CMR INSTITUTE OF TECHNOLOGY	USN										

Internal Assesment Test - I

Sub:	**Introduction to Python, Data and Control Systems**							Code:	**22MBABA303**
Date:	**20-01-2024**	Duration:	**90 mins**	Max Marks:	**50**	Sem:	**III**	Branch:	**MBA**

SET- II

		Marks	OBE CO	OBE RBT
	Part A - Answer Any Two Full Questions (2* 20 = 40 marks)			
1 (a)	**Difference between a word and a sentence in programming.** Unlike human languages, the Python vocabulary is actually pretty small. We call this "vocabulary" the "reserved words". These are words that have very special meaning to Python. When Python sees these words in a Python program, they have one and only one meaning to Python. Later as you write programs you will make up your own words that have meaning to you called *variables*. You will have great latitude in choosing your names for your variables, but you cannot use any of Python's reserved words as a name for a variable. **Reserve words:** and del global not with as elif if or yield assert else import pass break except in raise class finally is return continue for lambda try def from nonlocal while	[03]	CO1	L1
(b)	**Explain the concept of debugging in python program.** When Python **spits out an error** or even when it gives you a result that is **different from what you had intended**, then begins the hunt for the **cause of the error.** Debugging is the **process of finding the cause of the error in your code.** When you are debugging a program, and especially if you are working on a **hard bug, there are four things** to try: When Python spits out an error or even when it gives you a result that is different from what you had intended, then begins the hunt for the cause of the error. Debugging is the process of finding the cause of the error in your code. When you are debugging a program, and especially if you are working on a hard bug, there are four things to try: **Reading** Examine your code, read it back to yourself, and check that it says what you meant to say. **Running** Experiment by making changes and running different versions. Often if you display the right thing at the right place in the program, the problem becomes obvious, but sometimes you have to spend some time to build scaffolding. **Ruminating** Take some time to think! What kind of error is it: syntax, runtime, semantic? What information can you get from the error messages, or from the output of the program? What kind of error could cause the problem you're seeing? What did you change last, before the problem appeared? **Retreating** At some point, the best thing to do is back off, undoing recent changes, until you get back to a program that works and that you understand. Then you can start rebuilding.	[07]	CO1	L2
(c)	**Analyze the importance of terminology in programming languages.** These programming language translators fall into two general categories: (1) interpreters and (2) compilers. An ***interpreter*** reads the source code of the program as written by the programmer, parses the source code, and interprets the instructions on the fly. Python is an interpreter and when we are running Python interactively, we can type a line of Python (a sentence) and Python processes it immediately and is ready for us to type another line of Python.	[10]	CO1	L4

	Some of the lines of Python tell Python that you want it to remember some value for later. We need to pick a name for that value to be remembered and we can use that symbolic name to retrieve the value later. We use the term *variable* to refer to the labels we use to refer to this stored data. A ***compiler*** needs to be handed the entire program in a file, and then it runs a process to translate the high-level source code into machine language and then the compiler puts the resulting machine language into a file for later execution.			
2 (a)	**Define the term Interpreter.** An ***interpreter*** reads the source code of the program as written by the programmer, parses the source code, and interprets the instructions on the fly. Python is an interpreter and when we are running Python interactively, we can type a line of Python (a sentence) and Python processes it immediately and is ready for us to type another line of Python	[03]	CO1	L1
(b)	**Discuss the role of comments in Python programming with an example.** As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why. For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called *comments*, and in Python they start with the # **symbol:** *# compute the percentage of the hour that has elapsed* Percentage = (minute * 100) / 60 In this case, the comment appears on a line by itself. You can also put comments at the end of a line: Percentage = (minute * 100) / 60 *# percentage of an hour* Everything from the # to the end of the line is ignored; it has no effect on the program. Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out *what* the code does; it is much more useful to explain *why*. This comment is redundant with the code and useless: v = 5 *# assign 5 to v* This comment contains useful information that is not in the code: v = 5 *# velocity in meters/second.* Good variable names can reduce the need for comments, but long names can make complex expressions hard to read, so there is a trade-off.	[07]	CO1	L2
(c)	**Discuss the three major three types of errors that occur in python programming.** As your programs become increasingly sophisticated, you will encounter three general types of errors: **Syntax errors** These are the first errors you will make and the easiest to fix. A syntax error means that you have violated the "grammar" rules of Python. Python does its best to point right at the line and character where it noticed it was confused. The only tricky bit of syntax errors is that sometimes the mistake that needs fixing is actually earlier in the program than where Python *noticed* it was confused. So the line and character that Python indicates in a syntax error may just be a starting point for your investigation. **Logic errors** A logic error is when your program has good syntax but there is a mistake in the order of the statements or perhaps a mistake in how the statements relate to one another. A good example of a logic error might be, "take a drink from your water bottle, put it in your backpack, walk to the library, and then put the top back on the bottle." **Semantic errors** A semantic error is when your description of the steps to take is syntactically perfect and in the right order, but there is simply a mistake in the program. The program is perfectly correct but it does not do what you *intended* for it to do.	[10]	CO1	L2
3 (a)	**Define boolean data type and write example?** **Python boolean** type is one of the built-in data types provided by Python, which represents one of the two values i.e. True or False. Generally, it is used to represent the truth values of the expressions. **Example:** **Input:** 1==1 **Output:** True **Input:** 2<1 **Output:** False	[03]	CO2	L1
(b)	**Describe the concept of a function and its use in Python.**	[07]	CO2	L2

	In the context of programming, a *function* is a named sequence of statements that performs a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name. We have already seen one example of a *function call*: >>> type(32) <class 'int'> The name of the function is type. The expression in parentheses is called the *argument* of the function. The argument is a value or variable that we are passing into the function as input to the function. The result, for the type function, is the type of the argument. It is common to say that a function "takes" an argument and "returns" a result. The result is called the *return value*. So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A *function definition* specifies the name of a new function and the sequence of statements that execute when the function is called. Once we define a function, we can reuse the function over and over throughout our program. Here is an example: **def** print_lyrics(): print("I'm a lumberjack, and I'm okay.") print('I sleep all night and I work all day.') def is a keyword that indicates that this is a function definition. The name of the function is print_lyrics. The rules for function names are the same as for variable names: letters, numbers and some punctuation marks are legal, but the first character can't be a number. You can't use a keyword as the name of a function, and you should avoid having a variable and a function with the same name. The empty parentheses after the name indicate that this function doesn't take any arguments. Later we will build functions that take arguments as their inputs. The first line of the function definition is called the *header*; the rest is called the *body*. The header has to end with a colon and the body has to be indented. By convention, the indentation is always four spaces. The body can contain any number of statements.			

(c)	**Write a Python program that uses variables, expressions, and statements to perform a specific task.**
	Variables:
	One of the most powerful features of a programming language is the ability to manipulate *variables*. A variable is a name that refers to a value.
	An *assignment statement* creates new variables and gives them values:
	>>> message = 'And now for something completely different'
	>>> n = 17
	>>> pi = 3.1415926535897931
	This example makes three assignments. The first assigns a string to a new variable named message; the second assigns the integer 17 to n; the third assigns the (approximate) value of _ to pi.
	To display the value of a variable, you can use a print statement:
	>>> print(n)
	17
	>>> print(pi)
	3.141592653589793
	The type of a variable is the type of the value it refers to.
	>>> type(message)
	<class 'str'>
	>>> type(n)
	<class 'int'>
	>>> type(pi)
	<class 'float'>
	A *statement* is a unit of code that the Python interpreter can execute. We have seen two kinds of statements: print being an expression statement and assignment. When you type a statement in interactive mode, the interpreter executes it and displays the result, if there is one.
	A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.
	For example, the script
	print(1)
	x = 2
	print(x)
	produces the output
		[10]	CO2	L3

1
2
The assignment statement produces no output.
An *expression* is a combination of values, variables, and operators. A value all by itself is considered an expression, and so is a variable, so the following are all legal expressions (assuming that the variable x has been assigned a value):
17
x
x + 17
If you type an expression in interactive mode, the interpreter *evaluates* it and displays the result:
>>> 1 + 1
2
But in a script, an expression all by itself doesn't do anything! This is a common source of confusion for beginners.

Part B - Compulsory (01*10=10 marks) – CASE STUDY

4 **Write a Python program on arithmetic expressions. Identify the variables, their data types and write the output. [10]** CO2 L3

Arithmetic Expressions: An arithmetic expression is a combination of **numeric values, operators,** and sometimes **parenthesis**. The result of this type of expression is also a **numeric value**. The operators used in these expressions are arithmetic operators like **addition, subtraction, etc**. Here are some arithmetic operators in Python:

Example:

Let's see an exemplar code of arithmetic expressions in Python :

Python3

```python
# Arithmetic Expressions
x = 40
y = 12

add = x + y
sub = x - y
pro = x * y
div = x / y

print(add)
print(sub)
print(pro)
print(div)
```

Output

```
52
28
480
3.3333333333333335
```

Data type:
type(x) – int
type(y) – int
type(3.3) – float
type('hello world') - str

*****