operator.

>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) = False.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) = Ttrue.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b)=False.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) = True.

Assignment Operators

Python assignment operators are used for assigning the value of the right operand to the left operand. Various assignment operators used in Python are (+=, - = , *=, /= , etc.)

Logical Operators

Logical operators in Python are used for conditional statements are true or false. Logical operators in Python are AND, OR and NOT. For logical operators following condition are applied.

- For AND operator – It returns TRUE if both the operands (right side and left side) are true
- For OR operator- It returns TRUE if either of the operand (right side or left side) is true
- For NOT operator- returns TRUE if operand is false

Example

```
a = True
b = False
print(('a and b is',a and b))
print(('a or b is',a or b))
print(('not a is',not a))
```

Bitwise Operators

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

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

Operator	Meaning	Example
&	Bitwise AND	x& y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)
^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x>> 2 = 2 (0000 0010)
<<	Bitwise left shift	x<< 2 = 40 (0010 1000)

Membership operators

in and ***not in*** are the membership operators in Python. They are used to test whether a value or variable is found in a sequence.

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

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Identity operators

is and ***is not*** are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Identity operators in Python		
Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

Control Statements

Control Statement in Python performs different computations or actions depending on whether a specific Boolean constraint evaluates to true or false. These statements allow the computer to select or repeat an action. Control statement selects one option among all options and repeats specified section of the program.

Selection control Structures

Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in python are:

- if statement
- if..else statements
- nested if statements
- if-elif ladder

<u>if statement</u>

if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
ifcondition:
  # Statements to execute if
  # condition is true
```

Python uses indentation to identify a block.

 # Python program to illustrate If statement

i = 20

```
if (i> 10):
   print ("10 is less than 20")
print ("I am Not in if")
```

if- else Statement

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won’t. But what if we want to do something else if the condition is false. Here comes the *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Syntax:

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```

 # python program to illustrate If else statement

```
a =20
b = 30
if(a<b):
    print("a is smaller than b")
else:
    print("a is greater than b")
```

nested-if Statement

A nested if is an if statement that is the target of another if statement. Nested if statements means an if statement inside another if statement. Yes, Python allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1):

   # Executes when condition1 is true

if (condition2):

      # Executes when condition2 is true

   # if Block is end here

# if Block is end here
```

```
# python program to illustrate nested If statement
i = 10
if (i == 10):
    #  First if statement
    if (i< 15):
        print ("i is smaller than 15")
    # Nested - if statement
    # Will only be executed if statement above
    # it is true
```

```
if (i< 12):
    print ("i is smaller than 12 too")
else:
    print ("i is greater than 15")
```

if-elif-else ladder
Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:-

```
if (condition):

statement

elif (condition):

statement

.

.

else:

statement
```

Python program to illustrate if-elif-else ladder

```
a,b,c=10,20,30
if (a >= b) and (a >= b):
largest = a

elif (b >= c):
largest = b
else:
largest = c
print("Largest among three numbers",largest)
```

Loops in python

Python programming language provides following types of loops to handle looping requirements. Python provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

1. **While Loop:**
   In python, while loop is used to execute a block of statements repeatedly until a given a condition is satisfied. And when the condition becomes false, the line immediately after the loopin program is executed.
   Syntax :

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

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.
Example:

```
# Python program to illustratewhile loop
count =0
while(count < 3):
    count =count +1
    print("Welcome")
```

Using else statement with while loops:
While loop executes the block until a condition is satisfied. When the condition becomes false, the statement immediately after the loop is executed. The else clause is only executed when your while condition becomes false.

Syntax

```
whilecondition:
     # execute these statements
else:
     # execute these statements



#Python program to illustratecombining else with while
count =0
while(count < 3):
    count =count +1
    print("Welcome")
else:
    print("In Else Block")
```

for Loop Statements
It has the ability to iterate over the items of any sequence, such as a list or a string.
Syntax
foriterating_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.

Example

```
for letter in 'Python':     # First Example
print 'Current Letter :', letter

fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Second Example
print 'Current fruit :', fruit
```

Using else statement with for loops:

If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 20.

Example

```
for num in range(10,20):
fori in range(2,num):
ifnum%i == 0:
        j=num/i
print( '%d equals %d * %d' % (num,i,j))
break
else:
print num, 'is a prime number'
```

Loop Control Statements

Loop control statements change execution from its normal sequence.

break statement

It terminates the current loop and resumes execution at the next statement. 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

Example

```
var = 10
whilevar> 0:
print 'Current variable value :', var
var = var -1
ifvar == 5:
break
```

Output

Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6

Continue statement

It returns the control to the beginning of the while 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
```

Example

```
var = 10
whilevar> 0:
var = var -1
ifvar == 5:
continue
print(“'Current variable value :”, var)
```

Output

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

pass Statement

It is used when a statement is required syntactically but you do not want any command or code to execute.

The pass statement is a *null* operation; nothing happens when it executes. The pass is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example

Syntax

```
pass
```

Example

```
for letter in 'Python':
if letter == 'h':
pass
print 'This is pass block'
print 'Current Letter :', letter
```

Output

Current Letter : P
Current Letter : y
Current Letter : t
This is pass block
Current Letter : h
Current Letter : o
Current Letter : n

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.Python provides many built-in functions like print(), etc. but the user can also create their own functions. These functions are called *user-defined functions.*

Defining a Function

Functions can be defined 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 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.

Syntax of Function

deffunction_name(parameters):

"""docstring"""

statement(s)

Example of a function

```
def greet(name):
    """This function greets to
    the person passed in as
    parameter"""
    print("Hello, " + name + ". Good morning!")
```

Types of arguments

In Python, user-defined functions can take four different types of arguments. The argument types and their meanings, however, are pre-defined and can’t be changed. But a developer can, instead, follow these pre-defined rules to make their own custom functions. The following are the four types of arguments

- Default arguments

- Required arguments
- Keyword arguments
- Variable-length arguments

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. Below is a typical syntax for default argument. In function call2 the default value of age is 35.

defemp(name, age = 35):
print("Name: ", name)
print("Age: ", age)

emp(age=50, name="arun") #Function call11
emp(name="Anand") #Function call2

Output
Name: arun
Age 50
Name: Anand
Age 35

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.

defemp(name, age):
print("Name: ", name)
print("Age: ", age)
emp(age=50, name="arun") #Function call11
emp(name="Anand") #Function call2

To call the function emp(), you definitely need to pass two arguments, otherwise it gives a syntax error as follows
Name: arun
Age 50
Traceback (most recent call last):
File "main.py", line 7, in <module>
emp(name="Anand")
TypeError: emp() missing 1 required positional argument: 'age'

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.

```
defemp( name, age = 35 ):
print("Name: ", name)
print( "Age: ", age)


emp( age=50, name="arun" ) #Function call11
emp( name="Anand" ,35) #Function call2
```

Output

```
emp( name="Anand" ,35) #Function call2
                ^
SyntaxError: non-keyword arg after keyword arg
```

In the function call2 the second argument age can not be identified with its name.

Variable number of arguments

This is very useful when we do not know the exact number of arguments that will be passed to a function.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. n asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments.

```
# Function definition is here
defprintinfo( arg1,*vartuple):
"This prints a variable passed arguments"
print"Output : "
print arg1
forvarinvartuple:
printvar
return;
# Now you can call printinfo function
printinfo( 5 )
printinfo(12, 50, 20)
```

Output

Output:
10
Output:

70
60
50

Python Anonymous/Lambda Function

In Python, anonymous function is a function that is defined without a name.

While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.
Hence, anonymous functions are also called lambda functions.

Syntax of Lambda Function in python

lambda arguments: expression

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.
Here is an example of lambda function that doubles the input value.

Eg1:

```
# Program to show the use of lambda functions
double = lambda x: x * 2

# Output: 10
print(double(5))
```

Eg2:

```
# Program to show the use of lambda functions
sum = lambda x,y,z: x+y+z
# Output: 30
print(sum(5,10,15))
```

Filter Function
In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as `arguments`). Lambda functions are used along with built-in functions like filter(), map() etc.

```
list1 = [1, 5, 4, 6, 8, 11, 3, 12]

list2 = list(filter(lambda x: (x%2 == 0) , list1))

# Output: [4, 6, 8, 12]
print(list2)
```

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.
Here is an example use of map() function to double all the items in a list.

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

# Output: [2, 10, 8, 12, 16, 22, 6, 24]
print(new_list)
```

Use of lambda() with reduce()
The reduce() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new reduced result is returned. This performs a repetitive operation over the pairs of the list.

```
fromfunctools import reduce
list1 = [1, 5, 4, 6, 8, 11, 3, 12]

list2=reduce(lambda x,y:x+y,list1)
print(list2)
```

UNIT – II DATA STRUCTURES AND PACKAGES

Strings –List – Tuples – Dictionaries–Sets – Exception Handling: Built-in Exceptions – User-defined exception– Modules and Packages.

Strings

The string can be defined as the sequence of characters represented in the quotation marks. In python, we can use single, double, or triple quotes to define a string.

String handling in python is a straightforward task since there are various inbuilt functions and operators provided.In the case of string handling, the operator + is used to concatenate two strings as the operation *"hello"+" python"* returns *"hello python"*. Strings in Python are immutable. You can't change a string in-place, but you can copy parts of strings to another string to get the same effect.

The operator * is known as repetition operator as the operation "Python " *2 returns "Python Python ".

A substring (a part of a string) froma string extracted by using a slice.A slice is defined by using square brackets, a start offset, an end offset, and an optional step size. Some of these can be omitted. The slice will include characters from offset start to one before end.

1. [:] extracts the entire sequence from start to end.
2. [start :] specifies from the start offset to the end.
3. [: end] specifies from the beginning to the end offset minus 1.
4. [start : end] indicates from the start offset to the end offset minus 1.
5. [start : end : step] extracts from the start offset to the end offset minus 1, skipping characters by step.

Example

```
str1 = 'Good Morning' #string str1
str2 = ' how are you' #string str2
print (str1[0:2]) #printing first two character using slice operator
print (str1[3]) #printing 3rd character of the string
print (str1*2) #printing the string twice
print (str1 + str2) #printing the concatenation of str1 and str2
```

output

```
Go
d
Good MorningGood Morning
Good Morning how are you
```

Strings Built-in Functions

```
name="Welcome Good How Morning How Are You"
name1="     hai   "
print(name1.lstrip(' '))
print(name1.rstrip(' '))
print(name1.strip(' '))
print(max(name1))
print(name[4:-5])
print(name.count("how",0,len(name)))
print(name.endswith("you",0,len(name)))
print(name.find("how",0,len(name)))
print(name.isalnum())
print(name.islower())
print(name.upper())
print(name.istitle())
s="-"
l=["a","b","c"]
print(s.join(l))
print(max(name))
print(name.swapcase())
print(name.title())
print(name.replace("Good","bad"))
print(name.split(" "))
```

output

```
hai
hai
hai
i
ome Good How Morning How Ar
0
False
-1
False
False
WELCOME GOOD HOW MORNING HOW ARE YOU
True
a-b-c
w
wELCOMEgOODhOWmORNINGhOWaREyOU
Welcome Good How Morning How Are You
Welcome bad How Morning How Are You
['Welcome', 'Good', 'How', 'Morning', 'How', 'Are', 'You']
```

Lists

Lists are positionally ordered collections of arbitrarily typed objects, and they have no fixed size. They are also mutable—unlike strings, lists can be modified in place by assignment to offsets as well as a variety of list method calls.A list is created by placing all the items (elements) inside a square bracket [ ], separated by commas. The same value can occur more than once in a list. Empty list is created with list = []

```
>>>a_list = ['a', 'b', 'mpilgrim', 'z', 'example']
>>>a_list
['a', 'b', 'mpilgrim', 'z', 'example']
>>>a_list[0]
 'a'
>>>a_list[4]
 'example'
>>>a_list[-1]
'example'
>>>a_list[-3]
'mpilgrim'
```

The above code works in the following manner

1. First, define a list of five items. Note that they retain their original order.
2. A list can be used like a zero-based array. The first item of any non-empty list is always a_list[0].
3. The last item of this five-item list is a_list[4], because lists are always zero-based.
4. A negative index accesses items from the end of the list counting backwards. The last item of any non-empty list is always a_list[-1].
5. If the negative index is confusing to you, think of it this way: a_list[-n] == a_list[len(a_list) - n]. So in this list, a_list[-3] == a_list[5 - 3] == a_list[2].

Slicing a List

A range of items in a list can be accesses by using the slicing operator (:). Part of a list, called a “slice”is obtained by specifying two indices. The return value is a new list containing all the items of the list, in order, starting with the first slice index (in this case a_list[1]), up to but not including the second slice index (in this case a_list[3]).

```
>>>a_list
['a', 'b', 'mpilgrim', 'z', 'example']
>>>a_list[1:3]
['b', 'mpilgrim']
>>>a_list[1:-1]
['b', 'mpilgrim', 'z']
>>>a_list[0:3]
['a', 'b', 'mpilgrim']
>>>a_list[:3]
```

['a', 'b', 'mpilgrim']
>>>a_list[3:]
['z', 'example']
>>>a_list[:]
['a', 'b', 'mpilgrim', 'z', 'example']

Adding Items to a List

There are four ways to add items to a list.

1. The + operator concatenates lists to create a new list. A list can contain any number of items; there is no size limit (other than available memory).
2. The append() method adds a single item to the end of the list.
4. Lists are implemented as classes. "Creating" a list is really instantiating a class. As such, a list has methods that operate on it. The extend() method takes one argument, a list, and appends each of the items of the argument to the original list.
5. The insert() method inserts a single item into a list. The first argument is the index of the first item in the list that will get bumped out of position. List items do not need to be unique; for example, there are now two separate items with the value 'Ω': the first item, a_list[0], and the last item, a_list[6].

>>>a_list = ['a']
>>>a_list = a_list + [2.0, 3]
>>>a_list
['a', 2.0, 3]
>>>a_list.append(True)
>>>a_list
['a', 2.0, 3, True]
>>>a_list.extend(['four', 'Ω'])
>>>a_list
['a', 2.0, 3, True, 'four', 'Ω']
a_list.insert(0, 'Ω')
>>>a_list
['Ω', 'a', 2.0, 3, True, 'four', 'Ω']

Append: Adds its argument as a single element to the end of a list. The length of the list increases by one.
Extend(): Iterates over its argument and adding each element to the list and extending the list. The length of the list increases by number of elements in it's argument

Removing Items from a List

Lists can expand and contract automatically. There are several different ways to remove items from a list. The elements of a list is removed by using del function. Elements can also removed from the List by using built-in remove() function but an Error arises if element doesn't exist in the set. Remove() method only removes one element at a time, to remove range of elements, iterator

is used. Pop() function can also be used to remove and return an element from the set, but by default it removes only the last element of the set, to remove element from a specific position of the List, index of the element is passed as an argument to the pop() method.

Method 1 : Using del statement

```
>>>a_list = ['a', 'b', 'new', 'mpilgrim', 'new']
>>>a_list[1]
'b'
>>>dela_list[1]
>>>a_list
['a', 'new', 'mpilgrim', 'new']
>>>a_list[1]
'new'
```

Method 2 : Using remove() function

```
>>>a_list.remove('new')
>>>a_list
['a', 'mpilgrim', 'new']
>>>a_list.remove('new')
>>>a_list
['a', 'mpilgrim']
```

Method 2 : Using pop() function

When called without arguments, the pop() list method removes the last item in the list and returns the value it remove.

```
>>>a_list = ['a', 'b', 'new', 'mpilgrim']
>>> a_list.pop()
'mpilgrim'
>>>a_list
['a', 'b', 'new']
a_list.pop(1) ② 'b'
>>>a_list
['a', 'new']
>>> a_list.pop()
'new'
>>> a_list.pop()
'a'
```

Python List Built-in Methods

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
Pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

Python List Built-in functions

Python provides the following built-in functions which can be used with the lists.

Function	Description
cmp(list1, list2)	It compares the elements of both the lists.
len(list)	It is used to calculate the length of the list.
max(list)	It returns the maximum element of the list.
min(list)	It returns the minimum element of the list.
list(seq)	It converts any sequence to the list.

Tuples

A tuple is an immutable list. A tuple can not be changed in any way once it is created. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets. Creating a tuple is as simple as putting different comma-separated values.
Lists have methods like append(), extend(), insert(), remove(), and pop(). Tuples have none of these methods. Tuples are faster than lists. The operators like concatenation (+), repetition (*), Membership (in) works in the same way as they work with the list. We can store list inside tuple or tuple inside the list up to any number of level.Tuples use less space. Tuples can be used as a dictionary keys.Empty list is created with list = ().

Creating a tuple with one element is a bit tricky.Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is, in fact, a tuple.

```
# Creating a tuple having one element
my_tuple = ("hello",)
print(type(my_tuple))  # <class 'tuple'>
```

```
>>>a_tuple = ("a", "b", "mpilgrim", "z", "example")

>>>a_tuple

('a', 'b', 'mpilgrim', 'z', 'example')
```

Access Tuple Elements

There are various ways in which we can access the elements of a tuple.

1. Indexing

We can use the index operator [] to access an item in a tuple where the index starts from 0.So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an element outside of tuple (for example, 6, 7,...) will raise an IndexError.The index must be an integer; so we cannot use float or other types. This will result in TypeError.

```
my_tuple = ('p','e','r','m','i','t')

print(my_tuple[0])   # 'p'

print(my_tuple[5])   # 't'
```

2. Negative Indexing

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

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

Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple.

But deleting a tuple entirely is possible using the keyword del.

```
my_tuple = ('p','r','o','g','r','a','m','i','z')

# can't delete items
# TypeError: 'tuple' object doesn't support item deletion
# delmy_tuple[3]

# Can delete an entire tuple
delmy_tuple

# NameError: name 'my_tuple' is not defined
print(my_tuple)
```

Other Tuple Operations

1. Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword in.

```
my_tuple = ('a','p','p','l','e',)

# In operation
# Output: True
print('a' in my_tuple)

# Output: False
print('b' in my_tuple)

# Not in operation
# Output: True
print('g' not in my_tuple)
```

2. Iterating Through a Tuple

Using a for loop we can iterate through each item in a tuple.

```
# Output:
# Hello John
# Hello Kate
for name in ('John','Kate'):
print("Hello",name)
```

The following are true about tuples

1. You can’t add elements to a tuple. Tuples have no append() or extend() method.

2. You can’t remove elements from a tuple. Tuples have no remove() or pop() method. To explicitly remove an entire tuple, del statement is used.

3. You can find elements in a tuple, since this doesn’t change the tuple.

4. You can also use the in operator to check if an element exists in the tuple.

Comparison between lists and tuples

List	Tuple
The literal syntax of list is shown by the [].	The literal syntax of the tuple is shown by the ().
The List is mutable.	The tuple is immutable.
The List has the variable length.	The tuple has the fixed length.
The list provides more functionality than tuple.	The tuple provides less functionality than the list.
The list Is used in the scenario in which we need to store the simple collections with no constraints where the value of the items can be changed.	The tuple is used in the cases where we need to store the read-only collections i.e., the value of the items can not be changed. It can be used as the key inside the dictionary.

Sets

set is a collection which is unordered and unindexed. In Python sets are written with curly brackets. The elements of the set can not be duplicate. Unlike other collections in python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of

the set by the index. However, we can print them all together or we can get the list of elements by looping through the set.

The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

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

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

Creating Set

Creating an empty set is a bit tricky. 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.

```
# initialize a with {}
a = {}

# check data type of a
# Output: <class 'dict'>
print(type(a))

# initialize a with set()
a = set()

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

Access Items

You cannot access items in a set by referring to an index, since sets are unordered the items has no index.

But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

Example:

```
thisset={"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

Adding or modifying the items to the set

Once a set is created, you cannot change its items, but you can add new items.Single element can be added using the add() method and multiple elements can be added using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

Example for add()

```
thisset={"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

Example for update()

```
thisset={"apple", "banana", "cherry"}

thisset.update(["orange", "mango", "grapes"])

print(thisset)
```

Removing Items

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. Similarly, an item can be removed and returned using the `pop()` method. All items from a set is removed using clear().The del keyword will delete the set completely.

Example

```
# initialize my_set
my_set = {1, 3, 4, 5, 6}
print(my_set)

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

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

```
# discard an element
# not present in my_set
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)
```

Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. This can be do with operators or methods.Union is performed using | operator or the method union().Union of A and B is a set of all elements from both sets.Intersection is performed using & operator or the method intersection().Intersection of A and B is a set of elements that are common in both sets.Difference is performed using - operator or method difference(). Difference of A and B (A - B) is a set of elements that are only in A but not in B. Similarly, B - A is a set of element in B but not in A.Symmetric difference is performed using ^ operator or method symmetric_difference().Symmetric Difference of A and B is a set of elements in both A and B except those that are common in both.

Example:

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
print(A|B)
print(A&B)
print(A-B)
print(A^B)
```

Output

```
{1, 2, 3, 4, 5, 6, 7, 8}
{4, 5}
{1, 2, 3}
{1, 2, 3, 6, 7, 8}
```

Python Frozenset

Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets.

Dictionaries

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds key:value pair. Key value is provided in the dictionary to make it more optimized. Each key-value pair in a Dictionary is separated by a colon :, whereas each key is separated by a ‘comma’. A Dictionary in Python works similar to the Dictionary in a real world. Keys of a Dictionary must be unique and of *immutable* data type such as Strings, Integers and tuples, but the key-values can be repeated and be of any type.

Creating a Dictionary

In Python, a Dictionary can be created by placing sequence of elements within curly **{}** braces, separated by ‘comma’. Dictionary holds a pair of values, one being the Key and the other corresponding pair element being its **Key:value**. Values in a dictionary can be of any datatype and can be duplicated, whereas keys can’t be repeated and must be *immutable*.
Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing to curly braces{}.

```
# Creating an empty Dictionary
Dict={}
print("Empty Dictionary: ")
print(Dict)

# Creating a Dictionary
# with Integer Keys
Dict={1: 'Arun', 2: 'Deepa', 3: 'Kalai'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

# Creating a Dictionary
# with Mixed keys
Dict={'Name': 'Arun', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)

# Creating a Dictionary
# withdict() method
Dict=dict({1: 'Arun', 2: 'Kalai', 3:'Deepa'})
print("\nDictionary with the use of dict(): ")
print(Dict)

# Creating a Dictionary
# with each item as a Pair
Dict=dict([(1, 'Arun'), (2, 'Deepa')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Accessing elements from a Dictionary

In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets.There is also a method called get() that will also help in accessing the element from a dictionary.

```
Dict = {1: 'arun', 'name': 'kalai', 3: 'kumar'}
```

```
# accessing a element using key
print("Acessing a element using key:")
print(Dict['name'])

# accessing a element using key
print("Acessing a element using key:")
print(Dict[1])

# accessing a element using get()
# method
print("Acessing a element using get:")
print(Dict.get(3))
```

Removing Elements from Dictionary

In Python Dictionary, deletion of keys can be done by using the del keyword. Using del keyword, specific values from a dictionary as well as whole dictionary can be deleted. Other functions like pop() and popitem() can also be used for deleting specific values and arbitrary values from a Dictionary. All the items from a dictionary can be deleted at once by using clear() method.

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

Dictionary Methods

METHODS	DESCRIPTION

copy()	They copy() method returns a shallow copy of the dictionary.
clear()	The clear() method removes all items from the dictionary.
pop()	Removes and returns an element from a dictionary having the given key.
popitem()	Removes the arbitrary key-value pair from the dictionary and returns it as tuple.
get()	It is a conventional method to access a value for a key.
dictionary_name.values()	returns a list of all the values available in a given dictionary.
str()	Produces a printable string representation of a dictionary.
update()	Adds dictionary dict2's key-values pairs to dict
setdefault()	Set dict[key]=default if key is not already in dict
keys()	Returns list of dictionary dict's keys
items()	Returns a list of dict's (key, value) tuple pairs
has_key()	Returns true if key in dictionary dict, false otherwise
fromkeys()	Create a new dictionary with keys from seq and values set to value.
type()	Returns the type of the passed variable.
cmp()	Compares elements of both dict.

Python Exceptions

An exception can be defined as an abnormal condition in a program resulting in the disruption in the flow of the program.
Whenever an exception occurs, the program halts the execution, and thus the further code is not executed. Therefore, an exception is the error which python script is unable to tackle with.
Python provides us with the way to handle the Exception so that the other part of the code can be executed without any disruption. However, if we do not handle the exception, the interpreter doesn't execute all the code that exists after the that.

Errors can also occur at runtime and these are called exceptions. They occur, for example, when a file we try to open does not exist (FileNotFoundError), dividing a number by zero (ZeroDivisionError), module we try to import is not found (ImportError) etc.

Whenever these type of runtime error occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

Illegal operations can raise exceptions. There are plenty of built-in exceptions in Python that are raised when corresponding errors occur.

Python Built-in Exceptions

Exception	Cause of Error
AssertionError	Raised when assert statement fails.
AttributeError	Raised when attribute assignment or reference fails.
EOFError	Raised when the input() functions hits end-of-file condition.
FloatingPointError	Raised when a floating point operation fails.
GeneratorExit	Raise when a generator's close() method is called.
ImportError	Raised when the imported module is not found.
IndexError	Raised when index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
KeyboardInterrupt	Raised when the user hits interrupt key (Ctrl+c or delete).
MemoryError	Raised when an operation runs out of memory.
NameError	Raised when a variable is not found in local or global scope.
NotImplementedError	Raised by abstract methods.
OSError	Raised when system operation causes system related error.
OverflowError	Raised when result of an arithmetic operation is too large to be represented.
ReferenceError	Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError	Raised when an error does not fall under any other category.
StopIteration	Raised by next() function to indicate that there is no further item to be returned by iterator.

SyntaxError	Raised by parser when syntax error is encountered.
IndentationError	Raised when there is incorrect indentation.
TabError	Raised when indentation consists of inconsistent tabs and spaces.
SystemError	Raised when interpreter detects internal error.
SystemExit	Raised by sys.exit() function.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError	Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError	Raised when a Unicode-related error occurs during encoding.
UnicodeDecodeError	Raised when a Unicode-related error occurs during decoding.
UnicodeTranslateError	Raised when a Unicode-related error occurs during translating.
ValueError	Raised when a function gets argument of correct type but improper value.
ZeroDivisionError	Raised when second operand of division or modulo operation is zero.

The try and except Block: Handling Exceptions

The try and except block in Python is used to catch and handle exceptions. Python executes code following the try statement as a “normal” part of the program. The code that follows the except statement is the program’s response to any exceptions in the preceding try clause. In try block you can write the code which is suspicious to raise an exception, and in except block, you can write the code which will handle this exception.

Syntax

```
try:
    #block of code
```

```
except Exception1:
   #block of code

except Exception2:
   #block of code

#other code
```

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.A try clause can have any number of except clause to handle them differently, but only one will be executed in case an exception occurs.

Syntax

```
try:
   #block of code

except Exception1:
   #block of code

else:
   #this code executes if no except block is executed
```

Declaring multiple exceptions

The python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions. This can be achieved by writing names of exception classes in except clause seperated by comma.

Syntax

```
try:
   #block of code

except (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)
   #block of code

else:
   #block of code
```

try-finally clause

The try statement in Python can have an optional finally clause. In case if there is any code which you want to be executed, whether exception occurs or not, then that code can be placed inside the finally block. When an exception occurs, the control immediately goes to finally block and all

the lines in finally block gets executed first. After that the control goes to except block to handle exception.

Syntax

```
try:
    #block of code

except (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)
    #block of code

else:
    #block of code
finally:
    # block of code
    # this will always be executed
```

Raising exceptions

An exception can be raised by using the raise clause in python. The syntax to use the raise statement is given below.

syntax

```
raise Exception_class,<value>
```

Example

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    if b is 0:
        raise ArithmeticError;
    else:
        print("a/b = ",a/b)
except ArithmeticError:
    print("The value of b can't be 0")
```

Output:

```
Enter a: 10
Enter b: 0
The value of b can't be 0
```

User-Defined Exceptions

It is possible to define our own exception in Python. The built-in and user-defined exceptions in Python using try, except and finally statements.

In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from Exception class. Most of the built-in exceptions are also derived from this class.

```
# define Python user-defined exceptions
class Error(Exception):
  """Base class for other exceptions"""
pass

classValueTooSmallError(Error):
  """Raised when the input value is too small"""
pass

classValueTooLargeError(Error):
  """Raised when the input value is too large"""
pass

# our main program
# user guesses a number until he/she gets it right

# you need to guess this number
number = 10

while True:
try:
i_num = int(input("Enter a number: "))
ifi_num< number:
raiseValueTooSmallError
elifi_num> number:
raiseValueTooLargeError
break
exceptValueTooSmallError:
print("This value is too small, try again!")
print()
```

```
exceptValueTooLargeError:
print("This value is too large, try again!")
print()

print("Congratulations! You guessed it correctly.")
```

Example

```
classCustomException(Exception):

pass

classValueNegativeError(CustomException):

pass

classValueTooLargeError(CustomException):

pass

try:

i_num = int(input("Enter a number: "))

ifi_num< 0:

raiseValueNegativeError

elifi_num> 100:

raiseValueTooLargeError

exceptValueNegativeError:

print("This value is negative enter positive number")

print()

exceptValueTooLargeError:

print("This value is too large, try again!")

print()

print("Congratulations! You guessed it correctly.")
```

Python Modules
A module is a file containing Python definitions and statements. A module can define functions, classes and variables. A module can also include runnable code. Python modules are .py files that consist of Python code. Any Python file can be referenced as a module.Grouping related code into a module makes the code easier to understand and use.

A simple module - calc.py

```
def add(x, y):
   return (x+y)

def subtract(x, y):
   return (x-y)
```

The import statement
We can use any Python source file as a module by executing an import statement in some other Python source file.

The import has the following syntax

import module1[, module2[,... moduleN]

Python's from statement lets you import specific attributes from a module into the current namespace. The from...import has the following syntax

from modname import name1[, name2[, ... nameN]]

It is also possible to import all the names from a module into the current namespace by using the following import statement −

*from modname import **

This provides an easy way to import all the items from a module into the current namespace. When interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches for importing a module. For example, to import the module calc.py, we need to put the following command at the top of the script :

main.py

```
from cal import subtract
import calc
print calc.add(10, 2)
print subtract(78,45)
```

Built-in Modules
There are several built-in modules in Python, which you can import whenever you like.

Math module

Python Math module provides access to the mathematical functions. These include trigonometric functions, representation functions, logarithmic functions, angle conversion functions, etc. So, we can do many complex mathematical operations with the help of the Python Math functions

eg

```
import math
number = -2.34
print('The given number is :', number)
print('Floor value is :', math.floor(number))
print('Ceiling value is :', math.ceil(number))
print('Absolute value is :', math.fabs(number))
```

Random module

The random module gives access to various useful functions and one of them being able to generate random numbers, which is *randint()*.***randint()*** is an inbuilt function of the *random module* in Python

Syntax :

```
randint(start, end)
```

```
Python3 program explaining work
# of randint() function

# imports random module
import random

# Generates a random number between
# a given positive range
r1 = random.randint(0, 10)
print("Random number between 0 and 10 is % s" % (r1))

# Generates a random number between
# two given negative range
r2 = random.randint(-10, -1)
print("Random number between -10 and -1 is % d" % (r2))

# Generates a random number between
# a positive and a negative range
r3 = random.randint(-5, 5)
print("Random number between -5 and 5 is % d" % (r3))
```

Packages in Python

Packages are namespaces which contain multiple packages and modules themselves. A package is a collection of Python modules, i.e., a package is a directory of Python modules containing an additional __init__.py file. The __init__.py distinguishes a package from a directory

that just happens to contain a bunch of Python scripts. Packages can be nested to any depth, provided that the corresponding directories contain their own __init__.py file.

When you import a module or a package, the corresponding object created by Python is always of type module. This means that the distinction between module and package is just at the file system level. Note, however, when you import a package, only variables/functions/classes in the __init__.py file of that package are directly visible, not sub-packages or modules.

Packages allow for a hierarchical structuring of the module namespace using **dot notation**. In the same way that **modules** help avoid collisions between global variable names, **packages** help avoid collisions between module names. Packages can contain nested subpackages to arbitrary depth.

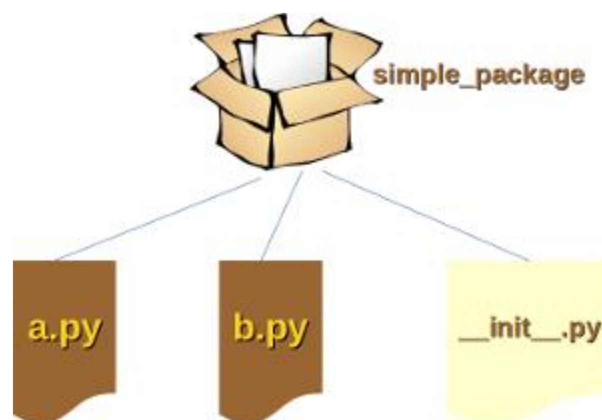
__init__.py

The package folder contains a special file called __init__.py, which stores the package's content. It serves two purposes:

1. The Python interpreter recognizes a folder as the package if it contains __init__.py file.
2. __init__.py exposes specified resources from its modules to be imported.

An empty __init__.py file makes all functions from above modules available when this package is imported. Note that __init__.py is essential for the folder to be recognized by Python as a package.

The __init__.py file is normally kept empty. However, it can also be used to choose specific functions from modules in the package folder and make them available for import.



Steps to Create a Python Package

1. Create a directory and give it your package's name.
2. Put your functions in it.
3. Create a __init__.py file in the directory

importing packages

```
from <package_name> import *
from<package_name>import<modules_name>[,<module_name>...]
from<package_name>import<module_name>as<alt_name>
```

Class

```
class Student:
noofstudents = 0
  name=""
age=0

def __init__(self, name, age):
    self.name = name
self.age = age
Student.noofstudents += 1



defdisplaydetails(self):
print( "Name : ", self.name,  ", Age: ", self.age)


s1 = Student("Jothi", 37)

s2 = Student("Senthil", 35)
s3= Student("Kalai",39)
s1.displaydetails()
s2.displaydetails()
s3.displaydetails()
print("Total Number of Students %d" % Student.noofstudents)
```