Subject :Python Programming
Subject Code:16MCA21

IA1 -Solution

1a)Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

- **Python is Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- **Python is a Beginner's Language:** Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Python's features include:

- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
- **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
- **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode:**Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python provides interfaces to all major commercial databases.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable:** Python provides a better structure and support for large programs than shell scripting.

b)

```
#program to find the largest of three nos:
a=int(input("enter first no"))
b=int(input("enter second no"))
c=int(input("enter third no"))
if b<a>c:
    print a," is greatest"
elif a<b>c:
     print b,"is greatest"
else:
    print c,"is greatest"
```

c) i)True ii) True

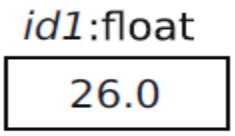
2a) A name that refers to a value is called a *variable*. In Python,variable names can use letters, digits, and the underscore symbol (but theycan't start with a digit). For example, X, species5618, and degrees_celsius are all allowed, but 777 isn't (it would be confused with a number), and neither isno-way! (it contains punctuation).

Subject :Python Programming
Subject Code:16MCA21

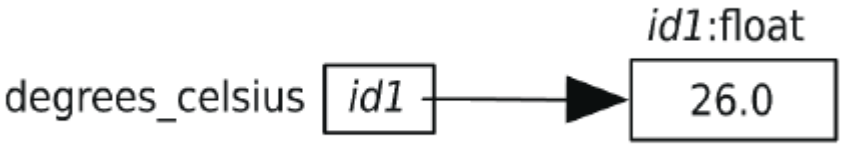
You create a new variable by *assigning* it a value:

```
>>> degrees_celsius = 26.0
```

a *memory model*—that will let us trace what happens when Python executes a Python program. This memory model will help us accurately predict and explain what Python does when it executes code, a skill that is a requirement for becoming a good programmer.Every location in the computer's memory has a *memory address*, much like an address for a house on a street, that uniquely identifies that location.



During execution of a program,every value that Python keeps track of is stored inside an object in computer memory.In our memory model, a variable contains the memory address of the object to which it refers:



b)#python program demonstrating customer class

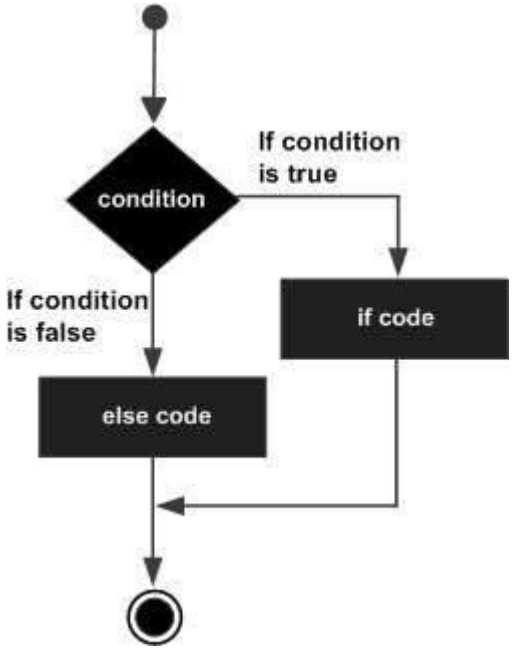
```
class customer:
        def _init_(self,cname,balance):
                self.cname=cname
                self.balance=balance
        def  withdraw(amt):
                 if amt>self.balance:
                      print"insufficient funds"
                 else:
                       self.balance=self.balance-amt
                       print "amt is debited"
         def   deposit(amt):
                   self.balance=self.balance+amt
                   print "amt is credited"
#main code
 C=customer()
c.withdraw(1000)
c.deposit(10000)
```

3a) Conditionals in Python:

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

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

Flow Diagram



The *elif* Statement

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

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

syntax

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

b)output:

able

ble

le

e

4a) 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.

Defining a Function

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

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

Syntax

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

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

Example

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return
```

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

```
#!/usr/bin/python
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print str
   return;
# Now you can call printme function
printme("I'm first call to user defined function!")
printme("Again second call to the same function")
```

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

```
I'm first call to user defined function!
Again second call to the same function
```

Subject :Python Programming
Subject Code:16MCA21

b) output:5

5a) A string is a sequence of characters.

A character is simply a symbol. For example, the English language has 26 characters.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encoding used.

In Python, string is a sequence of Unicode character. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

For example –

```
var1 = 'Hello World!'

var2 = "Python Programming"
```

Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.

The syntax for triple quotes consists of three consecutive **single or double** quotes.

```
#!/usr/bin/python
para_str = """
```

[\n], or just a NEWLINE within

the variable assignment will also show up.

```
"""print para_str
```

b) String methods:

str.capitalize()	Returns a copy of the string with the first letter capitalized and the rest lowercase
str.count(s)	Returns the number of nonoverlapping occurrences of s in the string
str.endswith(end)	Returns True iff the string ends with the characters in the end string—this is case sensitive.
str.find(s)	Returns the index of the first occurrence of s in the string, or -1 if s doesn't occur in the string—the first character is at index 0. This is case sensitive.
str.find(s, beg)	Returns the index of the first occurrence of s at or after index beg in the string, or -1 if s doesn't occur in the string at or after index beg—the first character is at index 0. This is case sensitive.
str.find(s)	Returns the index of the first occurrence of s in the string, or -1 if s doesn't occur in the string—the first character is at index 0. This is case sensitive.
str.find(s, beg)	Returns the index of the first occurrence of s at or after index beg in the string, or -1 if s doesn't occur in the string at or after index beg—the first character is at index 0. This is case sensitive.
str.find(s, beg, end)	Returns the index of the first occurrence of s between indices beg (inclusive) and end (exclusive) in the string, or -1 if s does not occur in the string between indices beg and end—the first character is at index 0. This is case sensitive.
str.format(«expressions»)	Returns a string made by substituting for placeholder fields in the string—each field is a pair of braces ('{' and '}') with an integer in between; the expression arguments are numbered from left to right starting

```
6a) def binary_search_recursive(li, left, right, key):
  while True:
```

```
    if left > right:
      return -1
    mid = (left + right) / 2
    if li[mid] == key:
      return mid
    if li[mid] > key:
      right = mid - 1
    else:
      left = mid + 1
    return binary_search_recursive(li, left, right, key)

if __name__ == "__main__":
  li = [1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12]
  left = 0
  right = len(li)
  for key in  [8, 6, 1, 12, 7]:
    index = binary_search_recursive(li, left, right, key)
    print key, index
```

b)i)4 ii)4 iii)5.5 iv)True

7a)n=3

```
For  I in range(0,n):
   For j in range(0,i+1):
   Print("*",end=" ")
 Print(" ")
```

b)

```
ph=2
if ph<3:
   print(ph,"is very acidic! Be careful")
elif ph>3 and ph<7:
    print(ph,"is acidic")
```

8) a)docstring A **docstring** is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a **docstring** becomes the __doc__ special attribute of that object. All modules should normally have **docstrings**, and all functions and classes exported by a module should also have**docstrings**.

b) A module is a python file that (generally) has only defenitions of variables, functions, and classes. A **Python module** is simply a **Python** source file, which can expose classes, functions and global variables. When imported from another **Python** source file, the file name is treated as a namespace. A **Python package** is simply a directory of**Python module**(s)

A Python module is simply a Python source file, which can expose classes, functions and global variables.When imported from another Python source file, the file name is treated as a namespace.A Python package is simply a directory of Python module(s).

```
import mypackage.mymodule
```

or

```
from mypackage.mymodule import myclass
```

There are Many Ways to Import a Module #

Python provides at least three different ways to import modules. You can use the *import* statement, the *from* statement, or the builtin *__import__* function. (There are more contrived ways to do this too, but that's outside the scope for this small note.)

Anyway, here's how these statements and functions work:

- **import X** imports the module X, and creates a reference to that module in the current namespace. Or in other words, after you've run this statement, you can use *X.name* to refer to things defined in module X.

- **from X import *** imports the module X, and creates references in the current namespace to all *public* objects defined by that module (that is, everything that doesn't have a name starting with "_"). Or in other words, after you've run this statement, you can simply use a plain *name*to refer to things defined in module X. But X itself is not defined, so *X.name* doesn't work. And if *name* was already defined, it is replaced by the new version. And if *name* in X is changed to point to some other object, your module won't notice.
- **from X import a, b, c** imports the module X, and creates references in the current namespace to the given objects. Or in other words, you can now use *a* and *b* and *c* in your program.
- Finally, **X = __import__('X')** works like **import X**, with the difference that you 1) pass the module name as a string, and 2) explicitly assign it to a variable in your current namespace.

c)Methds with _ underscore:

Any method (or other name) beginning and ending with two underscores is considered special by Python. The help documentation for strings shows these methods, among many others:

```
|  Methods defined here:
|
|  __add__(...)
|      x.__add__(y) <==> x+y
```

These methods are typically connected with some other syntax in Python: use of that syntax will trigger a method call. For example, string method __add__ is called when anything is added to a string:

```
>>> 'TTA' + 'GGG'
'TTAGGG'
>>> 'TTA'.__add__('GGG')
'TTAGGG'
```

The documentation describes when these are called. Here we show both versionsof getting the absolute value of a number:

```
>>> abs(-3)
3
>>> -3 .__abs__()
3
```

We need to put a space after -3 in the second instance (with the underscores)so that Python doesn't think we're making a floating-point number -3.(remember that we can leave off the trailing 0).

Let's add two integers using this trick:

```
>>> 3 + 5
8
>>> 3 .__add__(5)
8
```