GE8151- PROBLEM SOLVING AND PYTHON PROGRAMMING

UNIT II- DATA, EXPRESSIONS, STATEMENTS

Interactive mode and script mode

- Python has two basic modes: interactive and script mode

Interactive mode:

- Interactive mode is a command line shell which gives immediate feedback for each statement, while running previously fed statements in active memory.

Example	Output
>>>x = 5 >>>x + 1	6

Script mode

- Script mode is the normal mode where the scripted and finished **.py** files are run in the Python interpreter.

Example	Output
x=5 x=x+1 print(x)	6

Comments

- Comments are the non-executable statements explain what the program does.
- Python supports two types of comments:

1) Single lined comment:

- User wants to specify a single line comment, then comment must start with **#**

Example:

```
# This is single line comment
```

2) Multi lined Comment:

- Multi lined comment can be given inside triple quotes(' ' ').

Example:

```
''' This
 Is
 Multipline comment'''
```

Python Input and Output Statements

Python Input Statements

- Python provides two built-in functions to read a line of text from standard input device, keyboard.
- These function are:

1. input( )

- Interprets and evaluates the input ie., if user enters integer input, an integer will be returned.

2. raw_input()

- raw_input( ) takes exactly what user typed and passes it back as string.
- It doesn't interpret the user input. Even an integer value of 10 is of string only.

Python Output Statements

- print( ) function is used to output data to the standard output device(screen).
- print statement is used where zero or more expressions are passed separated by commas.

Example Program	Output
n= **input**("enter the number") print("simple input statement=",n+10) n=**raw_input**("enter the number") print("simple raw_input statement=",n+10)	enter the number 5 simple input statement= 15 enter the number 5 "simple raw_input statement",n+10

Expressions

Definition:

- An **expression** is a combination of values, variables, and operators.
- A value and a variable, itself considered as an expression

Example:

```
17          #expression
x           #expression
x + 17      #expression
```

Statements

Definition:

- A **statement** is a unit of code that the Python interpreter can execute.
- We have seen two kinds of statement: print and assignment.
- Python allows the use of line continuation character (\ ) to denote that the line should continue.

Example:

```
Total = mark1+\
        mark2+\
        mark3
```

- Technically an expression is also a statement, but it is probably simpler to think of them as different things.
- The important difference is that an expression has a value; a statement does not.

Variables

Definition:

- Variable is an identifiers that refer to a values.
- Variable is a name of the memory location where data is stored.
- Once a variable is stored that means a space is allocated in memory.

Example:

```
a=10
pi=3.14
name='cse'
```

Where a, pi and name are **Variables**
10, 3.14 and cse are **Values**

Assigning values to Variable:

- When we assign any value to the variable that variable is declared automatically.
- The assignment is done using the equal (=) operator.

Multiple Assignments:

- Multiple assignments can be done in Python at a time.
- There are two ways to assign values in Python:

1. Assigning single value to multiple variables:

Example:	Output:
x=y=z=50 print x print y print z	50 50 50

2. Assigning multiple values to multiple variables:

Example:	Output:
a,b,c=5,10,15 print a print b print c	5 10 15

Tokens

Definition:

- Token is the smallest unit inside the given program.
- Tokens can be defined as a punctuator mark, reserved words and each individual word in a statement.

There are following tokens in Python:

- Keywords
- Identifiers
- Literals
- Operators

Keywords

- Keywords are special reserved words which convey a special meaning to the compiler/interpreter.
- Each keyword have a special meaning and a specific operation.
- List of Keywords used in Python are:

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

Identifiers

- Identifiers are the names given to the fundamental building blocks in a program.
- These can be **variables**, **class**, **object**, **functions**, **lists**, **dictionaries** etc.
- There are certain rules defined for naming i.e., Identifiers.
  1. An identifier is a long sequence of characters and numbers.
  2. Keyword should not be used as an identifier name.
  3. Python is case sensitive. So using case is significant.
  4. First character of an identifier can be character, underscore (_) but not digit.

Literals (Values and Types)

- Literals can be defined as a data that is given in a variable or constant.
- A value is one of the basic things a program works with, like a letter or a number
- Python support the following **literals or standard data types**

a) Numbers

b) String

c) Boolean → True and False

d) List → [ ] – empty list

e) Tuple → ( ) – empty tuple

f) Dictionary →{ } – empty dictionary

a) Numeric (Number data types) literals

Definition

- Numeric Literals are immutable.
- Numeric literals can belong to following four different numerical types.

Numerical Data types	Description	Example
int	Signed integers	100
long	Long integers	ox1234 and 5678L
float	Floating point	3.14
complex	Complex number	5+78j

b) String literals

Definition

- Strings are the **group (sequence or collection)** of characters, digits, and symbols enclosed within quotation marks (‘ or “) .

Example:

"cse" , 'mech', '12345'

Types of Strings:

- There are two types of Strings supported in Python:

a) Single line String → Strings that are terminated within a single line are known as Single line Strings.

Example:

text1='hello'

b) Multi line String → A piece of text that is spread along multiple lines is known as Multiple line String.

- There are two ways to create Multiline Strings:

1) Adding back slash at the end of each line:

Example:	**Output**
str1='cse\ mech'	>>> print (str1) csemech

2) Using triple quotation marks:

Example:	**Output**
>>> **str2**= """welcome to first year students"""	>>> print (str2) welcome to first year students

c) Boolean literals

- Boolean is one more data type supported in python.
- It takes the two values: True or False.

Example:

```
print True        # True is a Boolean value.
print False       # False is a Boolean value.
```

d) List

- List contains items of different data types. Lists are mutable i.e., modifiable.
- The values stored in List are separated by commas(,) and enclosed within a square brackets([ ]). We can store different type of data in a List.
- Value stored in a List can be retrieved using the slice operator([ ] and [:]).
- The plus sign (+) is the list concatenation and asterisk(*) is the repetition operator.

Example	Output
>>> list1=[10,30,50,70,90] >>> list2=[20,40,60,80,100]	
	>>> list1 [10, 30, 50, 70, 90]
	>>>list2 [20, 40, 60, 80, 100]
	>>>list1[0:3] [10, 30, 50]
	>>> list1[3:] [70, 90]
	>>> list1+list2 [10, 30, 50, 70, 90, 20, 40, 60, 80, 100]
	>>> list1*2 [10, 30, 50, 70, 90, **10, 30, 50, 70, 90**]

e) Tuples

- Tuple is another form of collection where different type of data can be stored.
- It is similar to list where data is separated by commas.

- The main difference between is **lists** and **tuples** are:
    - Lists are enclosed in square bracket([ ]) and their elements and size can be changed,
    - Tuples are enclosed in parenthesis (( )) and their elements and size cannot be changed.

Example	**Output**
>>> tuple1=('cse',10,20.5,'mech')	>>> tuple1 ('cse', 10, 20.5, 'mech')
>>> tuple2=('eee',30)	>>> tuple2 ('eee', 30)
	>>> tuple1+tuple2 ('cse', 10, 20.5, 'mech', 'eee', 30)
	>>>tuple2[0] 'eee'
	>>>tuple1[2:] (20.5, 'mech')
	>>> tuple1[:2] ('cse', 10)

f) Dictionary

- Dictionary is a collection which works on a key-value pair.
- It works like an associated array where no two keys can be same.
- Dictionaries are enclosed by curly braces ({}) and **values** and **keys** can be retrieved by square bracket (**[ ]**).

Example:
>>> dictionary={'name':'raja','rollno':100,'dept':'cse'}
>>> dictionary {'rollno': 100, 'dept': 'cse', 'name': 'raja'}
>>> dictionary.keys() ['rollno', 'dept', 'name']
>>> dictionary.values() [100, 'cse', 'raja']

Operators

Definition:

- An **operator** is a symbol that specifies an operation to be performed on the operands.
- The values are known as Operands.
- An **Operands** is data item.

Example: 1

a+b

Where '+' is operator and 'a','b' are the operands.

Example: 2

4 + 5 = 9

Here 4 and 5 are Operands and (+) , (=) signs are the operators. They produce the output 9.

Types of Operators:

1. Arithmetic Operators.
2. Relational Operators.
3. Assignment Operators.
4. Logical Operators.
5. Membership Operators.
6. Identity Operators.
7. Bitwise Operators.

1. Arithmetic Operators:

Operators	Description	Example
+	Addition	10 + 5 = 15
-	Subtraction	10 – 5 = 5
*	Multiplication	10 * 5 = 50
/	Division	10 / 5 = 2.0
//	Floor division	10 // 5 = 2
%	Modulus division	10 % 5 = 0
**	Exponent(raise to power)	10 ** 5 = 10000

Example Program:

#**Arithmetic operators**

```
a=int(input("enter the a value"))
b=int(input("enter the b value"))
c=a+b
print("Addition is:",c)
c=a-b
print("Subtraction is:",c)
c=a*b
print("Multiplication is:",c)
c=a/b
print("Division is:",c)
c=a//b
print("Floor Division is:",c)
c=a%b
print("Mod Division is:",c)
c=a**b
print("Power is:",c)
```

2. Relational Operators:

Operators	Description	Example
<	Less than	5<10
>	Greater than	10>5
<=	Less than or equal to	5<=10
>=	Greater than or equal to	10>=5
==	Equal to	10==10
!=	Not equal to	5!=10
<>	Not equal to(similar to !=)	5<>10

Example Program:

 # Positive or Zero or Negative number

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

3. Assignment Operators:

Operators	Description	Example	Explanation
=	Assignment	a = 10	a =10
+=	Add and assign	a += b	a =a+b
-=	Subtract and Assign	a -= b	a =a-b
*=	Multiply and assign	a *= b	a =a*b
/=	Divide and Assign	a /= b	a =a/b
//=	Floor division and assign	a //= b	a =a//b
%=	Modulus and assign	a %= b	a =a%b
**=	Exponent and assign	a **= b	a =a**b

Example Program:

 # Sum of Digit

```
n=int(input("Enter a number "))
sum = 0
while (n>0):
    r = n%10
    sum = sum + r
    n = n //10
print("Sum of Digit is ",sum)
```

4. Logical Operators:

Operators	Description	Example
and	Logical AND(When both conditions are true output will be true)	(5>4) and (3>2)
or	Logical OR (If any one condition is true output will be true)	(5>4) or (3<2)
not	Logical NOT(Compliment the condition i.e., reverse)	not(5>4)

Example Program:	Output:
a=(5>4) **and** (3>2) print (a) b=(5>4) **or** (3<2) print (b) c=**not**(5>4) print (c)	True True False

5. Membership Operators:

Operators	Description	Example

in	**Returns true** if a variable is **in** sequence of another variable, **else false**.	a=10
		list=[10,20,30,40,50];
		if (a **in** list):

not in	**Returns true** if a variable is **not in** sequence of another variable, **else false**.	b=80
		list=[10,20,30,40,50];
		if(b **not in** list):

Example Program:	Output:

a=10
b=80
list=[10,20,30,40,50];
if (a **in** list):
  print ("a is in given list")
else:
  print ("a is not in given list")
if(b **not in** list):
  print ("b is not given in list")
else:
  print ("b is given in list")
	a is in given list
	
	b is not given in list

6. Identity Operators:

Operators	Description	Example
is	**Returns true** if identity of **two operands are same**, else false	a=20 b=20 if(a **is** b):
is not	**Returns true** if identity of **two operands are not same**, else false.	a=20 b=10 if(a **is not** b):

Example Program:	Output:

a=20
b=20
if(a **is** b):
  print("a,b have same identity")
else:
  print("a, b are different")
b=10
if(a **is not** b):
  print("a,b have different identity")
else:
  print("a,b have same identity")
	a,b have same identity
	a,b have different identity

7. Bitwise Operators.

Operators	Description	Example
&	Bitwise AND	a & b = 0010
|	Bitwise OR	a | b = 0011
^	Bitwise exclusive OR	a ^ b = 0001
~	Bitwise complement	~a = 1101
<<	Shift left	a << 2 = 1000
>>	Shift right	a >> 2 = 0000

Example Program:

```
a = 2
b = 3
print("Bitwise AND Operator  is = ", a & b)
print("Bitwise OR Operator  is = ", a | b)
print("Bitwise EXCLUSIVE OR Operator  is = ", a ^ b)
print("Bitwise NOT Operator  is = ", ~a)
print("Bitwise LEFT SHIFT Operator is = ", a << 2)
print("Bitwise RIGHT SHIFT Operator  is = ", b >> 2)
```

Output:

```
Bitwise AND Operator  is =  2
Bitwise OR Operator  is =  3
Bitwise EXCLUSIVE OR Operator  is =  1
Bitwise NOT Operator  is =  -3
Bitwise LEFT SHIFT Operator is =  8
Bitwise RIGHT SHIFT Operator  is =  0
```

Precedence of operators

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

Operators	Meaning
( )	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisions, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

Example Program	**Output**
a = 20 b = 10 c = 15 d = 5 e = 0 **e = (a + b) * c / d** print ("Result is ", e) **e = ((a + b) * c) / d** print ("Result is ", e) **e = (a + b) * (c / d);** print ("Result is ", e) **e = a + (b * c) / d;** print ("Result is ", e)	Result is 90.0 Result is 90.0 Result is 90.0 Result is 50.0

Functions

Definition:

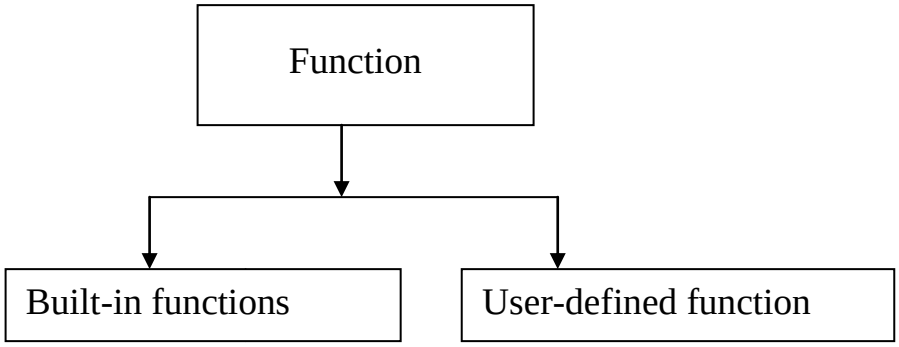
- Function is a group of statements that perform a specific task.
- If a program is large, it is difficult to understand the steps involved in it.
- Hence, it is subdivide into a number of smaller programs called **subprogram** or **functions** or **modules**.
- Each subprogram specifies one or more actions to be performed for the large program.
- Functions may or may not take arguments and may or may not produce results.

Advantage of functions (Why functions?):

- Decomposing large program into smaller functions makes program easy to understand, maintain and debug.
- Functions developed for one program can **reuse** with or without modification when need.
- Reduces program development time and cost.
- It is easy to locate and isolate faulty function.

Types of Functions

- Functions are classified into **two types**



1. **Built-in functions** - Functions that are built into Python.
2. **User-defined function**s - Functions defined by the users themselves.

1. Built-in Function

- The Python interpreter has a number of functions that are **always available for use**. These functions are called built-in functions.
- The user can not modify the function according to their requirements.
- For example,

  input( ), print( )
- They are listed below,

Method	Description
abs ( )	returns absolute value of a number
all( )	returns true when all elements in iterable is true

Method	Description
any( )	Checks if any Element of an Iterable is True
ascii( )	Returns String Containing Printable Representation
bin( )	converts integer to binary string
bool ( )	Coverts a Value to Boolean
bytearray( )	returns array of given byte size
bytes( )	returns immutable bytes object
callable( )	Checks if the Object is Callable
chr( )	Returns a Character (a string) from an Integer
classmethod( )	returns class method for given function
compile( )	Returns a Python code object
complex( )	Creates a Complex Number
delattr( )	Deletes Attribute From the Object
dict( )	Creates a Dictionary
dir( )	Tries to Return Attributes of Object

Method	Description
divmod( )	Returns a Tuple of Quotient and Remainder
enumerate( )	Returns an Enumerate Object
eval( )	Runs Python Code Within Program
exec( )	Executes Dynamically Created Program
filter( )	constructs iterator from elements which are true
float( )	returns floating point number from number, string
format( )	returns formatted representation of a value
frozenset( )	returns immutable frozenset object
getattr( )	returns value of named attribute of an object
globals( )	returns dictionary of current global symbol table
hasattr( )	returns whether object has named attribute
hash( )	returns hash value of an object
help( )	Invokes the built-in Help System
hex( )	Converts to Integer to Hexadecimal

Method	Description
id( )	Returns Identify of an Object
input( )	reads and returns a line of string
int( )	returns integer from a number or string
isinstance( )	Checks if a Object is an Instance of Class
issubclass( )	Checks if a Object is Subclass of a Class
iter( )	returns iterator for an object
len( )	Returns Length of an Object
list( ) Function	creates list in Python
locals( )	returns dictionary of current local symbol table
map( )	Applies Function and Returns a List
max( )	returns largest element
memoryview( )	returns memory view of an argument
min( )	returns smallest element
next( )	Retrieves Next Element from Iterator

Method	Description
object( )	Creates a Featureless Object
oct( )	converts integer to octal
open( )	Returns a File object
ord( )	returns Unicode code point for Unicode character
pow( )	returns x to the power of y
print( )	Prints the Given Object
property( )	returns a property attribute
range( )	return sequence of integers between start and stop
repr( )	returns printable representation of an object
reversed( )	returns reversed iterator of a sequence
round( )	rounds a floating point number to ndigits places.
set( )	returns a Python set
setattr( )	sets value of an attribute of object
slice( )	creates a slice object specified by range()

Method	Description
sorted( )	returns sorted list from a given iterable
staticmethod( )	creates static method from a function
str( )	returns informal representation of an object
sum( )	Add items of an Iterable
super( )	Allow you to Refer Parent Class by super
tuple( ) Function	Creates a Tuple
type( )	Returns Type of an Object
vars( )	Returns __dict__ attribute of a class
zip( )	Returns an Iterator of Tuples
python __import__( )	Advanced Function Called by import

2. User-defined Functions

- The functions defined by the users according to their requirements are called **User-defined Functions.**
- The user can modify the function according to their requirements.
- **Example:**

  swap( ), addition( ), cse( ), mech( )

Advantages of user-defined functions

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmers working on large project can divide the workload by making different functions.

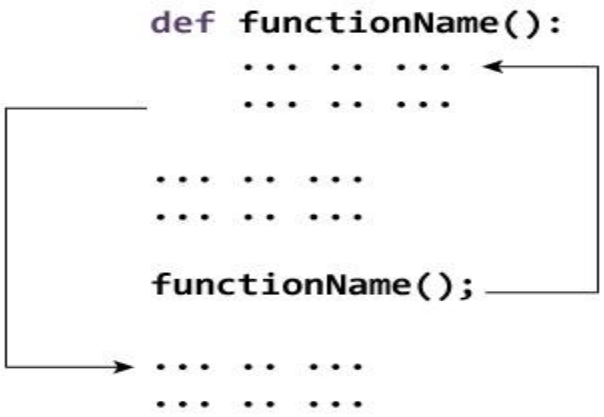
Rules of user-defined function

1. Keyword **def** marks the start of function header.
2. A function name to uniquely identify it.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (**:**) to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional **return** statement to return a value from the function.

Syntax of Function

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

How Function works in Python?



Example Program :1

```
def swap(a,b):              #Function definition
    temp=a
    a=b
    b=temp
    print("After swap a=",a,"b=",b)
    return
a=int(input("enter first number"))
b=int(input("enter second number"))
print("Before swap a=",a,"b=",b)
swap(a,b)                   #Function call
```

Example Program :2	Output
#user-defined functions def addition(x,y): sum = x + y return sum a = 5 b = 6 print("The sum is", addition(a,b))	The sum is 11

Function Call

- Once we have defined a function, we can call it from another function, program or even the Python prompt.

```
swap(a,b)
```

- To call a function we simply type the function name with appropriate parameters.

Docstring

- The first string after the function header is called the docstring and is short for documentation string. It is used to explain in brief, what a function does.
- For example:

```
temp=a
a=b
b=temp
print("After swap a=",a,"b=",b)
```

The return statement

- The **return** statement is used to exit a function and go back to the place from where it was called.
- Syntax of return

```
return [expression_list]
```

Example Program :	**Output**

#Factorial Program
def factorial(n):
  if n == 0:
  return 1
  else:
  return n * factorial(n-1)
num=int(input("Enter a number"))
print(factorial(num))
	Enter a number 5
	
	120

Scope and Lifetime of variables

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

Example Program	**Output**

def my_func():
  x = 10 # Scope variable
  print("Value inside function:",x)
x = 20 # Global variable
my_func()
print("Value outside function:",x)
	Value inside function: 10
	
	Value outside function: 20

Flow of execution

- Flow of execution specifies the order in which statements are executed.
- Program execution starts from the first statement of the program.
- One statement is executed at a time from top to bottom.
- Function definitions do not alter the flow of execution of the program, and the statements inside the function are not executed until the function is called.
- When a function is called, the control flow jumps to the body of the function, executes all the statements, and return back to the place in the program where the function call was made.
- Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it.
- When it gets to the end of the program, it terminates.

Parameters and arguments

- **Arguments** are the values provided to the function during the function call.
- **Parameters** are name used inside the function definition to refer to the value passed as an argument.
- Inside the function, the value of arguments passed during function call is assigned to parameters.

Example Program	**Output**

import math
def raise1(num,power): #parameters num, power
  print(math.pow(num,power))
a=10
b=2
raise1(a,b) #arguments a,b
	100

Functions with no arguments

- The empty parentheses after the function name indicate that this function doesn't take any arguments.

Example Program	**Output**
import math def show_PI(): print(math.pi) show_PI() #no arguments	3.141592653589793

Functions with arguments

- Functions may also receive arguments (variables passed from the caller to the function).
- Arguments in function call are assigned to function parameters.

Example Program	**Output**

import math
def circle(r):
  area=math.pi*r*r
  perimeter=2*math.pi*r
  print("area=",area)
  print("perimeter=",perimeter)
r=float(input("Enter the r value"))
circle(r) #with arguments
	Enter the r value
	3
	area= 28.274333882308138
	perimeter= 18.84955592153876

Function Arguments

- Following types of formal arguments are used in Python

    1. Required Arguments
    2. Keyword Arguments
    3. Default Arguments
    4. Variable – length Arguments (or) Arbitrary Arguments

1. Required Arguments

- In Required arguments, the number of arguments passed in the function call should match exactly with the function definition.

Example	Output
def welcome(str1,str2): print(str1 + ', ' + str2) welcome("welcome" , "First year ")	welcome , First year

2. Keyword Arguments

- When we call a function with some values, these values get assigned to the arguments according to their position.

Example	Output

def multiple_display(message, times):
  for i in range(times):
  print(message)
multiple_display(message="welcome first year", times=4)
	welcome first year
	welcome first year
	welcome first year
	welcome first year

3. Default Arguments

- We can provide a default value to an argument by using the assignment operator (=).

Example	Output
def mydetail(name,age=20): print("name=",name) print("age=",age) return; mydetail(name="raja",age=18) mydetail(name="kumar")	name= raja age= 18 name= kumar age= 20

4. Variable – length Arguments (or) Arbitrary Arguments

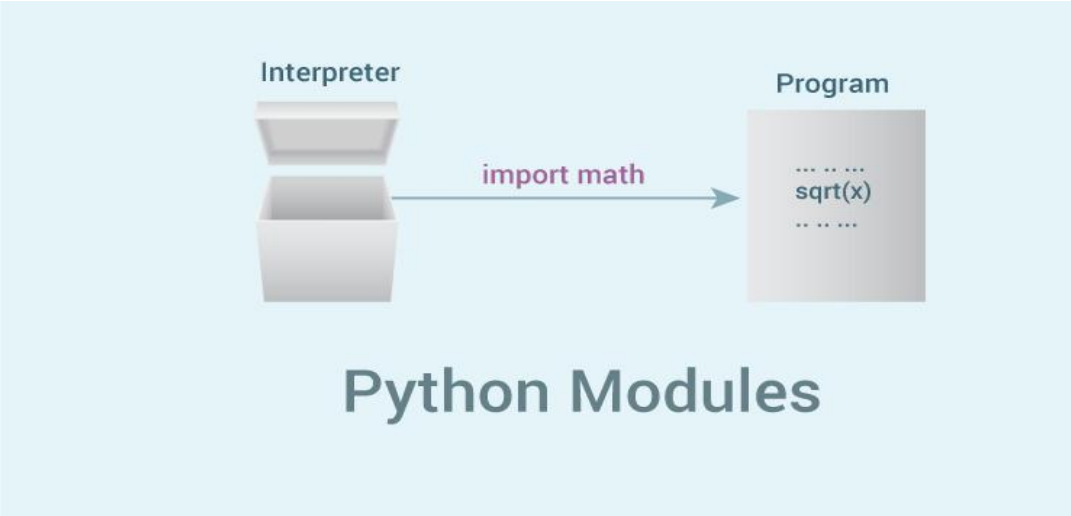
- Sometimes, the number of arguments that will be passed into a function is not known in advance.
- Python allows us to handle this kind of situation through function calls with arbitrary number of arguments.
- In the function definition we use an asterisk (*) before the parameter name to denote this kind of argument.

Example	Output

def dept(*names):
# names is a tuple with arguments 	
for name in names:
print("Hello",name)
dept("cse","mech","ece","civil")
	Hello cse
	Hello mech
	Hello ece
	Hello civil

Modules

Definition:



- Modules refer to a file containing Python statements and definitions.
- It defines functions, classes and variables and includes runnable code also.
- Functions are groups of code and modules are groups of functions.
- We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.
- We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Create a module:

- Type the following and save it as example.py.

```
# Python Module example
def add(a, b):
   """This program adds two
   numbers and return the result"""
   result = a + b
   return result
```

Here, we have defined a function **add()** inside a module named example. The function takes in two numbers and returns their sum.

How to import modules in Python?

- We can import the definitions inside a module to another module or the interactive interpreter in Python.
- We use the **import** keyword to do this. To import our previously defined module example we type the following in the Python prompt.

```
>>> import example
```

- This does not enter the names of the functions defined in **example** directly in the current symbol table. It only enters the module name **example** there.
- Using the module name we can access the function using **dot (.)** operation. For example:

```
>>> example.add(4,5)

9
```

- Python has a ton of standard modules available.
- You can check out the full list of Python standard modules and what they are for. These files are in the Lib directory inside the location where you installed Python.
- Standard modules can be imported the same way as we import our user-defined modules.
- There are various ways to import modules. They are listed as follows.

Python from...import statement

- We can import specific names form a module without importing the module as a whole.
- Syntax:

```
from  python_file  import  function_name
```

- Example:

Example	Output
# import pi & e from math module import math print("The value of pi is", math.pi) print("The value of Eulers is", math.e)	The value of pi is 3.141592653589793 The value of Eulers is 2.718281828459045

- We imported only the attribute pi form the module.
- In such case we don't use the dot operator. We could have imported multiple attributes as follows.

```
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
```

Import with renaming

- We can import a module by renaming it as follows.
- We have renamed the **math** module as **m**. This can save us typing time in some cases.
- Note that the name **math** is not recognized in our scope. Hence, **math.pi** is invalid, **m.pi** is the correct implementation.

Example	Output
# import module by renaming it import math as m print("The value of pi is", m.pi)	The value of pi is 3.141592653589793

Import all names

- We can import all names(definitions) form a module using the following construct.

  Syntax:

```
from  python_file  import *
```

- We imported all the definitions from the math module. This makes all names except those beginnig with an underscore, visible in our scope.
- Importing everything with the asterisk (*****) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier

Example	Output
# import all names form # the standard module math from math import * print("The value of pi is", pi)	The value of pi is 3.141592653589793

Example Programs:

Example Programs:	**Output:**

#from …..import Statement
def add(a,b):
  result=a+b
  return result
def sub(a,b):
  result=a-b
  return result
	>>> from import1 import add
	>>> add(8,9)
	17
	>>> from import1 import sub
	>>> sub(9,5)
	4

Illustrative programs:

1. Exchange the values of two variables

Programs	**Output:**

def swap(a,b): #Function definition
  a,b=b,a
  print("After swap a=",a,"b=",b)
  return
a=int(input("enter first number"))
b=int(input("enter second number"))
print("Before swap a=",a,"b=",b)
swap(a,b) #Function call
	enter first number 10
	enter second number 20
	Before swap a= 10 b= 20
	After swap a= 20 b= 10

2. Exchange the values using third (temporary) variable

Programs	**Output:**

def swap(a,b): #Function definition
  temp=a
  a=b
  b=temp
  print("After swap a=",a,"b=",b)
  return
a=int(input("enter first number"))
b=int(input("enter second number"))
print("Before swap a=",a,"b=",b)
swap(a,b) #Function call
	enter first number 10
	enter second number 20
	Before swap a= 10 b= 20
	After swap a= 20 b= 10

3. Circulate the values of n variables

Programs	**Output:**

def circulate(a,b,c):
  temp=a
  a=b
  b=c
  c=temp
  print("Before circulate a=",a,"b=",b,"c=",c)
a=int(input("enter the a value"))
b=int(input("enter the b value"))
c=int(input("enter the c value"))
print("Before circulate a=",a,"b=",b,"c=",c)
circulate(a,b,c)
	enter the a value 5
	enter the b value 6
	enter the c value 7
	Before circulate a= 5 b= 6 c= 7
	Before circulate a= 6 b= 7 c= 5

4. Distance between two points.

Programs	**Output:**

import math
def Distance(x1,y1,x2,y2):
  dx=x2-x1;
  dy=y2-y1;
  dist = dx**2 + dy**2
  result = math.sqrt(dist)
  return result
x1=int(input("entr x1"))
y1=int(input("entr y1"))
x2=int(input("entr x2"))
y2=int(input("entr y2"))
print(Distance(x1, y1, x2, y2))
	enter x1 2
	enter y1 2
	enter x2 4
	enter y2 4
	2.8284271247461903