KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Coimbatore – 641 021.
(For the Candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

SUBJECT: PROGRAMMING IN PYTHON
SEMESTER: V
SUB.CODE:16CAU501B **CLASS: III BCA**

SCOPE

This course is an introduction to the Python programming language. The course cover data types, control flow, object-oriented programming, and graphical user interface-driven applications.

OBJECTIVES

The educate students how to:

- use Python interactively
- use Python types, expressions, and None • use string literals and string type
- use Python statements (if...elif..else, for, pass, continue, . . .)
- utilize high-level data types such as lists and dictionaries.

UNIT-I

Planning the Computer Program: Concept of problem solving-Problem definition- Program design-Debugging-Types of errors in programming-Documentation.

UNIT-II

Techniques of Problem Solving: Flowcharting-decision table-algorithms-Structured programming concepts-Programming methodologies: top-down and bottom-up Programming.

UNIT-III

Overview of Programming: Structure of a Python Program-Elements of Python.

UNIT-IV

Introduction to Python: Python Interpreter-Using Python as calculator-Python shell- Indentation. Atoms-Identifiers and keywords-Literals-Strings-Operators(Arithmetic operator, Relational operator, Logical or Boolean operator, Assignment, Operator, Ternary operator, Bit wise operator, Increment or Decrement operator).

UNIT-V

Creating Python Programs: Input and Output Statements-Control statements(Branching, Looping, Conditional Statement, Exit function, Difference between break, continue and pass.). Defining Functions-Default arguments.

SUGGESTED READINGS

1. Allen Downey, Jeffrey Elkner, Chris Meyers, (2012). How to think like a computer scientist : learning with Python , Freely available online.

2. Budd,T.,(2011).*Exploring Python*, (1st ed.) TMH

WEBSITES

1. http://docs.python.org/3/tutorial/index.html.
2. http://interactivepython.org/courselib/static/pythonds.
3. http://www.ibiblio.org/g2swap/byteofpython/read/.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

Coimbatore – 641 021.

(For the Candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

SUBJECT: PROGRAMMING IN PYTHON
SEMESTER: V
SUB.CODE:16CAU501B **CLASS: III BCA**

LECTURE PLAN
DEPARTMENT OF COMPUTER APPLICATIONS

S.No	Lecture Duration Period	Topics to be Covered	Support Material/Page Nos
		UNIT-I	
1	1	Planning the Computer Program – Concept of problem solving	T1: 1,W3
2	1	Problem Definition	T1:1 , W3
3	1	Program Design	T1: 3-4, W1
4	1	Debugging	T1: 4, T1: 219 – 228
5	1	Types of errors in Programming	T1: 4-6
6	1	Documentation	T1: 241 – 247, W4 ,W2
7	1	Recapitulation and Discussion of important Questions	
	Total No of Hours Planned For Unit 1=7		
		UNIT-II	
1	1	Techniques of Problem Solving	W6
2	1	Decision table	W5
3	1	Algorithms	T1:142-143,W9
4	1	Structured Programming Concepts	T1:6-8,W7
5	1	Programming Methodologies: Top down and Bottom Up Programming	W8

6	1	Recapitulation and Discussion of important Questions	
	Total No of Hours Planned For Unit II=6		
		UNIT-III	
1	1	Overview of Programming	T1:1-3, W10
2	1	Overview of Programming – History of python	T1: 4-6, W10
3	1	Structure of a python program	T1:8,T2: 1-2, W10
4	1	Elements of python	W11
5	1	Recapitulation and Discussion of important Questions	
	Total No of Hours Planned For Unit III=5		
		UNIT-IV	
1	1	Introduction to python – Python Interpreter	R1: 1- 5
2	1	Using Python as a Calculator	W1
3	1	Python Shell – Indentation	R1:5-8 , W1
4	1	Atoms – Identifiers and Keywords	T1:11-14,W12
5	1	Literals , Strings	T1:71-78 , T2: 65 – 66, W1
6	1	Operators – Arithmetic , Relational, Logical or Boolean Operator	T1 : 16 – 17, T1: 35 - 37
7	1	Operators –Assignment , Ternary, Bitwise Operator, Increment or Decrement Operator	T2: 13 – 14, T2: 3 – 4
8	1	Recapitulation and Discussion of important Questions	
	Total No of Hours Planned For Unit IV=8		
		UNIT-V	
1	1	Creating Python Programs : Input and Output Statements	T2 : 22, W1
2	1	Control Statement – Branching and Looping	T2: 33- 38 , W1,W10
3	1	Control Statement – Conditional and Exit	T2: 33- 38 , W1,W10
4	1	Difference between break, continue and pass	T2: 38 – 39, W1
5	1	Defining Functions	T1: 21 – 28, T2: 46 -52

6	1	Default Arguments	T1: 28 – 30 , T2: 59 – 60 ,W10
7	1	Recapitulation and discussion of Important Questions	
8	1	Discussion of previous year ESE Question Paper	
9	1	Discussion of previous year ESE Question Paper	
10	1	Discussion of previous year ESE Question Paper	
	Total No of Hours Planned for unit V=10		
Total Planned Hours	**36**		

SUGGESTED READINGS

T1: Allen Downey, Jeffrey Elkner, Chris Meyers, (2012). How to think like a computer scientist : learning with Python , Freely available online.

T2: Budd,T.,(2011).*Exploring Python*, (1st ed.) TMH

R1:Richard L.Halterman , (2011), "Learning to Program with Python"

WEBSITES

W1: http://docs.python.org/3/tutorial/index.html

W2: http://interactivepython.org/courselib/static/pythonds

W3: http://www.ibiblio.org/g2swap/byteofpython/read/

W4: https://devguide.python.org/documenting

W5:https://rosettacode.org/wiki/Decision_tables

W6: www.codeavengers.com/notes/planning/flowch

W7: clas.mq.edu.au/speech/synthesis/basic-programming/index.html

W8: www.tutorialspoint.com/programming-methodologies

W9: www.tutorialspoint.com/python/python-algorithm-design.htm

W10: www.tutorialspoint.com/python

W11: //pentagle.net/python/handbook/Elelments.html

W12: www.journaldev.com/13976/python-keywords

UNIT – I

SYLLABUS

Planning the Computer Program: Concept of problem solving-Problem definition- Program design-Debugging-Types of errors in programming-Documentation.

PLANNING THE COMPUTER PROGRAM

CONCEPT OF PROBLEM SOLVING

- Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem solving skills.

PROBLEM DEFINITION

- The problem is *'I want a program which creates a backup of all my important files'*.
- Although, this is a simple problem, there is not enough information for us to get started with the solution. A little more **analysis** is required. For example, how do we specify which files are to be backed up? Where is the backup stored? How are they stored in the backup?
- After analyzing the problem properly, we **design** our program. We make a list of things about how our program should work. In this case, I have created the following list on how *I* want it to work. If you do the design, you may not come up with the same kind of problem - every person has their own way of doing things, this is ok.

1. The files and directories to be backed up are specified in a list.
2. The backup must be stored in a main backup directory.

3. The files are backed up into a zip file.
4. The name of the zip archive is the current date and time.
5. We use the standard **zip** command available by default in any standard Linux/Unix distribution. Windows users can use the Info-Zip program. Note that you can use any archiving command you want as long as it has a command line interface so that we can pass arguments to it from our script.

THE SOLUTION

- As the design of our program is now stable, we can write the code which is an **implementation** of our solution.

FIRST VERSION

EXAMPLE: 10.1. A BACKUP SCRIPT - THE FIRST VERSION

```
#!/usr/bin/python
# Filename: backup_ver1.py

import os
import time

# 1. The files and directories to be backed up are specified in a
list.
source = ['/home/swaroop/byte', '/home/swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents',
r'D:\Work'] or something like that

# 2. The backup must be stored in a main backup directory
target_dir = '/mnt/e/backup/' # Remember to change this to what
you will be using
```

```
# 3. The files are backed up into a zip file.
# 4. The name of the zip archive is the current date and time
target = target_dir + time.strftime('%Y%m%d%H%M%S') + '.zip'

# 5. We use the zip command (in Unix/Linux) to put the files in a
zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))

# Run the backup
if os.system(zip_command) == 0:
  print 'Successful backup to', target
else:
  print 'Backup FAILED'
```

OUTPUT

```
$ python backup_ver1.py
Successful backup to /mnt/e/backup/20041208073244.zip
```

- Now, we are in the **testing** phase where we test that our program works properly. If it doesn't behave as expected, then we have to **debug** our program i.e. remove the *bugs* (errors) from the program.

How It Works

- You will notice how we have converted our *design* into *code* in a step-by-step manner.

- We make use of the os and time modules and so we import them. Then, we specify the files and directories to be backed up in the source list. The target directory is where store all the backup files and this is specified in the target_dir variable. The name of the zip archive that we are going to create is the current date and time which we fetch using the time.strftime() function. It will also have the .zipextension and will be stored in the target_dir directory.
- The time.strftime() function takes a specification such as the one we have used in the above program. The %Y specification will be replaced by the year without the cetury. The %m specification will be replaced by the month as a decimal number between 01 and 12 and so on. The complete list of such specifications can be found in the [Python Reference Manual] that comes with your Python distribution. Notice that this is similar to (but not same as) the specification used in print statement (using the % followed by tuple).
- We create the name of the target zip file using the addition operator which *concatenates* the strings i.e. it joins the two strings together and returns a new one. Then, we create a string zip_command which contains the command that we are going to execute. You can check if this command works by running it on the shell (Linux terminal or DOS prompt).
- The **zip** command that we are using has some options and parameters passed. The -q option is used to indicate that the zip command should work **q**uietly. The -r option specifies that the zip command should work **r**ecursively for directories i.e. it should include subdirectories and files within the subdirectories as well. The two options are combined and specified in a shorter way as -qr. The options are followed by the name of the zip archive to create followed by the list of files and directories to backup. We convert

the source list into a string using the join method of strings which we have already seen how to use.

- Then, we finally *run* the command using the os.system function which runs the command as if it was run from the *system* i.e. in the shell - it returns 0 if the command was successfully, else it returns an error number.
- Depending on the outcome of the command, we print the appropriate message that the backup has failed or succeeded and that's it, we have created a script to take a backup of our important files!

Note to Windows Users

You can set the `source` list and `target` directory to any file and directory names but you have to be a little careful in Windows. The problem is that Windows uses the backslash (`\`) as the directory separator character but Python uses backslashes to represent escape sequences!

So, you have to represent a backslash itself using an escape sequence or you have to use raw strings. For example, use `'C:\\Documents'` or `r'C:\Documents'` but do **not** use `'C:\Documents'` - you are using an unknown escape sequence `\D` !

- Now that we have a working backup script, we can use it whenever we want to take a backup of the files. Linux/Unix users are advised to use the executable method as discussed earlier so that they can run the backup script anytime anywhere. This is called the **operation** phase or the **deployment** phase of the software.
- The above program works properly, but (usually) first programs do not work exactly as you expect. For example, there might be problems if you have not designed the program properly or if you have made a mistake in typing the

code, etc. Appropriately, you will have to go back to the design phase or you will have to debug your program.

DEBUGGING

- **What is debugging?**
    - Programming is a complex process, and because it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called bugs and the process of tracking them down and correcting them is called debugging. Three kinds of errors can occur in a program: syntax errors, runtime errors, and semantic errors. It is useful to distinguish between them in order to track them down more quickly.
        - **Syntax errors:** Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. Syntax refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. This sentence contains a syntax error. So does this one for most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without spewing error messages. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will print an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.
        - **Runtime errors:** The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened. Runtime errors are rare in the simple

programs you will see in the first few chapters, so it might be a while before you encounter one.

- **Semantic errors:** The third type of error is the semantic error. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do. The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.
- **Experimental debugging**: One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming. In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see. Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out, “When you have eliminated the impossible, whatever remains, however improbable, must be the truth.” (A. Conan Doyle, The Sign of Four) For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does something and make small modifications, debugging them as you go, so that you always have a working program.
-

TYPES OF ERRORS IN PROGRAMMING

- Different kinds of errors can occur in a program, and it is useful to distinguish among them in order to track them down more quickly:

• **Syntax errors** are produced by Python when it is translating the source code into byte code. They usually indicate that there is something wrong with the syntax of the program. Example: Omitting the colon at the end of a def statement yields the somewhat redundant message SyntaxError: invalid syntax.

• **Runtime errors** are produced by the runtime system if something goes wrong while the program is running. Most runtime error messages include information about where the error occurred and what functions were executing. Example: An infinite recursion eventually causes a runtime error of "maximum recursion depth exceeded."

• **Semantic errors** are problems with a program that compiles and runs but doesn't do the right thing. Example: An expression may not be evaluated in the order you expect, yielding an unexpected result. The first step in debugging is to figure out which kind of error you are dealing with. Although the following sections are organized by error type, some techniques are applicable in more than one situation.

1. Syntax errors

- Syntax errors are usually easy to fix once you figure out what they are. Unfortunately, the error messages are often not helpful. The most common 220 Debugging messages are SyntaxError: invalid syntax and SyntaxError: invalid token, neither of which is very informative.
- On the other hand, the message does tell you where in the program the problem occurred. Actually, it tells you where Python noticed a problem, which is not necessarily where the error is. Sometimes the error is prior to the location of the error message, often on the preceding line.
- If you are building the program incrementally, you should have a good idea about where the error is. It will be in the last line you added.

- If you are copying code from a book, start by comparing your code to the book's code very carefully. Check every character. At the same time, remember that the book might be wrong, so if you see something that looks like a syntax error, it might be.
- Here are some ways to avoid the most common syntax errors:

1. Make sure you are not using a Python keyword for a variable name.

2. Check that you have a colon at the end of the header of every compound statement, including for, while, if, and def statements.

3. Check that indentation is consistent. You may indent with either spaces or tabs but it's best not to mix them. Each level should be nested the same amount.

4. Make sure that any strings in the code have matching quotation marks.

5. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an invalid token error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!

6. An unclosed bracket—(, {, or [—makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.

7. Check for the classic = instead of == inside a conditional. If nothing works, move on to the next section...

1.1 <u>I can't get my program to run no matter what I do.</u>

If the compiler says there is an error and you don't see it, that might be because you and the compiler are not looking at the same code. Check your programming environment to make sure that the program you are editing is the one Python is trying to run. If you are not sure, try putting an obvious and deliberate syntax error at the beginning of the program. Now run (or import) it again. If the compiler doesn't find the new error, there is probably something wrong with the way your environment is set up. If this happens, one approach is to start again with a new program like "Hello, World!," and make sure you can get a known program to run. Then gradually add the pieces of the new program to the working one.

2. Runtime errors

- Once your program is syntactically correct, Python can import it and at least start running it. What could possibly go wrong?

2.1 My program does absolutely nothing.

- This problem is most common when your file consists of functions and classes but does not actually invoke anything to start execution. This may be intentional if you only plan to import this module to supply classes and functions. If it is not intentional, make sure that you are invoking a function to start execution, or execute one from the interactive prompt. Also see the "Flow of Execution" section below.

2.2 My program hangs.

- If a program stops and seems to be doing nothing, we say it is "hanging." Often that means that it is caught in an infinite loop or an infinite recursion.

  • If there is a particular loop that you suspect is the problem, add a print statement immediately before the loop that says "entering the loop" and another immediately after that says "exiting the loop."

- Run the program. If you get the first message and not the second, you've got an infinite loop. Go to the "Infinite Loop" section below.

  • Most of the time, an infinite recursion will cause the program to run for a while and then produce a "RuntimeError: Maximum recursion depth exceeded" error. If that happens, go to the "Infinite Recursion" section below.

- If you are not getting this error but you suspect there is a problem with a recursive method or function, you can still use the techniques in the "Infinite Recursion" section.

• If neither of those steps works, start testing other loops and other recursive functions and methods.

• If that doesn't work, then it is possible that you don't understand the flow of execution in your program. Go to the "Flow of Execution" section below.

Infinite Loop

➢ If you think you have an infinite loop and you think you know what loop is causing the problem, add a print statement at the end of the loop that prints the values of the variables in the condition and the value of the condition.

For example:

```
while x > 0 and y < 0 :
# do something to x
# do something to y
print "x: ", x
 print "y: ", y
print "condition: ", (x > 0 and y < 0)
```

➢ Now when you run the program, you will see three lines of output for each time through the loop. The last time through the loop, the condition should be false. If the loop keeps going, you will be able to see the values of x and y, and you might figure out why they are not being updated correctly.

Infinite Recursion

➢ Most of the time, an infinite recursion will cause the program to run for a while and then produce a Maximum recursion depth exceeded error.

- If you suspect that a function or method is causing an infinite recursion, start by checking to make sure that there is a base case. In other words, there should be some condition that will cause the function or method to return without making a recursive invocation. If not, then you need to rethink the algorithm and identify a base case.
- If there is a base case but the program doesn't seem to be reaching it, add a print statement at the beginning of the function or method that prints the parameters. Now when you run the program, you will see a few lines of output every time the function or method is invoked, and you will see the parameters. If the parameters are not moving toward the base case, you will get some ideas about why not.

Flow of Execution

- If you are not sure how the flow of execution is moving through your program, add print statements to the beginning of each function with a message like "entering function foo," where foo is the name of the function.
- Now when you run the program, it will print a trace of each function as it is invoked.

2.3 When I run the program I get an exception.

- If something goes wrong during runtime, Python prints a message that includes the name of the exception, the line of the program where the problem occurred, and a traceback.
- The traceback identifies the function that is currently running, and then the function that invoked it, and then the function that invoked that, and so on. In other words, it traces the path of function invocations that got you to where you are. It also includes the line number in your file where each of these calls occurs.
- The first step is to examine the place in the program where the error occurred and see if you can figure out what happened. These are some of the most common runtime errors:
- **Name Error:** You are trying to use a variable that doesn't exist in the current environment. Remember that local variables are local. You cannot refer to them from outside the function where they are defined.

- **Type Error:** There are several possible causes: • You are trying to use a value improperly. Example: indexing a string, list, or tuple with something other than an integer.

  • There is a mismatch between the items in a format string and the items passed for conversion. This can happen if either the number of items does not match or an invalid conversion is called for.

  • You are passing the wrong number of arguments to a function or method. For methods, look at the method definition and check that the first parameter is self. Then look at the method invocation; make sure you are invoking the method on an object with the right type and providing the other arguments correctly.

- **Key Error:** You are trying to access an element of a dictionary using a key value that the dictionary does not contain.
- **Attribute Error:** You are trying to access an attribute or method that does not exist.
- **Index Error:** The index you are using to access a list, string, or tuple is greater than its length minus one. Immediately before the site of the error, add a print statement to display the value of the index and the length of the array. Is the array the right size? Is the index the right value?

2.4 I added so many print statements I get inundated with output.

- One of the problems with using print statements for debugging is that you can end up buried in output. There are two ways to proceed: simplify the output or simplify the program.
- To simplify the output, you can remove or comment out print statements that aren't helping, or combine them, or format the output so it is easier to understand.
- To simplify the program, there are several things you can do. First, scale down the problem the program is working on. For example, if you are sorting an array, sort a small array. If the program takes input from the user, give it the simplest input that causes the problem.

- Second, clean up the program. Remove dead code and reorganize the program to make it as easy to read as possible. For example, if you suspect that the problem is in a deeply nested part of the program, try rewriting that part with simpler structure. If you suspect a large function, try splitting it into smaller functions and testing them separately.
- Often the process of finding the minimal test case leads you to the bug. If you find that a program works in one situation but not in another, that gives you a clue about what is going on.
- Similarly, rewriting a piece of code can help you find subtle bugs. If you make a change that you think doesn't affect the program, and it does, that can tip you off.

3. **Semantic errors**

- In some ways, semantic errors are the hardest to debug, because the compiler and the runtime system provide no information about what is wrong. Only you know what the program is supposed to do, and only you know that it isn't doing it.
- The first step is to make a connection between the program text and the behavior you are seeing. You need a hypothesis about what the program is actually doing. One of the things that make that hard is that computers run so fast.
- You will often wish that you could slow the program down to human speed, and with some debuggers you can. But the time it takes to insert a few well-placed print statements is often short compared to setting up the debugger, inserting and removing breakpoints, and "walking" the program to where the error is occurring.

3.1 My program doesn't work.

- You should ask yourself these questions:

• Is there something the program was supposed to do but which doesn't seem to be happening? Find the section of the code that performs that function and make sure it is executing when you think it should.

• Is something happening that shouldn't? Find code in your program that performs that function and see if it is executing when it shouldn't.

• Is a section of code producing an effect that is not what you expected? Make sure that you understand the code in question, especially if it involves invocations to functions or methods in other Python modules. Read the documentation for the functions you invoke. Try them out by writing simple test cases and checking the results.

- In order to program, you need to have a mental model of how programs work. If you write a program that doesn't do what you expect, very often the problem is not in the program; it's in your mental model.
- The best way to correct your mental model is to break the program into its components (usually the functions and methods) and test each component independently. Once you find the discrepancy between your model and reality, you can solve the problem.
- Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

3.2 I've got a big hairy expression and it doesn't do what I expect.

- Writing complex expressions is fine as long as they are readable, but they can be hard to debug. It is often a good idea to break a complex expression into a series of assignments to temporary variables.

For example:

```
self.hands[i].addCard (self.hands[self.findNeighbor(i)].popCard())
```

This can be rewritten as:

```
neighbor = self.findNeighbor (i)
 pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard (pickedCard)
```

- The explicit version is easier to read because the variable names provide additional documentation, and it is easier to debug because you can check the types of the intermediate variables and display their values.
- Another problem that can occur with big expressions is that the order of evaluation may not be what you expect. For example, if you are translating the expression x 2π into Python, you might write:

y = x / 2 * math.pi

- That is not correct because multiplication and division have the same precedence and are evaluated from left to right. So this expression computes xπ/2.
- A good way to debug expressions is to add parentheses to make the order of evaluation explicit:

y = x / (2 * math.pi)

- Whenever you are not sure of the order of evaluation, use parentheses. Not only will the program be correct (in the sense of doing what you intended), it will also be more readable for other people who haven't memorized the rules of precedence

<u>3.3 I've got a function or method that doesn't return what I expect.</u>

- If you have a return statement with a complex expression, you don't have a chance to print the return value before returning. Again, you can use a temporary variable. For example, instead of:

return self.hands[i].removeMatches()

you could write:

count = self.hands[i].removeMatches()
return count

- Now you have the opportunity to display the value of count before returning.

<u>3.4 I'm really, really stuck and I need help.</u>

- First, try getting away from the computer for a few minutes. Computers emit waves that affect the brain, causing these effects:

• Frustration and/or rage.

• Superstitious beliefs ("the computer hates me") and magical thinking ("the program only works when I wear my hat backward").

• Random-walk programming (the attempt to program by writing every possible program and choosing the one that does the right thing).

- If you find yourself suffering from any of these symptoms, get up and go for a walk. When you are calm, think about the program. What is it doing? What are some possible causes of that behavior? When was the last time you had a working program, and what did you do next?

- ➢ Sometimes it just takes time to find a bug. We often find bugs when we are away from the computer and let our minds wander. Some of the best places to find bugs are trains, showers, and in bed, just before you fall asleep.

3.5 No, I really need help.

- ➢ It happens. Even the best programmers occasionally get stuck. Sometimes you work on a program so long that you can't see the error. A fresh pair of eyes is just the thing.
- ➢ Before you bring someone else in, make sure you have exhausted the techniques described here. Your program should be as simple as possible, and you should be working on the smallest input that causes the error. You should have print statements in the appropriate places (and the output they produce should be comprehensible). You should understand the problem well enough to describe it concisely.
- ➢ When you bring someone in to help, be sure to give them the information they need:
  - • If there is an error message, what is it and what part of the program does it indicate?
  - • What was the last thing you did before this error occurred? What were the last lines of code that you wrote, or what is the new test case that fails?
  - • What have you tried so far, and what have you learned?
- ➢ When you find the bug, take a second to think about what you could have done to find it faster. Next time you see something similar, you will be able to find the bug more quickly.
- ➢ Remember, the goal is not just to make the program work. The goal is to learn how to make the program work.

DOCUMENTATION

Installation and documentation

If you use Mac OS X or Linux, then Python should already be installed on your computer by default. If not, you can download the latest version by visiting the Python home page, at **http://www.python.org** where you will also find loads of documentation and other useful information. Windows users can also download Python at this website. Don't forget this website; it is your first point of reference for all things Python.

GNU Free Documentation License

Preamble

- The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License

Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License.

Modifications

You may copy and distribute a Modified Version of the Document under the conditions of Sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it.

Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in Section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects

Aggregation with Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate,” and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of Section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections.

Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense, or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concern

POSSIBLE QUESTIONS

UNIT – I

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. What is Debugging?
2. Define Problem
3. What is meant by Problem Solving?
4. List the Types of Errors
5. What is the process of Syntax error?
6. What is the process of Runtime error?

PART – C (6 MARKS)

1. Explain the Concept of Problem Solving
2. Explain the types of Errors in Programming
3. Explain the process of Syntax error with suitable examples
4. Discuss in detail about the Documentation
5. Explain the process of Semantic error with suitable examples
6. Discuss in detail about Debugging
7. Explain the process of Runtime error with suitable examples
8. Difference between the Syntax error and Semantic Error
9. Difference between the Syntax error and Run time Error
10. Explain about GNU Free Documentation License

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

(For the Candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - I : (Objective Type Multiple choice Questions each Question carries one Mark)

PROGRAMMING IN PYTHON [16CAU501B]

PART - A (Online Examination)

Questions	**Opt1**	**Opt2**	**Opt3**	**Opt4**	**Key**
Last step in process of problem solving is to	solution	problem	solution	organizing the data	practicing the solution
Second step in problem solving process is to	solution	problem	solution	organizing the data	design a solution
Thing to keep in mind while solving a problem is	input data	output data	stored data	all of above	all of above
First step in process of problem solving is to	design a solution	define a problem	practicing the solution	organizing the data	define a problem
Error in a program is called	bug	debug	virus	noise	bug
Error which occurs when program tried to read from file without opening it is classified as	execution error messages	built in messages	user-defined messages	half messages	execution error messages
__________ is the process of formulating a problem, finding a solution, and expressing the solution	problem solving:	Recover	Format	Retrieve	problem solving
___________ is a set of instructions that specifies a computation	Process	Program	Syntax	Error	Program
_______ is an error in a program that makes it do something other than what the programmer intended.	Syntax error	Semantic error	Run time error	Logical Error	Semantic error

_____________ is an error in a program that makes it impossible to parse (and therefore impossible to interpret).	Syntax error	Semantic error	Run time error	Logical Error	Syntax error
_____________ is an error that does not occur until the program has started to execute but that prevents the program from continuing.	Syntax error	Semantic error	Run time error	Logical Error	Run time error
_________ is an another name for Run time error	Syntax error	Semantic error	Exception	Semantics	Exception
Which translate a program written in a high-level language into a lowlevel language all at once, in preparation for later execution?	Compile	Interpret	Script	bug	Compile
Which execute a program in a high-level language by translating it one line at a time.	Compile	Interpret	Script	bug	Interpret
_________ is a program in a high-level language before being compiled.	executable	source code	object code	program	source code
_________ is the output of the compiler after it translates the program.	Coding	source code	object code	program	object code
__________ is the structure of a program	Syntax	Source	Semantics	Algorithm	Syntax
What is a property of a program that can run on more than one kind of computer.	executable	source code	Coding	Portability	Portability
__________ is another name for object code that is ready to be executed.	executable	source code	Coding	program	executable
____________ is the meaning of a program	Syntax	Source	Semantics	Algorithm	Semantics
__________ is to examine a program and analyze the syntactic structure.	Syntax	Source	Semantics	parse	parse
_________ is the process of finding and removing any of the three kinds of programming errors.	bug	debugging	virus	noise	debugging
As the design of our program is now stable, we can write the code which is an _____________ of our solution.	problem	implementation	solving	solution	implementation

A single unit which is composed of small group of bits is known as	bit	bug	flag	byte	byte
BCD stands for __________________	Binary Coded Decimal	Binary Coded Digitals	Binary Characters Decimals	Binary Conducting Decimals	Binary Coded Decimal
Software mistakes during coding are known as	errors	failures	bugs	defects	bugs
Which of the following is not a part of Execution Flow during debugging?	Step Over	Step Into	Step Up	Step Out	Step Up
What are the types of requirements ?	Availability	Reliability	Usability	All of the mentioned	All of the mentioned
Any one of the languages that people speak that evolved naturally.	natural language	formal language	assembly language	machine language	natural language
Any one of the languages that people have designed for specific purposes, such as representing mathematical ideas or computer programs;	natural language	formal language	assembly language	machine language	formal language
A________________ is a sequence of instructions that specifies how to perform a computation.	task	program	problem	debugging	program
__________ Get data from the keyboard, a file, or some other device.	input	output	math	conditional	input
Display data on the screen or send data to a file or other device.	input	output	math	conditional	output
Perform basic mathematical operations like addition and multiplication.	input	output	math	conditional	math
: Check for certain conditions and execute the appropriate sequence of statements	input	output	math	conditional	conditional
Perform some action repeatedly, usually with some variation	input	output	repetition	conditional	repetition
The second type of error is a runtime error, so called because the error does not appear until __________	you merge the program	you concatenate the program	you copy the program	you run the program	you run the program

In semantic error, the computer will not _______ any error messages	generate	sustain	process	perform	generate
__________ are formal languages that have been designed to express computations.	language	natural language	Programming languages	machine language	Programming languages
Natural languages are full of ___________, which people deal with by using contextual clues and other information	redundancy	ambiguity	literalness	process	ambiguity
In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of ___________.	redundancy	ambiguity	literalness	process	redundancy
Natural languages are full of idiom and metaphor.	redundancy	ambiguity	literalness	process	literalness
Hardware and software specifications are part of	computing requirements	statement requirements	system flowchart	decision statement	computing requirements
Distinct parts of documentation are called	technical documentation and documentation for user	technical documentation and planning	planning and documentation for user	planning and user process	technical documentation and documentation for user
Program background, program functions and the computing requirements are part of	operations detail	predefined programs	decision box	statement box	operations detail
Set of diagrams and notes that accompany program implementation are known as	program execution	program planning	program documentation	program existence	program documentation
Program documentation is used by the	programmers	system analyst	modifying the program	all of above	all of above
able which shows the results and the executed instructions is called	trace table	sequence design	dimension design	implementation design	trace table
All the steps of a program in the form of printout is classified as	arithmetic trace	trace	transfer trace	trace routine	trace
Error which results in performing the unintended operations is classified as	system error	logical error	mismatched error	stop time error	logical error

Program traces are produced while	data is entered	deciding arithmetic operations	program is running	transferring control	program is running
____________ is considered a kind of modification, so you may distribute translations of the Document under the terms of Section 4	Termination	Translation	Aggregation	Revision	Translation
You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License.	Termination	Translation	Aggregation	Revision	Termination
Name given by a programmer to any particular data is classified as	Identification	Identifier	exponent	mantissa	Identifier
To write a program function i.e. program for the sum of four integers, the program refinement first level includes	calculate sum	print the values	input four numbers	display the values	input four numbers
Data which is used to test each feature of the program and is carefully selected is classified as	program output	program input	test data	test program	test data
Defining data requirements such as input and output is classified as	process definition	function definition	print defintion	writing purpose	function definition
Method used in writing and designing of a program is termed as	bottom up method top down method		split method	binary states method	top down method
Two kinds of programs process high-level languages into low-level languages:	compilers and translators	interpreters and assemblers	interpreters and compilers.	assemblers and compilers.	interpreters and compilers.
There are two ways to use the interpreter:	commandline mode and script mode	user mode and script mode	commandline mode and user mode	commandline mode and translator mode	commandline mode and script mode

UNIT – II

SYLLABUS

Techniques of Problem Solving: Flowcharting-decision table-algorithms-Structured programming concepts-Programming methodologies: top-down and bottom-up Programming.

TECHNIQUES OF PROBLEM SOLVING

FLOWCHARTING

Introduction to planning

- It is important to be able to plan code with the use of flowcharts.
- Even though you can code without a plan, it is not good practice to start coding without a guide to follow.

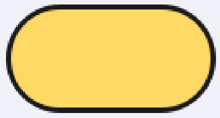
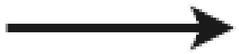
A good plan:

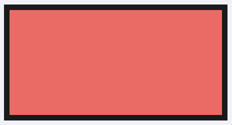
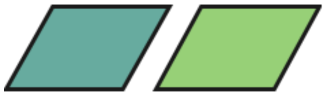
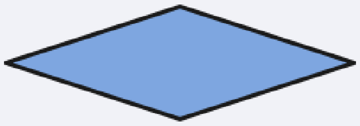
- creates a guide you can follow
- helps you plan efficient structure
- helps communicate to others what your code will do

- Poor planning can result in inefficient, unstructured code known as 'spaghetti code'.

FLOWCHARTS

- A standard way to plan code is with flowcharts.
- Specific parts of the flowchart represent specific parts of your code.

Symbol	Name	What it does in the code
	Start/End	Ovals show a start point or end point in the code
	Connection	Arrows show connections between different parts of the code

	Process	Rectangles show processes e.g. calculations (most things the computer does that does not involve an input, output or decision)
	Input/Output	Parallelograms show inputs and outputs (remember print is normally an input)
	Conditional/ Decision	Diamonds show a decision/conditional (this is normally if, else if/elif, while and for)

revision

adding text to flowcharts

- There is no one right or wrong way to label flowcharts; you are presenting the structure of your code in a way that humans can understand. Only add extra details to parts of the flowchart when it is not obvious what they do.

Creating Flowcharts

There are many tools you can use to create flow charts.

Pseudo Code

- Pseudo code is an ordered version of your code written for human understanding rather than machine understanding.
- There is no one set way to write pseudo code.
- Good pseudo code should:
    - not be in a specific coding language
    - draft the structure of your code
    - be understandable to humans

e.g. pseudo code

```
if number <= 10 then
  ouput small number sentence
```

python code

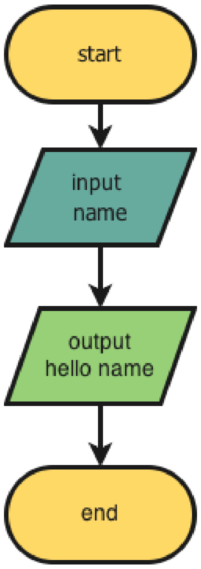
```
if number <=10:
  print("That's a small number!")
```

Note: Pseudo code may seem unnecessary but it is very useful to draft bits of code without worrying about the specifics of making it understandable to a computer.

Example 1

- This program asks the user their name then says "Hello [Name]":

flowchart for Example 1



Note: This flow chart only shows **ovals** and **parallelograms** because the code only has a **start** and **end** and one **input** and one **output**.

pseudo code for Example 1

```
input username
  output "Hello username"
```

Python code for Example 1

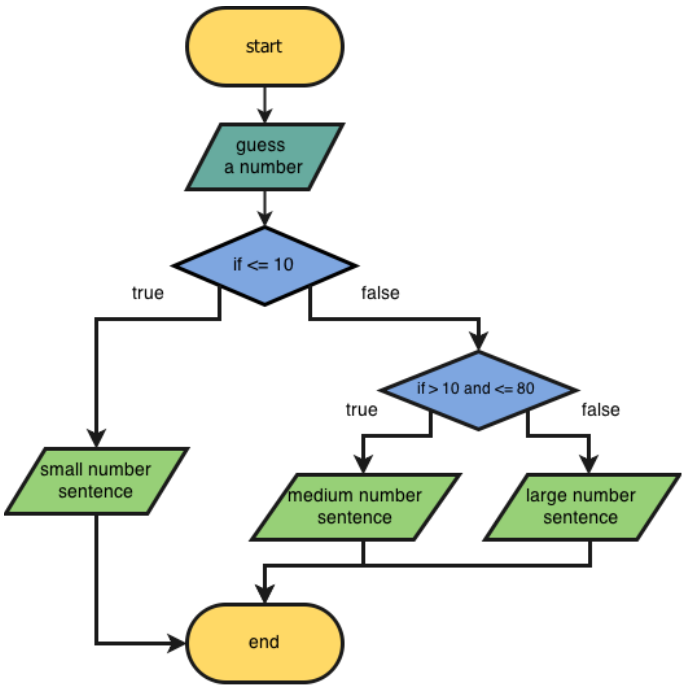
```
name= input("What is your name?")
  print("Hello "+ name)
```

Example 2

Description for Example 2

- This program asks the user to "Pick a number:" then prints 1 of 3 different outputs based on how big the number is.

flow chart for Example 2



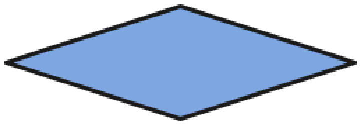
pseudo code for Example 2

```
number = input "Pick a number:"
if number <= 10 then
  output small number sentence
if number > 10 and <= 80
  output medium number sentence
else
  output large number sentence
```

Python code for Example 2

```
number=int(input("Pick a number: "))
if number <=10:
  print("That's a small number!")
elif number >10 and number <=80:
  print("That's a medium sized number")
else:
  print("Wow that's big!")
```

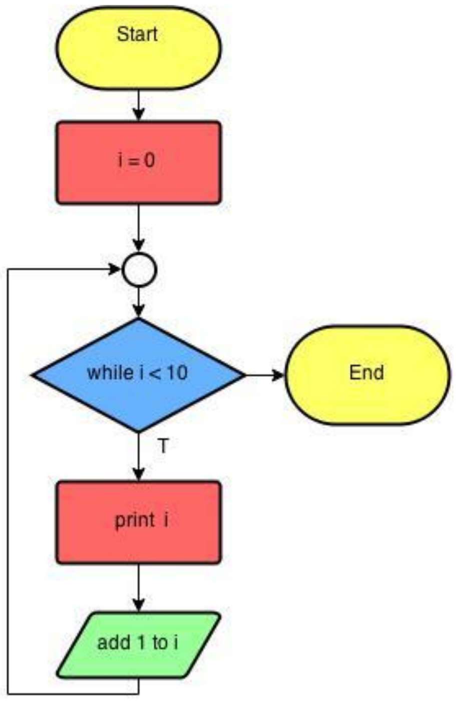
- The common way to represent **loops** is with a diamond symbol:



- This is the same way we show a decision (**if, else if/elif** etc).
- **For loops** normally repeat a nested block of code a set number of times or for a set range of data.
- **while loops** repeat a set block of code while a condition is true.
- The diamond symbol is used with **loops** to show the way the computer repeats a nested block of code then moves on to the next step.
- One of the easiest ways to think about it is that there are normally two paths the computer can go down when it reaches a loop:
    - the code in the loop
    - the code after the loop

For example:

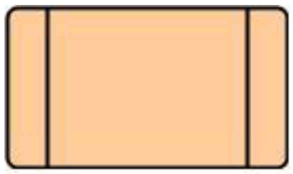
```
i=0
whilei<10:
  print(i)
  i+=1
```



Note: In this example there is no code after the loop so we represent this with the **end** symbol.

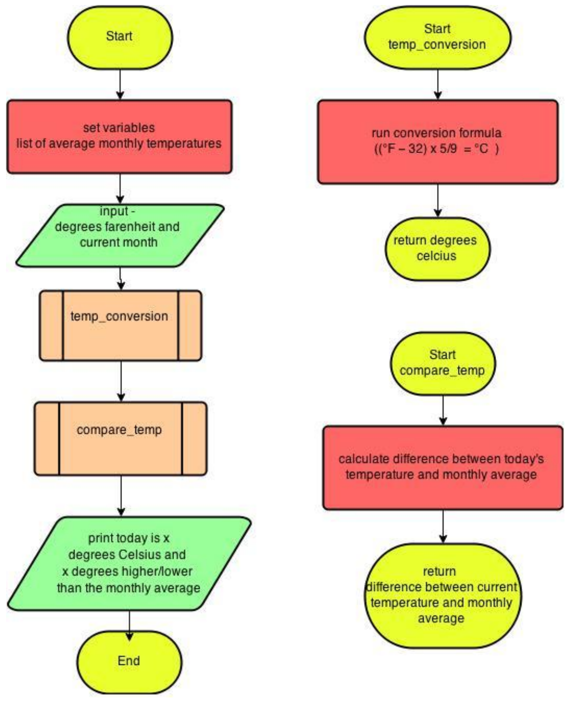
Functions In Flowcharts

- Another type of symbol that is important for more advanced programs is the symbol used for **functions**.
- **Functions** are sometimes also known as **modules** or **pre-defined processes**; they represent a completed block of code that can be called form other parts of a program.
- **Functions** are not taught in the Level 1 Python course and may not be required for basic programs.
- (They are not required for Level 1 NCEA in New Zealand but they are for Level 2.)
- **Functions** are represented with this symbol:



- Representing the contents of **functions** in flowcharts.
- A common way to represent the contents of functions is with separate unconnected flowcharts.

- The name of the function is normally added to the start symbol in these unconnected flowcharts.
- There may be times when you do not need to show the contents of every function in your flowchart.
- The example below shows a rough diagram of a program that makes use of functions that convert Farenheit to Celcius and compare the current temperate to the monthly average. This is a rough example only to show the idea of how you can represent functions in flowcharts; often the functions will be far more complex than the examples below.


Start
set variables
list of average monthly temperatures
input - degrees farenheit and current month
temp_conversion
compare_temp
print today is x degrees Celsius and x degrees higher/lower than the monthly average
End
Start temp_conversion
run conversion formula ((°F – 32) x 5/9 = °C)
return degrees celcius
Start compare_temp
calculate difference between today's temperature and monthly average
return difference between current temperature and monthly average

DECISION TABLE

- A decision table is used to represent conditional logic by creating a list of tasks depicting business level rules. Decision tables can be used when there is a consistent number of a condition that must be evaluated and assigned a specific set of actions to be used when the conditions are finally met.
- Decision tables are fairly similar to decision trees except for the fact that decision tables will always have the same number of conditions that need to be evaluated and actions that must be performed even if the set of branches being analyzed is resolved to true. A

decision tree, on the other hand, can have one branch with more conditions that need to be evaluated than other branches on the tree.

- Decision tables are a concise visual representation for specifying which actions to perform depending on given conditions. They are algorithms whose output is a set of actions. The information expressed in decision tables could also be represented as decision trees or in a programming language as a series of if-then-else and switch-case statements.

Example

- The limited-entry decision table is the simplest to describe. The condition alternatives are simple Boolean values, and the action entries are check-marks, representing which of the actions in a given column are to be performed.
- A technical support company writes a decision table to diagnose printer problems based upon symptoms described to them over the phone from their clients.
- The following is a balanced decision table (created by Systems Made Simple).

Printer troubleshooter

		Rules							
Conditions	Printer prints	No	No	No	No	Yes	Yes	Yes	Yes
	A red light is flashing	Yes	Yes	No	No	Yes	Yes	No	No
	Printer is recognized by computer	No	Yes	No	Yes	No	Yes	No	Yes
Actions	Check the power cable			✓					—
	Check the printer-computer cable	✓		✓					—
	Ensure printer software is installed	✓		✓		✓		✓	—

	Check/replace ink	✓	✓				✓		—
	Check for paper jam		✓		✓				—

➢ Of course, this is just a simple example (and it does not necessarily correspond to the reality of printer troubleshooting), but even so, it demonstrates how decision tables can scale to several conditions with many possibilities.

ALGORITHMS

➢ Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language. From the data structure point of view, following are some important categories of algorithms –

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

➢ Not all procedures can be called an algorithm. An algorithm should have the following characteristics –

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps , and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm?

- There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.
- As we know that all programming languages share basic code constructs like loops , flow-control , etc. These common constructs can be used to write an algorithm.
- We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.
- Example Let's try to learn algorithm-writing by using an example.
  - **Problem − Design an algorithm to add two numbers and display the result.**

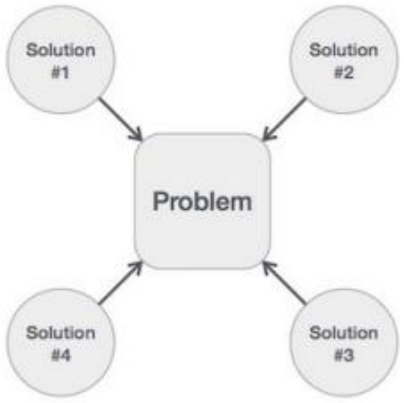
```
Step 1 − START
Step 2 − declare three integers a, b & c
Step 3 − define values of a & b
Step 4 − add values of a & b
Step 5 − store output of step 4 to c
Step 6 − print c
Step 7 − STOP
```

  - Algorithms tell the programmers how to code the program.
- Alternatively, the algorithm can be written as –

```
Step 1 − START ADD
Step 2 − get values of a & b
Step 3 − c ← a + b
Step 4 − display c
Step 5 − STOP
```

- In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

- Writing step numbers, is optional.
- We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



- Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

STRUCTURED PROGRAMMING

- Structured programming is concerned with the structures used in a computer program. Generally, structures of computer program comprise decisions, sequences, and loops. The **decision structures** are used for conditional execution of statements (for example, 'if statement). The **sequence structures** are used for the sequentially executed statements. The **loop structures** are used for performing some repetitive tasks in the program.
- Structured programming forces a logical structure in the program to be written in an efficient and understandable manner. The purpose of structured programming is to make the software code easy to modify when required. Some languages such as Ada, Pascal, and dBase are designed with features that implement the logical program structure in the software code. Primarily, the structured programming focuses on reducing the following statements from the program.
    - 'GOTO' statements.
    - 'Break' or 'Continue' outside the loops.
    - Multiple exit points to a function, procedure, or subroutine. For example, multiple 'Return' statements should not be used.
    - Multiple entry points to a function, procedure, or a subroutine.

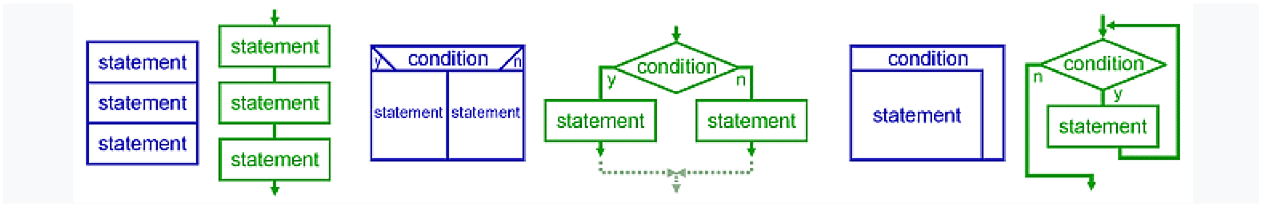
- Structured programming generally makes use of top-down design because program structure is divided into separate subsections. A defined function or set of similar functions is kept separately. Due to this separation of functions, they are easily loaded in the memory. In addition, these functions can be reused in one or more programs. Each module is tested individually. After testing, they are integrated with other modules to achieve an overall program structure. Note that a key characteristic of a structured statement is the presence of single entry and single exit point. This characteristic implies that during execution, a structured statement starts from one defined point and terminates at another defined point.
- Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of the structured control flow constructs of selection (if/then/else) and repetition (while and for), block structures, and subroutines in contrast to using simple tests and jumps such as the go to statement, which can lead to "spaghetti code" that is potentially difficult to follow and maintain.
- It emerged in the late 1950s with the appearance of the ALGOL 58 and ALGOL 60programming languages, with the latter including support for block structures. Contributing factors to its popularity and widespread acceptance, at first in academia and later among practitioners, include the discovery of what is now known as the structured program theoremin 1966, and the publication of the influential "Go To Statement Considered Harmful" open letter in 1968 by Dutch computer scientist Edsger W. Dijkstra, who coined the term "structured programming".
- Structured programming is most frequently used with deviations that allow for clearer programs in some particular cases, such as when exception handling has to be performed.

Control structures

Following the structured program theorem, all programs are seen as composed of control structures:

- "Sequence"; ordered statements or subroutines executed in sequence.
- "Selection"; one or a number of statements is executed depending on the state of the program. This is usually expressed with keywords such as if..then..else..endif.
- "Iteration"; a statement or block is executed until the program reaches a certain state, or operations have been applied to every element of a collection. This is usually expressed with keywords such as while, repeat, for or do..until. Often it is recommended that each loop should only have one entry point (and in the original structural programming, also only one exit point, and a few languages enforce this).

- "Recursion"; a statement is executed by repeatedly calling itself until termination conditions are met. While similar in practice to iterative loops, recursive loops may be more computationally efficient, and are implemented differently as a cascading stack.



Graphical representation of the three basic patterns — sequence, selection, and repetition — using NS diagrams (blue) and flow charts (green).

Subroutines

- Subroutines; callable units such as procedures, functions, methods, or subprograms are used to allow a sequence to be referred to by a single statement.

Blocks

- Blocks are used to enable groups of statements to be treated as if they were one statement. Block-structured languages have a syntax for enclosing structures in some formal way, such as an if-statement bracketed by if..fi as in ALGOL 68, or a code section bracketed by BEGIN..END, as in PL/I and Pascal, whitespace indentation as in Python - or the curly braces {...} of C and many later languages.

PROGRAMMING METHODOLOGIES

- Programming refers to the method of creating a sequence of instructions to enable the computer to perform a task. It is done by developing logic and then writing instructions in a programming language. A program can be written using various programming practices available. A **programming practice** refers to the way of writing a program and is used along with coding style guidelines. Some of the commonly used programming practices include top-down programming, bottom-up programming, structured programming, and information hiding.

Top-down Programming

- Top-down programming focuses on the use of modules. It is therefore also known as modular programming. The program is broken up into small modules so that it is easy to trace a particular segment of code in the software program. The modules at the top level are those that perform general tasks and proceed to other modules to perform a particular task. Each module is based on the functionality of its functions and procedures. In this approach, programming begins from the top level of hierarchy and progresses towards the lower levels. The implementation of modules starts with the main module. After the implementation of the main module, the subordinate modules are implemented and the process follows in this way. In top-down programming, there is a risk of implementing data structures as the modules are dependent on each other and they nave to share one or more functions and procedures. In this way, the functions and procedures are globally visible. In addition to modules, the top-down programming uses sequences and the nested levels of commands.

Disadvantages of top-down programming

- Top-down programming complicates testing. Noting executable exists until the very late in the development, so in order to test what has been done so far, one must write stubs .
- Furthermore, top-down programming tends to generate modules that are very specific to the application that is being written, thus not very reusable.
- But the main disadvantage of top-down programming is that all decisions made from the start of the project depend directly or indirectly on the high-level specification of the application. It is a well-known fact that this specification tends to change over time. When that happens, there is a great risk that large parts of the application need to be rewritten.

How does top-down programming work?

- Top-down programming tends to generate modules that are based on functionality, usually in the form of functions or procedures. Typically, the high-level specification of the system states functionality. This high-level description is then refined to be a sequence or a loop of simpler functions or procedures, that are then themselves refined, etc.
- In this style of programming, there is a great risk that implementation details of many data structures have to be shared between modules, and thus globally exposed. This in turn makes it tempting for other modules to use these implementation details, thereby creating unwanted dependencies between different parts of the application.

Bottom-up Programming

- Bottom-up programming refers to the style of programming where an application is constructed with the description of modules. The description begins at the bottom of the hierarchy of modules and progresses through higher levels until it reaches the top. Bottom-up programming is just the opposite of top-down programming. Here, the program modules are more general and reusable than top-down programming.

- It is easier to construct functions in bottom-up manner. This is because bottom-up programming requires a way of passing complicated arguments between functions. It takes the form of constructing abstract data types in languages such as C++ or Java, which can be used to implement an entire class of applications and not only the one that is to be written. It therefore becomes easier to add new features in a bottom-up approach than in a top-down programming approach.

Advantages of bottom-up programming

- Bottom-up programming has several advantages over top-down programming .
- Testing is simplified since no stubs are needed. While it might be necessary to write test functions, these are simpler to write than stubs, and sometimes not necessary at all, in particular if one uses an interactive programming environment such as Common Lisp or GDB.
- Pieces of programs written bottom-up tend to be more general, and thus more reusable, than pieces of programs written top-down. In fact, one can argue that the purpose bottom-up programming is to create an application-specific language . Such a language is suitable for implementing an entire class of applications, not only the one that is to be written. This fact greatly simplifies maintenance, in particular adding new features to the application. It also makes it possible to delay the final decision concerning the exact functionality of the application. Being able to delay this decision makes it less likely that the client has changed his or her mind between the establishment of the specifications of the application and its implementation.

How does bottom-up programming work?

- In a language such as C or Java, bottom-up programming takes the form of constructing abstract data types from primitives of the language or from existing abstract data types.
- In Common Lisp, in addition to constructing abstract data types, it is common to build *functions* bottom-up from simpler functions, and to use macros to construct new *special forms* from simpler ones.
- One may ask why it is not possible to construct functions and special forms bottom-up in other languages than Common Lisp. Constructing functions bottom-up requires a way of passing complicated arguments between functions. Common Lisp uses *lists* for such argument passing. Lists are flexible standardized data structures in the language. In other languages, data structures would have to be defined for such parameter passing only, making it more like an abstract data type than just a function. With respect to special forms, only the two-level syntax of Common Lisp allows a flexible enough macro facility for bottom-up programming of special forms.

POSSIBLE QUESTIONS

UNIT – II

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. Define Flowchart
2. What is meant by Decision table?
3. List the types of programming methodologies
4. Define Algorithm
5. List the characteristics of Algorithms

PART – C (6 MARKS)

1. Discuss in detail about Structured Programming.
2. Explain about Top down Programming Methodology
3. Explain about Bottom up Programming Methodology
4. Discuss in detail about Decision table
5. Explain the process of Flowcharts with suitable examples
6. Explain how to write an Algorithm
7. Explain the process of Functions in Flowchart
8. Difference between Top down and Bottom up Methodologies
9. Demonstrate the program to find the largest of three numbers with Flowcharts, pseudo code and Python
10. Describe Algorithms and its Characteristics

KARPAGAM ACADEMY OF HIGHER EDUCATION
Coimbatore – 641 021.
(For the Candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT
UNIT - II : (Objective Type Multiple choice Questions each Question carries one Mark)
PROGRAMMING IN PYTHON [16CAU501B]
PART - A (Online Examination)

Questions	**Opt1**	**Opt2**	**Opt3**	**Opt4**	**Key**
Method which uses a list of well defined inst	Program	Flowchart	Algorithm	Process	Algorithm
The chart that contains only function flow an	flowchart	Structure chart	Algorithm	Process	chart
The sequence logic will not be used while	Accepting input from user	Giving output to the user	Comparing two sets of data	Adding two numbers	Giving output to the user
Flowcharts and Algorithms are used for	Better Programming	Easy testing and Debugging	Efficient Coding	All	All
An Algorithm represented in the form of programming languages is	Flowchart	Pseudo code	Program	Process	Program
Which of the following is not a principle of structured programming?	Design the program in top-down manner	Write each program module as a series of control structures	Code the program so that it runs correctly without testing	Use good programming	Design the program in top-down manner
Which of the following is not a basic control structure	The process	The decision	The Loop	The sequential	The process
Which of the following is a pictorial representation of an algorithm?	Pseudo code	Program	Flowchart	Algorithm	Flowchart

Which of the following symbol in a flowchart are used to indicate all arithmetic processes of adding, subtracting, multiplying and dividing ?	Input/output	terminal	Processing	Decision	Processing
A flowchart that outlines the main segments of program is called as	Micro flowchart	Macro flowchart	Flowchart	Algorithm	Macro flowchart
A flowchart that outlines with all detail is called as	Micro flowchart	Macro flowchart	Flowchart	Algorithm	Micro flowchart
Pseudo code is also known as	Program Design Language	Hardware Language	Software Language	Algorithm	Program Design Language
Pseudo code emphasizes on	Development	Coding	Design	Debugging	Design
In which of the following pseudo code instructions are written in the order or sequence in which they are to be performed?	Selection Logic	Sequence Logic	Iteration Logic	Looping Logic	Sequence Logic
Which of the following logic is used to produce loops in program logic when one or more instruction may be executed several times depending on some conditions?	Selection Logic	Sequence Logic	Iteration Logic	Looping Logic	Iteration Logic
Selection logic also called as	Sequence Logic	Iteration Logic	Looping Logic	Decision Logic	Decision Logic
Which of the following program planning tool allows the programmers to plan program logic by writing program instruction in an ordinary language?	Flowchart	Pseudo code	Program	Looping	Pseudo code
Algorithm is	step by step execution of program	Object file	Executable file	Source file	step by step execution of program
Kite box in flow chart is used for	Connecter	Decision	Statement	Looping	Decision
Which of the following is not a characteristic of good algorithm?	Precise	Ambiguous	Finite number of steps	Logical flow of control	Finite number of steps

Diagrammatic representation of an algorithm is	Flowchart	Data flow Diagram	Algorithm design	Pseudo code	Flowchart
What symbol is used to represent output in a flowchart?	Square	Circle	Parallelogram	Triangle	Parallelogram
What is the standard terminal symbol for flowchart?	Square	Circle	Parallelogram	Triangle	Circle
The following pseudo code is an example of _______ structure: Get number While number is positive Add to sum	Sequence	Decision	Loop	Nested	Loop
The following pseudo code is an example of ______structure:Get number Get another number If first number is greater than second then Print first number Else print second number	Sequence	Decision	Loop	Nested	Decision
The following pseudo code is an example of ________structure: Get number Get another number Multiply numbers Print result	Sequence	Decision	Loop	Nested	Sequence
structured program can be easily broken down into routines or _______that can be assigned to any number of programmers	Segments	Modules	Units	Sequences	Modules
A Decision table is ____________	represents the information flow	documents rules, that select one or more actions, based on one or more conditions, from a set of possible conditions	gets an accurate picture of the system	shows the decision paths	documents rules, that select one or more actions, based on one or more conditions, from a set of possible conditions

The Structure Chart is ________	a document of what has to be accomplished	a hierarchical partitioning of the program	a statement of information - processing requirements	shows the decision paths	a hierarchical partitioning of the program
After a programmer plans the logic of a program ,she /he will next	Understand the problem	Translate the program	Test the program	Code the program	Code the program
In a case structure of the loop, the loop body continues to execute as long as the answer to the controlling question is yes, or true.	Else	Then	Default	Loop	Else
Which of the following statement cause program control to end up almost anywhere in the program?	go to	for	while	do while	go to
Which of the following statement allows us to make a decision from the number of choices?	break	Switch	for	go to	Switch
Which of the following keyword is followed by an integer or character constant?	switch	case	for	void	case
Which of the following enhances the versatility of the computer to perform a set of instructions repeatedly?	Function	Loop	header files	statement	Loop
Which of the following contains parenthesis after the „while" loop?	Condition	statement	count	value	Condition
The condition being tested within the loop may be relational or relational or logical operations	while	switch	break	continue	while
The three things inside the for loop are separated by	colon	comma	semicolon	hyphen	semicolon
Which of the following statement associated with an „if"?	switch	goto	break	do while	break

„do while" loop is useful when we want that statement within the loop must be executed	Only Once	At least Once	More than once	two	At least Once
The mechanism used to convey information to the function is the	Argument	commands	loops	statements	Argument
Which of the following can be replaced by if	switch	while	continue	for	switch
A _________________ is essentially the breaking down of a system to gain insight into its compositional sub-systems in a reverse engineering fashion.	bottom-up approach	program	top-down approach	down top approach	top-down approac h
A __________________ is the piecing together of systems to give rise to more complex systems, thus making the original systems sub-systems of the emergent system.	bottom-up approach	program	top-down approach	down top approach	bottom-up approach
__________ provide internal documentation of a program.	switch	comments	count	value	comments
_________ makes the statements clear and readable	switch	comments	count	indentation	indentation
___________ refers to its presentation style so that the program becomes more readable and presentable	good program	program	program representation	data representation	program representation
________________ is concerned with the structures used in a computer program.	Programming	Program	Structured programming	Object Oriented Programming	Structured programming
___________________ tends to generate modules that are based on functionality, usually in the form of functions or	Structured programming	Top-down programming	bottom up programming	Object Oriented Programming	Top-down programming
. A ________________ refers to the way of writing a program and is used along with coding style guidelines	Structured programming	Top-down programming	bottom up programming	programming practice	programming practice
Algorithm to search an item in a data structure	Search	Sort	Insert	Update	Search

Algorithm to sort items in a certain order	Search	Sort	Insert	Update	Sort
Algorithm to insert item in a data structure.	Search	Sort	Insert	Update	Insert
Algorithm to delete an existing item from a data structure.	delete	Sort	Insert	Update	delete
Algorithm to update an existing item in a data structure	Search	Sort	Insert	Update	Update
Each of its steps , and their inputs/outputs should be clear and must lead to only one meaning. Which means?	Output	Input	Unambiguous	Finiteness	Unambiguous
Algorithms must terminate after a finite number of steps	Output	Input	Unambiguous	Finiteness	Finiteness
____________ is an algorithm should have step-by-step directions, which should be independent of any programming code.	Output	Independent	Unambiguous	Finiteness	Independent
__________ can result in inefficient, unstructured code known as 'spaghetti code'	Algorithm	planning	Poor planning	good planning	Poor planning
A standard way to plan code is with ______________	Algorithm	planning	Poor planning	flowcharts	flowcharts

UNIT – III

SYLLABUS

Overview of Programming: Structure of a Python Program - Elements of Python.

OVERVIEW OF PROGRAMMING

- **Programming** is the process of taking an algorithm and encoding it into a notation, a programming language, so that it can be executed by a computer. Although many programming languages and many different types of computers exist, the important first step is the need to have the solution. Without an algorithm there can be no program.
- Python is one of those rare languages which can claim to be both simple and powerful. You will find that you will be pleasantly surprised on how easy it is to concentrate on the solution to the problem rather than the syntax and structure of the language you are programming in.
- The official introduction to Python is
- Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.
- I will discuss most of these features in more detail in the next section.

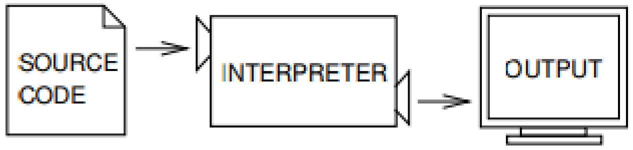
STRUCTURE OF PYTHON

The Python programming Language

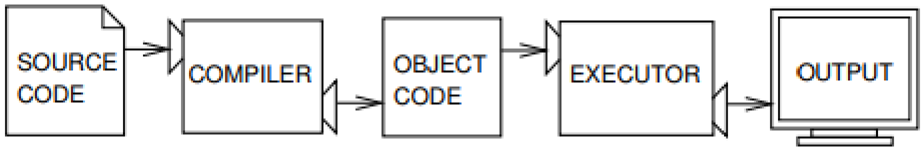
- The programming language you will be learning is Python. Python is an example of a high-level language; other high-level languages you might have heard of are C, C++, Perl, and Java.
- As you might infer from the name “high-level language,” there are also lowlevel languages, sometimes referred to as “machine languages” or “assembly 2 The way of the program languages.” Loosely speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages.
- But the advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level

languages are portable, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.

- Due to these advantages, almost all programs are written in high-level languages. Low-level languages are used only for a few specialized applications. Two kinds of programs process high-level languages into low-level languages: interpreters and compilers. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.



- A compiler reads the program and translates it completely before the program starts running. In this case, the high-level program is called the source code, and the translated program is called the object code or the executable. Once a program is compiled, you can execute it repeatedly without further translation.



- Python is considered an interpreted language because Python programs are executed by an interpreter. There are two ways to use the interpreter: command line mode and script mode. In command-line mode, you type Python programs and the interpreter prints the result:

```
$ python
Python 2.4.1 (#1, Apr 29 2005, 00:28:56)
Type "help", "copyright", "credits" or "license" for more information.
>>> print 1 + 1
2
```

- The first line of this example is the command that starts the Python interpreter. The next two lines are messages from the interpreter. The third line starts with >>>, which is the prompt the interpreter uses to indicate that it is ready. We typed print 1 + 1, and the interpreter replied 2. Alternatively, you can write a program in a file and use the interpreter to execute the contents of the file. Such a file is called a script. For example, we used a text editor to create a file named latoya.py with the following contents:

```
print 1 + 1
```

- By convention, files that contain Python programs have names that end with .py.
- To execute the program, we have to tell the interpreter the name of the script:

```
$ python latoya.py 2
```

- In other development environments, the details of executing programs may differ. Also, most programs are more interesting than this one.
- Most of the examples in this book are executed on the command line. Working on the command line is convenient for program development and testing, because you can type programs and execute them immediately. Once you have a working program, you should store it in a script so you can execute or modify it in the future.

FEATURES OF PYTHON

- **Simple**
  - Python is a simple and minimalistic language. Reading a good Python program feels almost like reading English, although very strict English! This pseudo-code nature of Python is one of its greatest strengths. It allows you to concentrate on the solution to the problem rather than the language itself.
- **Easy to Learn**
  - As you will see, Python is extremely easy to get started with. Python has an extraordinarily simple syntax, as already mentioned.
- **Free and Open Source**
  - Python is an example of a FLOSS (Free/LibrÃ© and Open Source Software). In simple terms, you can freely distribute copies of this software, read it's source code, make changes to it, use pieces of it in new free programs, and that you know you can do these things. FLOSS is based on the concept of a community which shares knowledge. This is one of the reasons why Python is so good - it has been created and is constantly improved by a community who just want to see a better Python.
- **High-level Language**
  - When you write programs in Python, you never need to bother about the low-level details such as managing the memory used by your program, etc.
- **Portable**
  - Due to its open-source nature, Python has been ported (i.e. changed to make it work on) to many platforms. All your Python programs can work on any of these platforms without requiring any changes at all if you are careful enough to avoid any system-dependent features.
  - You can use Python on Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion,

Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and even PocketPC !

- **Interpreted**
  - This requires a bit of explanation.
  - A program written in a compiled language like C or C++ is converted from the source language i.e. C or C++ into a language that is spoken by your computer (binary code i.e. 0s and 1s) using a compiler with various flags and options. When you run the program, the linker/loader software copies the program from hard disk to memory and starts running it.
  - Python, on the other hand, does not need compilation to binary. You just *run* the program directly from the source code. Internally, Python converts the source code into an intermediate form called bytecodes and then translates this into the native language of your computer and then runs it. All this, actually, makes using Python much easier since you don't have to worry about compiling the program, making sure that the proper libraries are linked and loaded, etc, etc. This also makes your Python programs much more portable, since you can just copy your Python program onto another computer and it just works!
- **Object Oriented**
  - Python supports procedure-oriented programming as well as object-oriented programming. In *procedure-oriented* languages, the program is built around procedures or functions which are nothing but reusable pieces of programs. In *object-oriented* languages, the program is built around objects which combine data and functionality. Python has a very powerful but simplistic way of doing OOP, especially when compared to big languages like C++ or Java.
- **Extensible**
  - If you need a critical piece of code to run very fast or want to have some piece of algorithm not to be open, you can code that part of your program in C or C++ and then use them from your Python program.
- **Embeddable**
  - You can embed Python within your C/C++ programs to give 'scripting' capabilities for your program's users.
- **Extensive Libraries**
  - The Python Standard Library is huge indeed. It can help you do various things involving regular expressions, documentation generation, unit testing, threading, databases, web browsers, CGI, ftp, email, XML, XML-RPC, HTML, WAV files, cryptography, GUI (graphical user interfaces), Tk, and other system-dependent stuff. Remember, all this is always available wherever Python is installed. This is called the 'Batteries Included' philosophy of Python.
  - Besides, the standard library, there are various other high-quality libraries such as wxPython, Twisted, Python Imaging Library and many more.

ELEMENTS OF PYTHON

- Python is a high level scripting language with object oriented features.

1. **Syntax**

- Python programs can be written using any text editor and should have the extension .py. Python programs do not have a required first or last line, but can be given the location of python as their first line: #!/usr/bin/python and become executable. Otherwise, python programs can be run from a command prompt by typing python file.py. There are no braces {} or semicolons ; in python. It is a very high level language. Instead of braces, blocks are identified by having the same indentation.
- Programming languages are traditionally introduced with a trivial example that does nothing more than write the words *Hello world* on the screen. The intention is to illustrate the essential components of the language, and familiarise the user with the details of entering and running programs.
- The good news is that in Python this is an extremely simple program:

```
print "Hello world"
```

- he program is self-explanatory: note that we indicated what we wanted to print to the screen by enclosing it in quotation marks.

2. **Interpreters, modules, and a more interesting program**

- There are two ways of using Python: either using its interactive interpreter as you have just done, or by writing modules. The interpreter is useful for small snippets of programs we want to try out. In general, all the examples you see in this book can be typed at the interactive prompt (»>). You should get into the habit of trying things out at the prompt: you can do no harm, and it is a good way of experimenting.
- However, working interactively has the serious drawback that you cannot save your work. When you exit the interactive interpreter everything you have done is lost. If you want to write a longer program you create a module. This is just a text file containing a list of Python instructions. When the module is run Python simply reads through it one line after another, as though it had been typed at the interactive prompt.
- When you start up IDLE, you should see the Python interactive interpreter. You can always recognise an interpreter window by the »> prompt whereas new module windows are empty. IDLE only ever creates one interpreter window: if you close it and need to get the interpreter back, select Python shell from the Run menu. You can have multiple module windows open

simultaneously: as described in Chapter 2 each one is really an editor which allows you to enter and modify your program code (because of this, they will often be referred to in this handbook as editor windows). To get a new empty module (editor) window select New window in the File menu.

- Here is an example of a complete Python module. Type it into an editor window and run it by choosing Run from the Run menu (or press the F5 key on your keyboard)3.1.

```
print "Please give a number: "
a = input()
print "And another: "
b = input()
print "The sum of these numbers is: "
print a + b
```

- If you get errors then check through your copy for small mistakes like missing punctuation marks. Having run the program it should be apparent how it works. The only thing which might not be obvious are the lines with input(). input() is a function which allows a user to type in a number and returns what they enter for use in the rest of the program: in this case the inputs are stored in a and b.
- If you are writing a module and you want to save your work, do so by selecting Save from the File menu then type a name for your program in the box. The name you choose should indicate what the program does and consist only of letters, numbers, and ``_" the underscore character. The name must end with a .py so that it is recognised as a Python module, e.g. prog.py. Furthermore, do NOT use spaces in filenames or directory (folder) names.

3. Variables

- **Names and Assignment**
- we used variables for the first time: a and b in the example. Variables are used to store data; in simple terms they are much like variables in algebra and, as mathematically-literate students, we hope you will find the programming equivalent fairly intuitive.
- Variables have names like a and b above, or x or fred or z1. Where relevant you should give your variables a descriptive name, such as firstname or height 3.2. Variable names must start with a letter and then may consist only of alphanumeric characters (i.e. letters and numbers) and the underscore character, ``_". There are some reserved words which you cannot use because Python uses them for other things; these are listed in Appendix B.
- We assign values to variables and then, whenever we refer to a variable later in the program, Python replaces its name with the value we assigned to it. This is best illustrated by a simple example:

```
>>> x = 5
>>> print x
5
```

- You assign by putting the variable name on the left, followed by a single =, followed by what is to be stored. To draw an analogy, you can think of variables as named boxes. What we have done above is to label a box with an ``x'', and then put the number 5 in that box.
- There are some differences between the syntax 3.3 of Python and normal algebra which are important. Assignment statements read right to left only. x = 5 is fine, but 5 = x doesn't make sense to Python, which will report a SyntaxError. If you like, you can think of the equals sign as an arrow pointing from the number on the right, to the variable name on the left: $x \leftarrow 5$ and read the expression as ``assign 5 tox'' (or, if you prefer, as ``x becomes 5''). However, we can still do many of things you might do in algebra, like:

```
>>> a = b = c = 0
```

- Reading the above right to left we have: ``assign 0 to c, assign c to b, assign b to a''.

```
>>> print a, b, c
0 0 0
```

- There are also statements that are alegbraically nonsense, that are perfectly sensible to Python (and indeed to most other programming languages). The most common example is incrementing a variable:

```
>>> i = 2
>>> i = i + 1
>>> print i
3
```

- The second line in this example is not possible in maths, but makes sense in Python if you think of the equals as an arrow pointing from right to left. To describe the statement in words: on the right-hand side we have looked at what is in the box labelled i, added 1 to it, then stored the result back in the same box

4. Types

- Your variables need not be numeric. There are several types. The most useful are described below:
- **Integer:** Any whole number:

```
>>> myinteger = 0
>>> myinteger = 15
>>> myinteger = -23
>>> myinteger = 2378
```

- **Float**: A floating point number, i.e. a non-integer.

```
>>> myfloat = 0.1
>>> myfloat = 2.0
>>> myfloat = 3.14159256
>>> myfloat = 1.6e-19
>>> myfloat = 3e8
```

- Note that although 2 is an integer, by writing it as 2.0 we indicate that we want it stored as a float, with the precision that entails.3.4 The last examples use exponentials, and in maths would be written 1.6×10^{-19} and 3×10^{8}. If the number is given in exponential form it is stored with the precision of floating point whether or not it is a whole number.
- **String**: A string or sequence of characters that can be printed on your screen. They must be enclosed in *either* single quotes *or* double quotes--not a mixture of the two, e.g.

```
>>> mystring = "Here is a string"
>>> mystring = 'Here is another'
```

- **Arrays and Lists**: These are types which contain more than one element, analogous to vectors and matrices in mathematics. Their discussion is deferred until Section 3.10 ``Arrays''. For the time being, it is sufficient to know that a list is written by enclosing it in square brackets as follows: mylist = [1, 2, 3, 5]
- If you are not sure what type a variable is, you can use the type() function to inspect it:

```
>>> type(mystring)
<type 'str'>
```

- 'str' tells you it is a string. You might also get <type 'int'> (integer) and <type 'float'> (float) [3.5].
- **Tuple:**A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.
- **Example**
- **Create a Tuple:**

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

- **Example**
- **You cannot change values in a tuple:**

```
thistuple = ("apple", "banana", "cherry")
thistuple[1] = "blackcurrant" # test changeability
print(thistuple)
```

5. Input and output

- **Computer** programs generally involve interaction with the user. The is called input and output. Output involves printing things to the screen (and, as we shall see later, it also involves writing data to files, sending plots to a printer, etc). We have already seen one way of getting input--the input() function in

Section 3.2. Functions will be discussed in more detail in Section 3.9, but for now we can use the input() function to get numbers (and only numbers) from the keyboard.

- You can put a string between the parentheses of input() to give the user a prompt. Hence the example in Section 3.2 could be rewritten as follows:

```
a = input("Please give a number: ")
b = input("And another: ")
print "The sum of these numbers is:", a + b
```

- [Note that in this mode of operation the input() function is actually doing output as well as input!]
- The print command can print several things, which we separate with a comma, as above. If the print command is asked to print more than one thing, separated by commas, it separates them with a space. You can also concatenate (join) two strings using the + operator (note that no spaces are inserted between concatenated strings):

```
>>> x = "Spanish Inquisition"
>>> print "Nobody expects the" + x
Nobody expects theSpanish Inquisition
```

- input() can read in numbers (integers and floats) only. If you want to read in a string (a word or sentence for instance) from the keyboard, then you should use the raw_input() function; for example:

```
>>> name = raw_input("Please tell me your name: ")
```

6. Arithmetic

- The programs you write will nearly always use numbers. Python can manipulate numbers in much the same way as a calculator (as well as in the much more complex and powerful ways you'll use later). Basic arithmetic calculations are expressed using Python's (mostly obvious) arithmetic operators.

```
>>> a = 2           # Set up some variables to play with
>>> b = 5
>>> print a + b
7
>>> print a - b       # Negative numbers are displayed as expected
-3
>>> print a * b       # Multiplication is done with a *
10
>>> print a / b       # Division is with a forward slash /
0.4
>>> print a ** b       # The power operation is done with **
32
```

```
>>> print b % a       # The % operator finds the remainder of a
division
1
>>> print 4.5 % 2      # The % operator works with floats too
0.5
```

- The above session at the interactive interpreter also illustrates comments. This is explanatory text added to programs to help anyone (including yourself!) understand your programs. When Python sees the #symbol it ignores the rest of the line. Here we used comments at the interactive interpreter, which is not something one would normally do, as nothing gets saved. When writing modules you should comment your programs comprehensively, though succinctly. They should describe your program in sufficient detail so that someone who is not familiar with the details of the problem but who understands programming (though not necessarily in Python), can understand how your program works. Examples in this handbook should demonstrate good practice.
- Although you should write comments from the point of view of someone else reading your program, it is in your own interest to do it effectively. You will often come back to read a program you have written some time later. Well written comments will save you a lot of time. Furthermore, the demonstrators will be able to understand your program more quickly (and therefore mark you more quickly too!).
- The rules of precedence are much the same as with calculators. ie. Python generally evaluates expressions from left to right, but things enclosed in brackets are calculated first, followed by multiplications and divisions, followed by additions and subtractions. If in doubt add some parentheses:

```
>>> print 2 + 3 * 4
14
>>> print 2 + (3 * 4)
14
>>> print (2 + 3) * 4
20
```

- Parentheses may also be nested, in which case the innermost expressions are evaluated first; ie.

```
>>> print (2 * (3 - 1)) * 4
16
```

7. <u>An example of a for loop</u>

- In programming a loop is a statement or block of statements that is executed repeatedly. for loops are used to do something a fixed number of times (where the number is known at the start of the loop). Here is an example:

```
sumsquares = 0    # sumsquares must have a value because we increment
                  # it later.

for i in [0, 1, 2, 3, 4, 5]:
    print "i now equal to:", i
    sumsquares = sumsquares + i**2    # sumsquares incremented here
    print "sum of squares now equal to:", sumsquares
    print "------"

print "Done."
```

- The indentation of this program is essential. Copy it into an empty module. IDLE will try and help you with the indentation but Using the range function

8. <u>An example of an if test</u>

- The if statement executes a nested code block if a condition is true. Here is an example:

```
age = input("Please enter your age: ")
if age > 40:
    print "Wow! You're really old!"
```

- Copy this into an empty module and try to work out how if works. Make sure to include the colon and indent as in the example.
- What Python sees is ``if the variable age is a number greater than 40 then print a suitable comment''. As in maths the symbol `` $>$ '' means ``greater than''. The general structure of an if statement is:

```
if [condition]:
    [statements to execute if condition is true]

[rest of program]
```

9. Comparison tests and Booleans

- The [condition] is generally written in the same way as in maths. The possibilites are shown below:

Comparison	What it tests
a < b	a is less than b
a <= b	a is less than or equal to b
a > b	a is greater than b
a >= b	a is greater than or equal to b
a == b	a is equal to b
a != b	a is not equal to b
a < b < c	a is less than b, which is less than c

- The == is not a mistake. One = is used for assignment, which is different to testing for equality, so a different symbol is used. Python will complain if you mix them up (for example by doing if a = 4).
- It will often be the case that you want to execute a block of code if two or more conditions are simultaneously fulfilled. In some cases this is possible using the expression you should be familiar with from algebra: a < b < c. This tests whether a is less than b and b is also less than c.
- Sometimes you will want to do a more complex comparison. This is done using boolean operators such as and and or:

```
if x == 10 and y > z:
   print "Some statements which only get executed if"
   print "x is equal to 10 AND y is greater than z."
if x == 10 or y > z:
   print "Some statements which get executed if either
   print "x is equal to 10 OR y is greater than z"
```

- These comparisons can apply to strings too3.9. The most common way you might use string comparions is to ask a user a yes/no question3.10:

```
answer = raw_input("Evaluate again? ")

if answer == "y" or answer == "Y" or answer == "yes":
   # Do some more stuff
```

10. else and elif statements

- else and elif statements allow you to test further conditions after the condition tested by the if statement and execute alternative statements accordingly. They are an extension to the if statement and may only be used in conjunction with it.
- If you want to execute some alternative statements if an if test fails, then use an else statement as follows:

```
if [condition]:
   [Some statements executed only if [condition] is true]
else:
   [Some statements executed only if [condition] is false]

[rest of program]
```

- If the first condition is true the indented statements directly below it are executed and Python jumps to [rest of program] Otherwise the nested block below the else statement is executed, and then Python proceeds to [rest of program].
- The elif statement is used to test further conditions if (and only if) the condition tested by the if statement fails:

```
x = input("Enter a number")

if 0 <= x <= 10:
   print "That is between zero and ten inclusive"
elif 10 < x < 20:
   print "That is between ten and twenty"
else:
   print "That is outside the range zero to twenty"
```

11. while loops

- while loops are like for loops in that they are used to repeat the nested block following them a number of times. However, the number of times the block is repeated can be variable: the nested block will be repeated whilst a condition is satisfied. At the top of the while loop a condition is tested and if true, the loop is executed.

```
while [condition]:
  [statements executed if the condition is true]

[rest of program]
```

- Here is a short example:

```
i = 1
while i < 10:
   print "i equals:", i
   i = i + 1

print "i is no longer less than ten"
```

12. Using library functions

- Python contains a large library of standard functions which can be used for common programming tasks. A function is just some Python code which is seperated from the rest of the program. This has several advantages: Repeated sections of code can be re-used without rewriting them many times, making your program clearer. Furthermore, if a function is separated from the rest of the program it provides a conceptual separation for the person writing it, so they can concentrate on either the function, or the rest of the program.
- Python has some built-in functions, for example type() and range() that we have already used. These are available to any Python program.
- To use a function we call it. To do this you type its name, followed by the required parameters enclosed in parentheses. Parameters are sometimes called arguments, and are similar to arguments in mathematics. In maths, if we write $\sin(\pi/4)$, we are using the $\sin$ function with the argument $\pi/4$. Functions in computing often need more than one variable to calculate their result. These should be separated by commas, and the order you give them in is important. Refer to the discussion of the individual functions for details.
- Even if a function takes no parameters (you will see examples of such functions later), the parentheses must be included.
- However, the functions in the library are contained in separate modules, similar to the ones you have been writing and saving in the editor so far. In order to use a particular module, you must explicitly importit. This gives you access to the functions it contains.
- The most useful module for us is the math library3.11. If you want to use the functions it contains, put the line from math import * at the top of your program.
- The math functions are then accesible in the same way as the built in functions. For example, to calculate the $\sin$ and $\cos$ of $\pi/3$ we would write a module like this:

```
from math import *
mynumber = pi / 3
```

```
print sin(mynumber)
print cos(mynumber)
```

- The math module contains many functions, the most useful of which are listed below. Remember that to use them you must from math import *.

Function	Description
sqrt(x)	Returns the square root of x
exp(x)	Return e^x
log(x)	Returns the natural log, i.e. $\ln x$
log10(x)	Returns the log to the base 10 of x
sin(x)	Returns the sine of x
cos(x)	Return the cosine of x
tan(x)	Returns the tangent of x
asin(x)	Return the arc sine of x
acos(x)	Return the arc cosine of x
atan(x)	Return the arc tangent of x
fabs(x)	Return the absolute value, i.e. the modulus, of x
floor(x)	Rounds a float *down* to its integer

- The math library also contains two constants: pi, π, and e, e. These do not require parentheses (see the above example).
- Note the floor function always rounds down which can produced unexpected results! For example

```
>>> floor(-3.01)
-4.0
```

13. <u>Arrays</u>

- The elements of a list can, in principle, be of different types, e.g. [1, 3.5, "boo!"]. This is sometimes useful but, as scientists you will mostly deal with arrays. These are like lists but each element is of the same type (either integers or floats). This speeds up their mathematical manipulation by several orders of magnitude.
- Arrays are not a ``core" data type like integers, floating points and strings. In order to have access to the array type we must import the Numeric library. This is done by adding the following line to the start of every program in which arrays are used:

```
from Numeric import *
```

- When you create an array you must then explicitly tell Python you are doing so as follows:

```
>>> from Numeric import *
>>> xx = array([1, 5, 6.5, -11])
>>> print xx
[  1.    5.    6.5 -11. ]
```

- The square brackets within the parentheses are required. You can call an array anything you could call any other variable.
- The decimal point at the end of 1, 5 and -11 when they are printed indicates they are now being stored as floating point values; all the elements of an array must be of the same type and we have included 6.5 in the array so Python automatically used floats.
-

14. Making your own functions

- The libraries provide a useful range of facilities but a programmer will often want or need to write their own functions if, for example, one particular section of a program is to be used several times, or if a section forms a logically complete unit.
- Functions must be defined before they are used, so we generally put the definitions at the very top of a program. Here is a very simple example of a function definition that returns the sum of the two numbers it is passed:

```
>>> def addnumbers(x, y):
        sum = x + y
        return sum

>>> x = addnumbers(5, 10)
>>> print x
15
```

- The structure of the definition is as follows:

1. The top line must have a def statement: this consists of the word def, the name of the function, followed by parentheses containing the names of the parameters passed as they will be referred to within the function. [3.12].
2. Then an indented code block follows. This is what is executed when the function is *called*, i.e. used.

3. Finally the return statement. This is the result the function will return to the program that called it. If your function does not return a result but merely executes some statements then it is not required.

- If you change a variable within a function that change will not be reflected in the rest of the program. For example:

```
>>> def addnumbers(x, y):
        sum = x + y
        x = 1000
        return sum

>>> x = 5
>>> y = 10
>>> answer = addnumbers(x, y)
>>> print x, y, answer
5 10 15
```

- Note that although the variable x was changed in the function, that change is not reflected outside the function. This is because the function has its own private set of variables. This is done to minimise the risk of subtle errors in your program
- If you really want a change to be reflected then return a list of the new values as the result of your function. Lists can then be accessed by offset in the same way as arrays:

```
>>> def addnumbers(x, y):
        sum = x + y
        x = 100000
        return [sum, x]

>>> x = 5
>>> y = 10
>>> answer = addnumbers(x, y)
>>> print answer[0]
15
>>> print answer[1]
100000
```

15. File input and output

- So far we have taken input from the keyboard and given output to the screen. However, You may want to save the results of a calculation for later use or read in data from a file for Python to manipulate. You give Python access to a file by opening it:

```
>>> fout = open("results.dat", "w")
```

- fout is then a variable like the integers, floats and arrays we have been using so far--fout is a conventional name for an output file variable, but you are free to choose something more descriptive. The openfunction takes two parameters. First a string that is the name of the file to be accessed, and second a mode. The possible modes are as follows:

Mode	Description
r	The file is opened for reading
w	The file is opened for writing, and any file with the same name is erased--be careful!
a	The file is opened for appending--data written to it is added on at the end

- There are various ways of reading data in from a file. For example, the readline() method returns the first line the first time it is called, and then the second line the second time it is called, and so on, until the end of the file is reached when it returns an empty string:

```
>>> fin = open("input.dat", "r")
>>> fin.readline()
'10\n'
>>> fin.readline()
'20\n'
>>> fin.readline()
''
```

(The \n characters are newline characters.)

- Note the parentheses are required. If they are not included the file will not be read. They are to tell Python that you are using the readline() function--a function is always followed by parentheses, whether it takes any arguments or not.
- You can see from the example that you tell Python to use methods by adding a full stop followed by the name of the method to the variable name of the file. This syntax may seem strange3.13 but for now just use the examples below as your guide to the syntax, and don't worry about what it means.
- The contents are read in as a string but if the data is numeric you need to coerce them into either floats or integers before they are used in calculations:

```
>>> fin = open("input.dat", "r")
>>> x = fin.readline()
>>> type(x)
<type 'str'>
>>> y = float(x)
>>> type(y)
<type 'float'>
>>> print y
10.0
```

- You can also use the readlines() method (note the plural) to read in all the lines in a file in one go, returning a list:

```
>>> fin.readlines()
['This is the first line of the file.\n', 'Second line of the file\n']
```

- To output or write to a file use the write() method. It takes one parameter--the string to be written. If you want to start a new line after writing the data, add a \n character to the end:

```
>>> fout = open("output.dat", "w")
>>> fout.write("This is a test\n")
>>> fout.write("And here is another line\n")
>>> fout.close()
```

- Note that in order to commit changes to a file, you must close() files as above.
- write() must be given a string to write. Attempts to write integers, floats or arrays will fail:

```
>>> fout = open("output.dat", "w")
>>> fout.write(10)
Traceback (most recent call last):
  File "<pyshell#65>", line 1, in ?
    fout.write(10)
TypeError: argument 1 must be string or read-only character
buffer, not int
```

- You must coerce numbers into strings using the str() function:

```
>>> x = 4.1
>>> print x
4.1
>>> str(x)
'4.1'
```

- If you are trying to produce a table then you may find the t character useful. It inserts a tab character, which will move the cursor forward to the next tabstop3.14. It is an example of a non-printed character. It leaves only white space--no characters are printed as such. However, it is still a string character, and must be enclosed in quotes. For example, to print the variables a and b separated by a tab you would type:

```
print a + "\t" + b
```

- This is not a perfect method of producing a table. If you are interested in the ``right way" of doing this, then ask a demonstrator about formatted output.

16. Putting it all together

- This section shows a complete, well commented program to indicate how most of the ideas discussed so far (Variables, Arrays, Files, etc.) are used together.

Below is a rewritten version of the example in Section 3.2, which did nothing more than add two numbers together. However, the two numbers are stored in arrays, the numbers are read in by a separate function, the addition is also done by a separate function, and the result is written to a file.

```
from Numeric import *

def addnumbers(x, y):   # Declare functions first.
    sum = x + y
    return sum

def getresponse():
    # Create a two element array to store the numbers
    # the user gives. The array is one of floating
    # point numbers because we do not know in advance
    # whether the user will want to add integers or
    # floating point numbers.
    response = zeros(2, Float)
    # Put the first number in the first element of
    # the list:
    response[0] = input("Please give a number: ")
    # Put the second number in the second element:
    response[1] = input("And another: ")
    # And return the array to the rest of the program
    return response
```

```
# Allow the user to name the file. Remember this is a string
# and not a number so raw_input is used.
filename = raw_input("What file would you like to store the result in?")

# Set up the file for writing:
output = open(filename, "w")

# Put the users response (which is what the getresponse() function
# returns into a variable called numbers
numbers = getresponse()

# Add the two elements of the array together using the addnumbers()
# function
answer = addnumbers(numbers[0], numbers[1])

# Turn the answer into a string and write it to file
stringanswer = str(answer)
output.write(stringanswer)

# And finally, don't forget to close the file!
output.close()
```

POSSIBLE QUESTIONS

UNIT – III

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. What is Programming?
2. List some elements of python
3. Write the syntax of python program
4. List some features of python
5. Define Variables

PART – C (6 MARKS)

1. Explain the Structure of Python Programming.
2. Explain the Features of Python.
3. Explain why python is called as an Interpreter Language.
4. Discuss in detail about Elements of python
5. Explain how to create a variable in python with example.
6. Write a python program to find factors of number
7. Explain the Data types in python.
8. Write a python program to solve the Quadratic Equation
9. Discuss in detail about Array with example
10. Write a python program to find the sum of natural numbers

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

(For the Candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - III : (Objective Type Multiple choice Questions each Question carries one Mark)

PROGRAMMING IN PYTHON [16CAU501B]

PART - A (Online Examination)

Questions	**Opt1**	**Opt2**	**Opt3**	**Opt4**	**Key**
Python is an _____________ Language	Logical	Interpreted	Procedural	Structural	Interpreted
Python was designed by __________________	John Chambers	Gentleman	Rossum	Ritchie	Rossum
Which of these in not a core data type?	Lists	Dictionary	Tuples	Class	Class
Which of the following will run without errors ?	round(45.8)	round(6352.898,2,5)	round()	round(7463.123,2,1)	round(45.8)
What error occurs when you execute? apple = mango	SyntaxError	NameError	ValueError	TypeError	NameError
Which of the following is not a complex number?	k = 2 + 3j	k = 2 + 3l	k = complex(2, 3)	k = 2 + 3J	k = 2 + 3l
What is the type of inf?	Boolean	Integer	Float	Complex	Float
What does ~4 evaluate to?	-5	-4	-3	3	-5
What does ~~~~~~5 evaluate to?	5	-11	11	-5	5
Which of the following is incorrect?	float(‘inf’)	float(‘nan’)	float(’56’+’78’)	float(’12+34′)	float(’12+34′)
What is the result of round(0.5) – round(-0.5)?	1	2	0	-1	2
What does 3 ^ 4 evaluate to?	81	12	0.75	7	7
Python is an example of a ________________	low level language	high level language	middle level language	assembly language	high level language

_________ is an instruction that causes the Python interpreter to display a value on the screen.	input statement	print statement	display	statement	print statement
Evaluate the expression given below if A= 16 and B = 15. A % B // A	0	1	1	-1	0
Which of the following operators has its associativity from right to left?	+	//	%	**	**
What is the value of x if: x = int(43.55+2/2)	43	44	22	23	44
Which of the following is the truncation division operator?	/	%	//	/	//
What is the value of the following expression:float(22//3+3/3)	8	8	8.3	8.33	8
What is returned by math.ceil(3.4)?	3	4	4.01	3.01	4
What is the output of this expression if x=22.19? print("%5.2f"%x)	56	56.24	56.23	56.236	56.24
What is the output of the code snippet shown below? X=”hi” print(“05d”%X)	00000hi	000hi	hi000	error	error
Consider the snippet of code shown below and predict the output. X=”san-foundry” print(“%56s”,X)	56 blank spaces before san-foundry	56 blank spaces before san and foundry	56 blank spaces after san-foundry	no change	56 blank spaces before san-foundry
Which of the following commands will create a list?	list1 = list()	list1 = [].	list1 = list([1, 2, 3])	all of the mentioned	all of the mentioned
What is the output when we execute list(“hello”)?	[‘h’, ‘e’, ‘l’, ‘l’, ‘o’].	[‘hello’].	[‘llo’].	[‘olleh’].	[‘h’, ‘e’, ‘l’, ‘l’, ‘o’].
Suppose listExample is [‘h’,’e’,’l’,’l’,’o’], what is len(listExample)?	5	4	None	Error	5
Suppose list1 is [2445,133,12454,123], what is max(list1) ?	2445	133	12454	123	12454
Suppose list1 is [3, 5, 25, 1, 3], what is min(list1) ?	3	5	25	1	1
Suppose list1 is [1, 5, 9], what is sum(list1) ?	1	9	15	Error	15

To shuffle the list(say list1) what function do we use ?	list1.shuffle()	shuffle(list1)	random.shuffle (list1)	random.shuffleList(list1)	random.shuffle(list1)
Suppose list1 is [2, 33, 222, 14, 25], What is list1[-1] ?	Error	None	25	2	25
Which of the following is a Python tuple?	[1, 2, 3].	(1, 2, 3)	{1, 2, 3}	{}	(1, 2, 3)
Suppose t = (1, 2, 4, 3), which of the following is incorrect?	print(t[3])	t[3] = 45	print(max(t))	print(len(t))	t[3] = 45
What will be the output? >>>t=(1,2,4,3) >>>t[1:3]	(1, 2)	(1, 2, 4)	(2, 4)	(2, 4, 3)	(2, 4)
What will be the output? >>>t=(1,2,4,3) >>>t[1:-1]	(1, 2)	(1, 2, 4)	(2, 4)	(2, 4, 3)	(2, 4)
What will be the output? d = {"john":40, "peter":45} d["john"]	40	45	“john”	“peter”	40
What will be the output? >>>t = (1, 2) >>>2 * t	(1, 2, 1, 2)	[1, 2, 1, 2].	(1, 1, 2, 2)	[1, 1, 2, 2].	(1, 2, 1, 2)
What will be the output? >>>my_tuple = (1, 2, 3, 4) >>>my_tuple.append((5, 6, 7)) >>>print len(my_tuple)	1	2	5	Error	Error
Which of these about a set is not true?	Mutable data type	Allows duplicate values	Data type with unordered values	Immutable data type	Immutable data type
Which of the following is not the correct syntax for creating a set?	set([[1,2],[3,4]])	set([1,2,2,3,4])	set((1,2,3,4))	{1,2,3,4}	set([[1,2],[3,4]])
What is the output of the following code? nums = set([1,1,2,3,3,3,4,4]) print(len(nums))	7	Error, invalid syntax for formation of set	4	8	4
Which of the following statements is used to create an empty set?	{ }	set()	[ ].	( )	set()

What is the output of the following piece of code when executed in the python shell? >>> a={5,4}
>>> b={1,2,4,5}
>>> a<b
	{1,2}	TRUE	FALSE	Invalid operation	TRUE

If a={5,6,7,8}, which of the following statements is false?	print(len(a))	print(min(a))	a.remove(5)	a[2]=45	a[2]=45
If a={5,6,7}, what happens when a.add(5) is executed?	a={5,5,6,7}	a={5,6,7}	Error as there is no add function for set data type	Error as 5 already exists in the set	a={5,6,7}
Read the code shown below carefully and pick out the keys? d = {"john":40, "peter":45}	"john", 40, 45, and "peter"	"john" and "peter"	40 and 45	d = (40:"john", 45:"peter")	"john" and "peter"
What is the output? d = {"john":40, "peter":45} d["john"]	40	45	"john"	"peter"	40
Suppose d = {"john":40, "peter":45}, to delete the entry for "john" what command do we use	d.delete("john":40)	d.delete("john")	del d["john"].	del d("john":40)	del d["john"].
Read the information given below carefully and write a list comprehension such that the output is: ['e', 'o'] w="hello" v=('a', 'e', 'i', 'o', 'u')	[x for w in v if x in v]	[x for x in w if x in v]	[x for x in v if w in v]	[x for v in w for x in w]	[x for x in w if x in v]
Which of the statements about dictionary values if false?	More than one key can have the same value	The values of the dictionary can be accessed as dict[key].	Values of a dictionary must be unique	Values of a dictionary can be a mixture of letters and numbers	Values of a dictionary must be unique
What is the output of the following snippet of code? >>> a={1:"A",2:"B",3:"C"} >>> del a	method del doesn't exist for the dictionary	del deletes the values in the dictionary	del deletes the entire dictionary	del deletes the keys in the dictionary	del deletes the entire dictionary
What is the output of the snippet of code shown below? ['hello', 'morning'][bool('')]	error	no output	hello	morning	hello

What is the output of the code shown? ['f', 't'][bool('spam')]	t	f	No output	error	t
____________ is a named collection of objects, where each object is identified by an index.	number	string	list	index	list
____________ is one of the values in a list (or other sequence).	number	string	list	element	element
__________is a thing to which a variable can refer.	number	object	list	element	object
____________ is a sequence type that is similar to a list except that it is immutable.	number	string	list	tuple	tuple
_____________is a type in which the elements cannot be modified	Mutable data type	immutable data type	component	Data type	immutable data type
____________ is a data type in which the elements can be modified.	Mutable data type	immutable data type	component	Data type	Mutable data type
____________ is a collection of key-value pairs that maps from keys to values.	number	string	dictionary	list	dictionary

UNIT – IV

SYLLABUS

Introduction to Python: Python Interpreter-Using Python as calculator-Python shell-Indentation. Atoms-Identifiers and keywords-Literals-Strings-Operators (Arithmetic operator, Relational operator, Logical or Boolean operator, Assignment, Operator, Ternary operator, Bit wise operator, Increment or Decrement operator).

INTRODUCTION TO PYTHON

- Python is simple to use, but it is a real programming language, offering much more structure and support for large programs than shell scripts or batch files can offer. On the other hand, Python also offers much more error checking than C, and, being a very-high-level language, it has high-level data types built in, such as flexible arrays and dictionaries. Because of its more general data types Python is applicable to a much larger problem domain than Awk or even Perl, yet many things are at least as easy in Python as in those languages.
- Python allows you to split your program into modules that can be reused in other Python programs. It comes with a large collection of standard modules that you can use as the basis of your programs — or as examples to start learning to program in Python. Some of these modules provide things like file I/O, system calls, sockets, and even interfaces to graphical user interface toolkits like Tk.
- Python is an interpreted language, which can save you considerable time during program development because no compilation and linking is necessary. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test functions during bottom-up program development. It is also a handy desk calculator.
- Python enables programs to be written compactly and readably. Programs written in Python are typically much shorter than equivalent C, C++, or Java programs, for several reasons:

  • The high-level data types allow you to express complex operations in a single statement;

  • Statement grouping is done by indentation instead of beginning and ending brackets;

  • No variable or argument declarations are necessary.

PYTHON INTERPRETER

Invoking the Interpreter

- The Python interpreter is usually installed as /usr/local/bin/python3.7 on those machines where it is available; putting /usr/local/bin in your Unix shell's search path makes it possible to start it by typing the command:

```
python3.7
```

- to the shell. Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., /usr/local/python is a popular alternative location.)
- On Windows machines, the Python installation is usually placed in C:\Python36, though you can change this when you're running the installer. To add this directory to your path, you can type the following command into the command prompt in a DOS box:

```
set path=%path%;C:\python36
```

- Typing an end-of-file character (Control-D on Unix, Control-Z on Windows) at the primary prompt causes the interpreter to exit with a zero exit status. If that doesn't work, you can exit the interpreter by typing the following command: quit().
- The interpreter's line-editing features include interactive editing, history substitution and code completion on systems that support readline. Perhaps the quickest check to see whether command line editing is supported is typing Control-P to the first Python prompt you get. If it beeps, you have command line editing; see Appendix Interactive Input Editing and History Substitution for an introduction to the keys. If nothing appears to happen, or if ^P is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.
- The interpreter operates somewhat like the Unix shell: when called with standard input connected to a tty device, it reads and executes commands interactively; when called with a file name argument or with a file as standard input, it reads and executes a *script* from that file.
- A second way of starting the interpreter is python -c command [arg] ..., which executes the statement(s) in *command*, analogous to the shell's -c option. Since Python statements often contain spaces or other characters that are special to the shell, it is usually advised to quote *command* in its entirety with single quotes.
- Some Python modules are also useful as scripts. These can be invoked using python -m module [arg] ..., which executes the source file for module as if you had spelled out its full name on the command line.

- When a script file is used, it is sometimes useful to be able to run the script and enter interactive mode afterwards. This can be done by passing -i before the script.
- All command line options are described in Command line and environment.

- **Argument Passing**

- When known to the interpreter, the script name and additional arguments thereafter are turned into a list of strings and assigned to the argv variable in the sys module. You can access this list by executing import sys. The length of the list is at least one; when no script and no arguments are given, sys.argv[0] is an empty string. When the script name is given as '-' (meaning standard input), sys.argv[0] is set to '-'. When -c*command* is used, sys.argv[0] is set to '-c'. When -m *module* is used, sys.argv[0] is set to the full name of the located module. Options found after -c *command* or -m *module* are not consumed by the Python interpreter's option processing but left in sys.argv for the command or module to handle.

- **Interactive Mode**

- When commands are read from a tty, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually three greater-than signs (>>>); for continuation lines it prompts with the *secondary prompt*, by default three dots (...). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
$ python3.7
Python 3.7 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Continuation lines are needed when entering a multi-line construct. As an example, take a look at this if statement:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Be careful not to fall off!")
...
Be careful not to fall off!
```

- For more on interactive mode, see Interactive Mode.

The Interpreter and Its Environment

- **Source Code Encoding**

➢ By default, Python source files are treated as encoded in UTF-8. In that encoding, characters of most languages in the world can be used simultaneously in string literals, identifiers and comments — although the standard library only uses ASCII characters for identifiers, a convention that any portable code should follow. To display all these characters properly, your editor must recognize that the file is UTF-8, and it must use a font that supports all the characters in the file.

➢ To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

```
# -*- coding: encoding -*-
```

➢ where *encoding* is one of the valid codecs supported by Python.

➢ For example, to declare that Windows-1252 encoding is to be used, the first line of your source code file should be:

```
# -*- coding: cp1252 -*-
```

➢ One exception to the *first line* rule is when the source code starts with a UNIX "shebang" line. In this case, the encoding declaration should be added as the second line of the file. For example:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

USING PYTHON AS A CALCULATOR

➢ Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, >>>. (It shouldn't take long.)

- **Numbers**

➢ The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages (for example, Pascal or C); parentheses (()) can be used for grouping. For example:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
```

```
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # division always returns a floating point number
1.6
```

- The integer numbers (e.g. 2, 4, 20) have type int, the ones with a fractional part (e.g. 5.0, 1.6) have typefloat. We will see more about numeric types later in the tutorial.
- Division (/) always returns a float. To do floor division and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use %:

```
>>> 17 / 3  # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3  # floor division discards the fractional part
5
>>> 17 % 3  # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2  # result * divisor + remainder
17
```

- With Python, it is possible to use the ** operator to calculate powers :

```
>>> 5 ** 2  # 5 squared
25
>>> 2 ** 7  # 2 to the power of 7
128
```

- The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

- If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```
>>> n  # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

- There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
>>> 4 * 3.75 - 1
14.0
```

- In interactive mode, the last printed expression is assigned to the variable _. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

- This variable should be treated as read-only by the user. Don't explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.
- In addition to int and float, Python supports other types of numbers, such as Decimal and Fraction. Python also has built-in support for complex numbers, and uses the j or J suffix to indicate the imaginary part (e.g. 3+5j).

- **Strings**

- Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result [2]. \ can be used to escape quotes:

```
>>> 'spam eggs'  # single quotes
'spam eggs'
```

```
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

- In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes. The print() function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
>>> print('"Isn\'t," they said.')
"Isn't," they said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

- If you don’t want characters prefaced by \ to be interpreted as special characters, you can use *raw strings* by adding an r before the first quote:

```
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

- String literals can span multiple lines. One way is using triple-quotes: """...""" or '''...'''. End of lines are automatically included in the string, but it's possible to prevent this by adding a \ at the end of the line. The following example:

```
print("""\
Usage: thingy [OPTIONS]
    -h                  Display this usage message
    -H hostname             Hostname to connect to
""")
```

- produces the following output (note that the initial newline is not included):

```
Usage: thingy [OPTIONS]
    -h                  Display this usage message
    -H hostname             Hostname to connect to
```

- Strings can be concatenated (glued together) with the + operator, and repeated with *:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

- Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```
>>> 'Py' 'thon'
'Python'
```

- This feature is particularly useful when you want to break long strings:

```
>>> text = ('Put several strings within parentheses '
...         'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

- This only works with two literals though, not with variables or expressions:

```
>>> prefix = 'Py'
>>> prefix 'thon'  # can't concatenate a variable and a string literal
  ...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  ...
SyntaxError: invalid syntax
```

- If you want to concatenate variables or a variable and a literal, use +:

```
>>> prefix + 'thon'
'Python'
```

- Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
>>> word = 'Python'
>>> word[0]  # character in position 0
'P'
>>> word[5]  # character in position 5
'n'
```

- Indices may also be negative numbers, to start counting from the right:

```
>>> word[-1]  # last character
'n'
>>> word[-2]  # second-last character
'o'
>>> word[-6]
'P'
```

- Note that since -0 is the same as 0, negative indices start from -1.
- In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing*allows you to obtain substring:

```
>>> word[0:2]  # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5]  # characters from position 2 (included) to 5 (excluded)
```

```
'tho'
```

- Note how the start is always included, and the end always excluded. This makes sure that s[:i] + s[i:] is always equal to s:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

- Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> word[:2]   # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]   # characters from position 4 (included) to the end
'on'
>>> word[-2:]  # characters from the second-last (included) to the end
'on'
```

- One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

- The first row of numbers gives the position of the indices 0…6 in the string; the second row gives the corresponding negative indices. The slice from *i* to *j* consists of all characters between the edges labeled *i* and *j*, respectively.
- For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of word[1:3] is 2.
- Attempting to use an index that is too large will result in an error:

```
>>> word[42]  # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

➢ However, out of range slice indexes are handled gracefully when used for slicing:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

➢ Python strings cannot be changed — they are immutable. Therefore, assigning to an indexed position in the string results in an error:

```
>>> word[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

➢ If you need a different string, you should create a new one:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

➢ The built-in function len() returns the length of a string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

- **Lists**

➢ Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

- Like strings (and all other built-in sequence type), lists can be indexed and sliced:

```
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

- All slice operations return a new list containing the requested elements. This means that the following slice returns a new (shallow) copy of the list:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

- Lists also support operations like concatenation:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

- Unlike strings, which are immutable, lists is a mutable type, i.e. it is possible to change their content:

```
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

- You can also add new items at the end of the list, by using the append() *method* (we will see more about methods later):

```
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

- Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

- The built-in function len() also applies to lists:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

- It is possible to nest lists (create lists containing other lists), for example:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
```

```
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

- **First Steps Towards Programming**

➢ Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
1
1
2
3
5
8
```

➢ This example introduces several new features.

➢ The first line contains a *multiple assignment*: the variables a and b simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.

➢ The while loop executes as long as the condition (here: a < 10) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) and != (not equal to).

- The *body* of the loop is *indented*: indentation is Python's way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; all decent text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.
- The print() function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```
>>> i = 256*256
>>> print('The value of i is', i)
The value of i is 65536
```

- The keyword argument *end* can be used to avoid the newline after the output, or end the output with a different string:

```
>>> a, b = 0, 1
>>> while a < 1000:
...     print(a, end=',')
...     a, b = b, a+b
...
0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

PYTHON SHELL

- In this course we will be using Python 3.4, but you would be fine if you choose to use Python 3.4 or above.
- In the last chapter, we have installed Python Interpreter. An Interpreter is a program which translates your code into machine language and then executes it line by line.
- We can use Python Interpreter in two modes:

1. Interactive Mode.

2. Script Mode.

- In Interactive Mode, Python interpreter waits for you to enter command. When you type the command, Python interpreter goes ahead and executes the command, then it waits again for your next command.
- In Script mode, Python Interpreter runs a program from the source file.

- **Interactive Mode**

- Python interpreter in interactive mode is commonly known as Python Shell. To start the Python Shell enter the following command in terminal or command prompt:

```
C:\Users\Q>python
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 20:20:57) [MSC v.1600 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- If you system has Python 2 and Python 3 both, for example Ubuntu comes with Python 2 and 3 installed by default. To start the Python 3 Shell enter python3 instead of just python.

```
q@vm:~$ python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- What you are seeing is called Python Shell. >>> is known as prompt string, it simply means that Python shell is ready to accept you commands. Python shell allows you type Python code and see the result immediately. In technical jargon this is also known as REPL short for Read-Eval-Print-Loop. Whenever you hear REPL think of an environment which allows you quickly test code snippets and see results immediately, just like a Calculator. In Python shell, enter the following calculations one by one and hit enter to get the result.

```
>>>
>>> 88 + 4
92
```

```
>>>
>>> 45 * 4
180
>>>
>>> 17 / 3
5.666666666666667
>>>
>>> 78 - 42
36
>>>
```

- ➢ In Python, we use print() function to print something to the screen.
- ➢ In the Python shell type print("Big Python") and hit enter:

```
>>>
>>> print("Big Python")
Big Python
>>>
```

- ➢ We have just used two important programming constructs - A function and a string.

1. print() is a function - A function in programming is a chuck of code which does something very specific. In our case, the print() function prints the argument (i.e "Big Python") it is given to the console.
2. A string is just a sequence of string enclosed inside single or double quotes. For example: "olleh", 'print it' are strings but 1 and 3 are not.
3. Don't worry, we are not expecting you understand these things at this point. In upcoming lessons we will discuss these concepts in great detail.
4. Commands such as 17 / 3, print("Big Python") are know as statements in programming. A statement is simply a instruction for the Python interpreter to execute. Statements are of different types as we will see. A program usually consists of sequence of statements.
5. To quit the Python shell in Windows hit Ctrl+Z followed by the Enter Key, On Linux or Mac hit Ctrl+D followed by Enter key.

- **Script Mode**

  - ➢ Python Shell is great for testing small chunks of code but there is one problem - the statements you enter in the Python shell are not saved anywhere.

- In case, you want to execute same set of statements multiple times you would be better off to save the entire code in a file. Then, use the Python interpreter in script mode to execute the code from a file.
- Create a new directory named python101, you can create this directory anywhere you want, just remember the location because we will use this directory to store all our programs throughout this course. Create another directory inside python101 named Chapter-03 to store the source files for this chapter.
- To create programs you can use any text editor, just make sure to save your file as plain text. However, if you are desperately looking for recommendation go for Sublime Text.
- Create a new file named hello.py inside Chapter-03 directory add the following code to it:
- **python101/Chapter-03/hello.py**

```
print("Woods are lovely dark and deep")
print("but I have promises to keep")
print("and miles to go before I sleep")
```

- By convention, all Python programs have .py extension. The file hello.py is called source code or source file or script file or module. To execute the program, open terminal or command prompt and change your current working directory to python101 using the cd command, then type the following command:

```
q@vm:~/python101/Chapter-03$ python3 hello.py
Woods are lovely dark and deep
but I have promises to keep
and miles to go before I sleep
q@vm:~/python101/Chapter-03$
```

- **Note**: On Windows use python hello.py to execute the the program.
- This command starts the Python interpreter in script mode and executes the statements in the hello.py file.
- We know that in the Python Shell, if you type any expression and hit enter, the Python interpreter evaluates the expression and displays the result.

```
>>>
>>> 12+8
20
```

```
>>> 75/2
37.5
>>> 100*2
200
>>> 100-24
76
>>>
```

- **However, if you type these statements in a file and run the file, you will get no output at all. Create a new file named no_output.py in the Chapter-03 directory and add the following code to it.**
- python101/Chapter-03/no_output.py

```
12+8
75/2
100*2
100-24
```

- **To the run the file enter the following command.**

```
q@vm:~/python101/Chapter-03$ python3 no_output.py
q@vm:~/python101/Chapter-03$
```

- **As you can see, the program didn't output anything.**
- **To print values from Python script you must explicitly use the print() function. Create a new file named no_output2.py with the following code:**
- python101/Chapter-03/no_output2.py

```
print(12+8)
print(75/2)
print(100*2)
print(100-24)
```

Output:

```
q@vm:~/python101/Chapter-03$ python3 no_output.py
20
37.5
200
76
q@vm:~/python101/Chapter-03$
```

INDENTATION

Whitespace is important in Python. Actually, **whitespace at the beginning of the line is important**. This is called **indentation**. Leading whitespace (spaces and tabs) at the beginning of the logical line is used to determine the indentation level of the logical line, which in turn is used to determine the grouping of statements.

This means that statements which go together **must** have the same indentation. Each such set of statements is called a **block**. We will see examples of how blocks are important in later chapters.

One thing you should remember is how wrong indentation can give rise to errors. For example:

```
i = 5
 print 'Value is', i # Error! Notice a single space at the start of the line
print 'I repeat, the value is', i
```

When you run this, you get the following error:

```
  File "whitespace.py", line 4
    print 'Value is', i # Error! Notice a single space at the start of the line
    ^
SyntaxError: invalid syntax
```

Notice that there is a single space at the beginning of the second line. The error indicated by Python tells us that the syntax of the program is invalid i.e. the program was not properly written. What this means to you is that *you cannot arbitrarily start new blocks of statements* (except for the main block which you have been using all along, of course). Cases where you can use new blocks will be detailed in later chapters such as the control flow chapter.

ATOMS

➢ Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is:

```
atom      ::=  identifier | literal | enclosure
enclosure ::=  parenth_form | list_display | dict_display | set_display
               | generator_expression | yield_atom
```

IDENTIFIERS AND KEYWORDS

Python Keywords

- Keywords are the reserved words in Python.
- We cannot use a keyword as variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language.
- In Python, keywords are case sensitive.
- There are 33 keywords in Python 3.3. This number can vary slightly in course of time.
- All the keywords except True, False and None are in lowercase and they must be written as it is. The list of all the keywords are given below.

Keywords in Python programming language

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	Except	in	raise	

Python Identifiers

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

➢ **Rules for writing identifiers**

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like myClass, var_1 and print_this_to_screen, all are valid example.
2. An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
3. Keywords cannot be used as identifiers.

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

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

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

5. Identifier can be of any length.

LITERALS

➢ Python supports string and bytes literals and various numeric literals:

```
literal ::=  stringliteral | bytesliteral
             | integer | floatnumber | imagnumber
```

- Evaluation of a literal yields an object of the given type (string, bytes, integer, floating point number, complex number) with the given value. The value may be approximated in the case of floating point and imaginary (complex) literals. See section Literals for details.
- All literals correspond to immutable data types, and hence the object's identity is less important than its value. Multiple evaluations of literals with the same value (either the same occurrence in the program text or a different occurrence) may obtain the same object or a different object with the same value.

STRINGS

- **A compound data type**
- Strings are qualitatively different from the other two because they are made up of smaller pieces— characters. Types that comprise smaller pieces are called compound data types. Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts. This ambiguity is useful. The bracket operator selects a single character from a string.

```
>>> fruit = "banana"
>>> letter = fruit[1]
>>> print letter
```

- The expression fruit[1] selects character number 1 from fruit. The variable letter refers to the result. When we display letter, we get a surprise:

```
a
```

- The first letter of "banana" is not a. Unless you are a computer scientist. In that case you should think of the expression in brackets as an offset from the beginning of the string, and the offset of the first letter is zero. So b is the 0th letter ("zero-eth") of "banana", a is the 1th letter ("one-eth"), and n is the 2th ("two-eth") letter.
- To get the first letter of a string, you just put 0, or any expression with the value 0, in the brackets:

```
>>> letter = fruit[0]
>>> print letter b
```

- The expression in brackets is called an index. An index specifies a member of an ordered set, in this case the set of characters in the string. The index indicates which one you want, hence the name. It can be any integer expression.
- **Length**
- The len function returns the number of characters in a string:

```
>>> fruit = "banana"
>>> len(fruit)
6
```

- To get the last letter of a string, you might be tempted to try something like this:

```
length = len(fruit)
```

```
last = fruit[length] # ERROR!
```

- That won't work. It causes the runtime error IndexError: string index out of range. The reason is that there is no 6th letter in "banana". Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, we have to subtract 1 from length:

```
length = len(fruit)
last = fruit[length-1]
```

- Alternatively, we can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

➢ **<u>Traversal and the for loop</u>**

- A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal. One way to encode a traversal is with a while statement:

```
index = 0
while index < len(fruit):
letter = fruit[index]
print letter
 index = index + 1
```

- This loop traverses the string and displays each letter on a line by itself. The loop condition is index < len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed. The last character accessed is the one with the index len(fruit)-1, which is the last character in the string
- Using an index to traverse a set of values is so common that Python provides an alternative, simpler syntax—the for loop:

```
for char in fruit:
   print char
```

- Each time through the loop, the next character in the string is assigned to the variable char. The loop continues until no characters are left.
- The following example shows how to use concatenation and a for loop to generate an abecedarian series. "Abecedarian" refers to a series or list in which the elements appear in alphabetical order. For example, in Robert McCloskey's book Make Way for Ducklings, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

```
prefixes = "JKLMNOPQ"
suffix = "ack"
for letter in prefixes:
    print letter + suffix
```

The output of this program is:

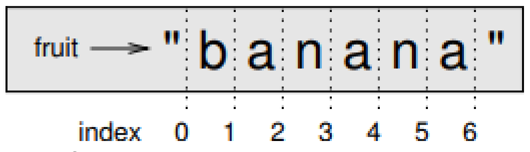
```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

- **<u>String slices</u>**

- **A s**egment of a string is called a slice. Selecting a slice is similar to selecting a character:

```
>>> s = "Peter, Paul, and Mary"
>>> print s[0:5]
Peter
>>> print s[7:11]
Paul
>>> print s[17:21]
Mary
```

- The operator [n:m] returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last. This behavior is counterintuitive; it makes more sense if you imagine the indices pointing between the characters, as in the following diagram:



- If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string. Thus:

```
>>> fruit = "banana"
>>> fruit[:3] 'ban'
>>> fruit[3:] 'ana'
```

- **<u>String comparison</u>**

- The comparison operators work on strings. To see if two strings are equal:

```
if word == "banana":
 print "Yes, we have no bananas!"
```

- Other comparison operations are useful for putting words in alphabetical order:

```
if word < "banana":
  print "Your word," + word + ", comes before banana."
 elif word > "banana": print "Your word," + word + ", comes after banana."
else: print "Yes, we have no bananas!"
```

- You should be aware, though, that Python does not handle upper- and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters. As a result: Your word, Zebra, comes before banana. A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison. A more difficult problem is making the program realize that zebras are not fruit.

➢ **Strings are immutable**

- It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string.
- For example:

```
greeting = "Hello, world!"
greeting[0] = ’J’ # ERROR!
print greeting
```

- Instead of producing the output Jello, world!, this code produces the runtime error TypeError: object doesn’t support item assignment.
- Strings are immutable, which means you can’t change an existing string. The best you can do is create a new string that is a variation on the original:

```
greeting = "Hello, world!"
newGreeting = ’J’ + greeting[1:]
print newGreeting
```

- The solution here is to concatenate a new first letter onto a slice of greeting. This operation has no effect on the original string.

➢ **A find function**

- What does the following function do?

```
def find(str, ch):
  index = 0
while index < len(str):
  if str[index] == ch:
  return index
  index = index + 1
return -1
```

- In a sense, find is the opposite of the [] operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns -1.
- This is the first example we have seen of a return statement inside a loop. If str[index] == ch, the function returns immediately, breaking out of the loop prematurely.

- If the character doesn't appear in the string, then the program exits the loop normally and returns -1.
- This pattern of computation is sometimes called a "eureka" traversal because as soon as we find what we are looking for, we can cry "Eureka!" and stop looking.
- **Looping and counting**
- The following program counts the number of times the letter a appears in a string:

```
fruit = "banana"
count = 0
for char in fruit:
if char == 'a':
count = count + 1
 print count
```

- This program demonstrates another pattern of computation called a counter. The variable count is initialized to 0 and then incremented each time an a is 7.9 The string module 77 found. (To increment is to increase by one; it is the opposite of decrement, and unrelated to "excrement," which is a noun.) When the loop exits, count contains the result—the total number of a's.

➢ **The string module**

- The string module contains useful functions that manipulate strings. As usual, we have to import the module before we can use it:

```
>>> import string
```

- The string module includes a function named find that does the same thing as the function we wrote. To call it we have to specify the name of the module and the name of the function using dot notation.

```
>>> fruit = "banana"
>>> index = string.find(fruit, "a")
>>> print index 1
```

- This example demonstrates one of the benefits of modules—they help avoid collisions between the names of built-in functions and user-defined functions. By using dot notation we can specify which version of find we want.
- Actually, string.find is more general than our version. First, it can find substrings, not just characters:

```
>>> string.find("banana", "na")
2
```

- Also, it takes an additional argument that specifies the index it should start at:

```
>>> string.find("banana", "na", 3)
4
```

- Or it can take two additional arguments that specify a range of indices: 78 Strings

```
>>> string.find("bob", "b", 1, 2)
-1
```

- In this example, the search fails because the letter b does not appear in the index range from 1 to 2 (not including 2).

➢ **Character classification**

- It is often helpful to examine a character and test whether it is upper- or lowercase, or whether it is a character or a digit. The string module provides several constants that are useful for these purposes.
- The string string.lowercase contains all of the letters that the system considers to be lowercase. Similarly, string.uppercase contains all of the uppercase letters. Try the following and see what you get:

```
>>> print string.lowercase
>>> print string.uppercase
>>> print string.digits
```

- We can use these constants and find to classify characters. For example, if find(lowercase, ch) returns a value other than -1, then ch must be lowercase:

```
def isLower(ch):
return string.find(string.lowercase, ch) != -1
```

- Alternatively, we can take advantage of the in operator, which determines whether a character appears in a string:

```
def isLower(ch):
 return ch in string.lowercase
```

- As yet another alternative, we can use the comparison operator:

```
def isLower(ch):
return ’a’ <= ch <= ’z’
```

- If ch is between a and z, it must be a lowercase letter.
- Another constant defined in the string module may surprise you when you print it

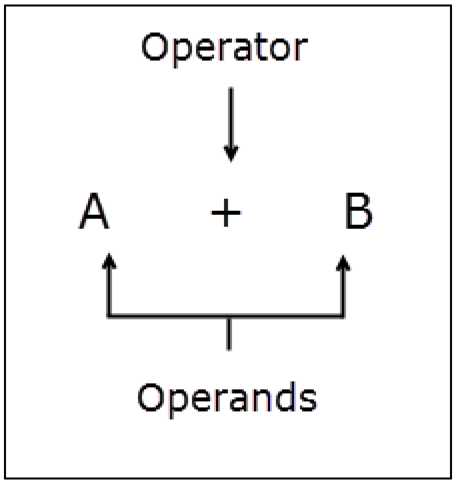
```
>>> print string.whitespace
```

- Whitespace characters move the cursor without printing anything. They create the white space between visible characters (at least on white paper). The constant string.whitespace contains all the whitespace characters, including space, tab (\t), and newline (\n).
- There are other useful functions in the string module, but this book isn’t intended to be a reference manual. On the other hand, the Python Library Reference is.

OPERATORS IN PYTHON

➢ The various operators provided by Python.

- **Operator**: An operator is a symbol which specifies a specific action.
- **Operand**: An operand is a data item on which operator acts.



- Some operators require two operands while others require only one.

Expression: An expression is nothing but a combination of operators, variables, constants and function calls that results in a value. For example:

```
## some valid expressions

1 + 8
(3 * 9) / 5
a * b + c * 3
a + b * math.pi
d + e * math.sqrt(441)
```

Let's start with Arithmetic Operators.

Arithmetic Operators #

- Arithmetic operators are commonly used to perform numeric calculations. Python has following Arithmetic operators.

Operator	Description	Example
+	Addition operator	100 + 45 = 145
-	Subtraction operator	500 - 65 = 435
*	Multiplication operator	25 * 4 = 100
/	Float Division Operator	10 / 2 = 5.0

Operator	Description	Example
//	Integer Division Operator	10 / 2 = 5
**	Exponentiation Operator	5 ** 3 = 125
%	Remainder Operator	10 % 3 = 1

- We use +, -and * operators in our daily life, so they don't deserve any explanation. However, the important thing to note is that + and - operators can be binary as well as unary. A unary operator has only one operand. We can use - operator to negate any positive number. For example: -5, in this case - operator is acting as a unary operator, whereas in 100 - 40, - operator is acting as a binary operator. Similarly, we can use unary +operator. For example, +4. As expression 4 and +4 are same, applying unary + operator in an expression generally has no significance.

Float Division Operator (/)

- The / operator performs a floating point division. It simply means that / returns a floating point result. For example:

```
>>>
>>> 6/3
2.0
>>>
>>> 3.14/45
0.06977777777777779
>>>
>>>
>>> 45/2.5
18.0
>>>
>>>
>>> -5/2
-2.5
>>>
```

Integer Division Operator (//)

- The // operator works similar to / operator, but instead of returning a float value it returns an integer. For example:

```
>>>
>>> 6//3
2
>>>
>>> 100//6
16
>>>
```

- Unlike / operator, when the result is negative then // operator rounds the result away from zero to the nearest integer.

```
>>>
>>> -5//2
-3
>>>
>>> -5/2
-2.5
>>>
```

Exponentiation Operator (**) #

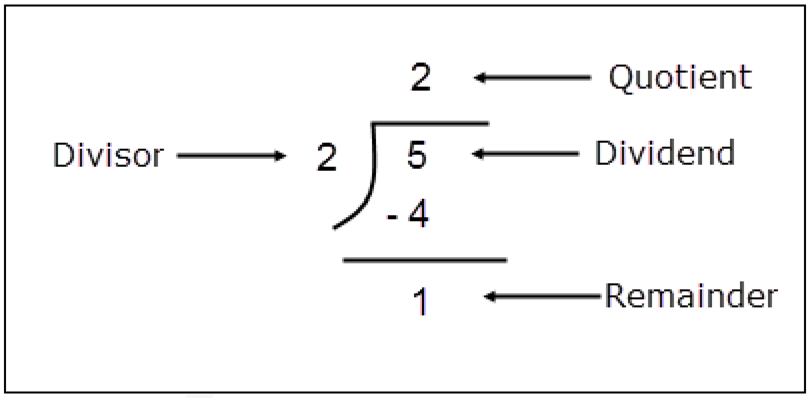
- We use ** operator to calculate a^b. For example:

```
>>>
>>> 21**2
441
>>>
>>> 5**2.2
34.493241536530384
>>>
```

Remainder Operator (%) #

- The % operator returns the remainder after after dividing left operand by the right operand. For example:

```
>>>
>>> 5%2
1
>>>
```



- The remainder operator % is a very useful operator in programming. One common use of % operator is to determine whether a number is even or not.

- A number is even if it is exactly divisible by 2. In other words, a number is even if when divided by 2, leaves 0 as remainder. We will learn how to write such program in lesson

Operator Precedence and Associativity #

- Consider the following expression:

```
10 * 5 + 9
```

- What will be it's result ?

- If multiplication is performed before addition then the answer will be 59. On the other hand, if addition is performed before multiplication then the answer will be 140. To solve this dilemma, we use Operator Precedence.

- Operators in Python are grouped together and given a precedence level. The precedence of operators is listed in the following table.

Operator	**Description**	**Associativity**
[v1, …], { v1, …}, { k1: v1, …}, (…)	List/set/dict/generator creation or comprehension, parenthesized expression	left to right
seq [n], seq [n : m], func (args…), obj.attr	Indexing, slicing, function call, attribute reference	left to right
**	Exponentiation	right to left
+x, -x, ~x	Positive, negative, bitwise not	left to right

Operator	Description	Associativity
*, /, //, %	Multiplication, float division, integer division, remainder	left to right
+, -	Addition, subtraction	left to right
<<, >>	Bitwise left, right shifts	left to right
&	Bitwise and	left to right
|	Bitwise or	left to right
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparision, membership and identity tests	left to right
not x	Boolean NOT	left to right
and	Boolean AND	left to right
or	Boolean OR	left to right
if-else	Conditional expression	left to right
lambda	lambda expression	left to right

- The operators in the upper rows has the highest precedence and it decreases as we move towards the bottom of the table. Whenever we have an expression where operators involved are of different precedence, the operator with a higher precedence is evaluated first. So, in the expression 10 * 5 + 9 evaluation of * operator is performed first followed by evaluation of + operator.

```
=> 10 * 5 + 9 (multiplication takes place first)
=> 50 + 9 (followed by addition)
=> 59 (Ans)
```

<u>Associativity of Operators</u> #

- In the precedence table operators in the same group have the same precedence, for example, (*, /, //, %) have the same precedence. Now consider the following expression:

```
5 + 12 / 2 * 4
```

- From the precedence table we know that both / and * have higher precedence than +, but the precedence of /and * is same, so which operator do you think will be evaluated first / or * ?

- To determine the order of evaluation when operator precedence is same we use Operator Associativity. Operator Associativity defines the direction in which operators of same precedence are evaluated, it can be either from left to right or right to left. Operators within same group have same associativity. As you can see in the table, the associativity of / and * is from left to right. So in the expression:

```
5 + 12 / 2 * 4
```

- The / operator will be evaluated first, followed by * operator. At last + operator is evaluated.

```
=> 5 + 12 / 2 * 4  (/ operator is evaluated first)
=> 5 + 6 * 4 (then * operator is evaluated)
=> 5 + 24 (at last + operator is evaluated)
=> 29 (Ans)
```

- The following are two noteworthy points to remember about the precedence table.

1. Associativity of most operators in the same group is from left to right except exponentiation operator (**). The associativity of exponentiation operator (**) is from right to left.
2. We sometimes use parentheses i.e () to change the order of evaluation. For example:
   ```
   2 + 10 * 4
   ```
   In the above expression * will be performed first followed by +. We can easily change operator precedence by wrapping parentheses around the expression or sub-expression which we want to evaluate first. For example:
   ```
   (2 + 10) * 4
   ```
   As precedence of () operator is higher than that of * operator (see precedence table), addition will be performed first followed by *.

   Here are some expressions and order in which they are evaluated:

Example 1:

```
Expression: 10 * 3 + (10 % 2) ** 1

1st Step: 10 * 3 + 0 ** 1
2nd Step: 10 * 3 + 0
3rd Step: 30 + 0
4th Step: 30
```

Example 2:

```
Expression: 45 % 2 - 5 / 2 + ( 9 * 3 - 1 )
```

```
1st Step: 45 % 2 - 5 / 2 + 26
2nd Step: 1 - 2.5 + 26
3rd Step: 24.5
```

Compound Assignment Operator #

- In programming it is very common to increment or decrement the value of a variable and then reassign the value back to the same variable. For example:

```
x = 10
x = x + 5
```

- Initially value of x is 10. In the second expression, we have added 10 to the existing value of x and then reassign the new value back to x. So now the value of x is 15.
- The second statement i.e x = x + 5 can be written in more succinctly manner using Compound Assignment Operator as follows:

```
x += 5
```

- Here += is known as Compound Assignment Operator. The following table lists other Compound Assignment operators available in Python.

Operator	Example	Equivalent Statement
+=	x += 4	x = x + 4
-=	x -= 4	x = x - 4
*=	x *= 4	x = x * 4
/=	x /= 4	x = x / 4
//=	x //= 4	x = x // 4
%=	x %= 4	x = x % 4
**=	x **= 4	x = x ** 4

- Unlike other C based languages likes like Java, PHP, JavaScript; Python doesn't have Increment operator (++) and decrement operator (--). In those languages, ++ and -- operators are commonly used to increment and decrement the value of variable by 1 respectively. For example, to increment/decrement the value of a variable by 1 in JavaScript you would do something like this:

```
x = 10;
x++;  // increment x by 1
console.log(x); // prints 11

x = 10;
x--;  // decrement x by 1
console.log(x); // prints 9
```

- We can easily emulate this behavior using compound assignment operator as follows:

```
x = 10
x += 1
print(x) ## prints 11

x = 10
x -= 1
print(x) ## prints 9
```

Type Conversion #

- Upto this point, we haven't given much thought about the type of data we have been using in expression in Python Shell as well as in our programs. When it comes to performing calculation involving data of different types Python has following rules:

1. When both operands involved in an expression are int, then the result will be an int.
2. When both operand involved in an expression are float, then the result will be a float.
3. When one operand is of float type and other is of type int then the result will always be a float value. In such cases, the Python interpreter automatically converts the int value to float temporarily, then performs the calculation. This process is known as Type Conversion.

Here are some examples:

```
>>>
>>> 45 * 3
135     # result is int because both operands are int
>>>
>>>
>>>
>>> 3.4 * 5.3
18.02    # result is float because both operands are float
>>>
>>>
```

```
>>>
>>> 88 * 4.3
378.4    # result is float because one operand is float
>>>
```

- In the last expression literal 88 is first converted to 88.0, and then the multiplication is carried out.

- Sometimes, it is desirable to convert data from one type to a different type at our will. To handle such situations Python provides us the following functions:

Function Name	**Description**	**Example**
int()	It accepts a string or number and returns a value of type int.	int(2.7) returns 2, int("30") returns 30
float()	It accepts a string or number and returns a value of type float	float(42) returns 42.0, float("3.4")returns 3.4
str()	It accepts any value and returns a value type str	str(12) returns "12", str(3.4) returns "3.4"

- Here are some examples:

<u>int() function</u> #

```
>>>
>>> int(2.7)  # convert 2.7 to int
2
>>>
>>> int("30") # convert "30" to int
30
>>>
```

- Note that when int() function converts a float number to int, it simply remove the digits after the decimal point. If you want to round a number use the round() function.

```
>>>
>>> int(44.55)
44
>>> round(44.55)
45
```

```
>>>
```

float() function #

```
>>>
>>> float(42)  # convert 42 to float
42.0
>>>
>>> float("3.4")  # convert "3.4" to float
3.4
>>>
```

str() function #

```
>>>
>>> str(12)   # convert 12 to str
'12'
>>>
>>> str(3.4)  # convert 3.4 to str
'3.4'
>>>
```

Breaking Statements into Multiple Lines #

- All the statements we have written until now are limited to one line. What if your statement becomes too long ?
- Typing long statements in one line is very hard to read on screen as well as on paper.
- Python allows us to break long expression into multiple lines using line continuation symbol (\ ). The \ symbol tells the Python interpreter that the statement is continued on the next line. For example:

```
>>>
>>> 1111100 + 45 - (88 / 43) + 783 \
... + 10 - 33 * 1000 + \
... 88 + 3772
1082795.953488372
>>>
```

- To expand a statement to multiple lines type the line continuation symbol (\ ) at the point where you want to break the statement and hit enter.

- When Python Shell encounters a statement which expands to multiple lines, it changes prompt string from >>>to ... . When you are done typing the statement hit enter to see the result.
- Here is another example which breaks the print() statement into multiple lines :

```
>>>
>>> print("first line\
...  second line\
...  third line")
first line second line third line
>>>
>>>
```

- The following example shows how to write multi-line statements in a Python script.

python101/Chapter-06/multiline_statements.py

```
result = 1111100 + 45 - (88 / 43) + 783 \
        + 10 - 33 * 1000 + \
        88 + 3772

print(result)

print("first line\
 second line\
 third line")
```

Output:

```
1082795.953488372
first line second line third line
```

bool Type #

- The bool data type represent two states i.e true or false. Python defines the values true and false using the reserved keywords True and False respectively. A variable of type bool can only contain one of these two values. For example:

```
>>>
>>> var1 = True
>>> var2 = False
>>>
>>> type(var1)
<class 'bool'>   # type of var1 is bool
>>>
>>> type(var2)
<class 'bool'>   # type of var2 is bool
>>>
>>>
>>> type(True)
```

```
<class 'bool'>   # type of True keyword is bool
>>>
>>> type(False)
<class 'bool'>   # type of False keyword is bool
>>>
>>> var1
True
>>>
>>> var2
False
>>>
```

- An expression which evaluates to a bool value True or False is known as boolean expression.
- We commonly use bool variables as flags. A flag is nothing but a variable which signals some condition in the program. If flag variable is set to False then it means that the condition is simply not true. On the other hand, if it is True then it means condition is true.
- Internally, Python uses 1 and 0 to represent True and False respectively. We can verify this fact by using int()function on True and False keywords as follows:

```
>>>
>>> int(True)   # convert keyword True to int
1
>>>
>>> int(False)   # convert keyword False to int
0
>>>
```

Truthy and Falsy Values #

- **Truthy values**: Values which are equivalent to bool value True is known as Truthy values.
- **Falsy values**: Values which are equivalent to bool value False is known as Falsy values.
- In Python, the following values are considered as falsy.
    1. None
    2. False
    3. Zero i.e 0, 0.0
    4. Empty sequence, for example, '', [], ()
    5. Empty dictionary i.e {}
- **Note:** Sequence and dictionary are discussed in later chapters.
- Everything else is considered as truthy . We can also test whether a value is truthy or falsy by using the bool()function. If value a truthy then bool() function returns True, otherwise it returns False. Here are some examples:

```
>>>
```

```
>>> bool("")    # an empty string is falsy value
False
>>>
>>> bool(12)    # int 12 is a truthy value
True
>>>
>>> bool(0)     # int 0 is falsy a value
False
>>>
>>> bool([])    # an empty list is a falsy value
False
>>>
>>> bool(())    # an empty tuple is a falsy value
False
>>>
>>> bool(0.2)   # float 0.2 is truthy a value
True
>>>
>>> bool("boolean")   # string "boolean" is a truthy value
True
>>>
```

- The significance of truthy and falsy values will become much more clear in the upcoming lessons.

Relational Operators #

- To compare values we use relational operators. Expression containing relational operators are known as relational expressions. If expression is true then a bool value True is returned and if the expression is false a bool value False is returned. Relational operators are binary operators. The following table lists relational operators available in Python.

Operator	Description	Example	Return Value
<	Smaller than	3 < 4	True
>	Greater than	90 > 450	False
<=	Smaller than or equal to	10 <= 11	True
>=	Greater than or equal to	31 >= 40	False
!=	Not equal to	100 !=	True

Operator	Description	Example	Return Value
		101	
==	Equal to	50==50	True

```
>>>
>>> 3 < 4       #  Is 3 is smaller than 4 ? Yes
True
>>>
>>> 90 > 450     # Is 90 is greater than 450 ? No
False
>>>
>>> 10 <= 11     # Is 10 is smaller than or equal to 11 ? Yes
True
>>>
>>> 31 >= 40     # Is 31 is greater than or equal to 40 ? No
False
>>>
>>> 100 != 101     # Is 100 is not equal to 101 ?  Yes
True
>>>
>>> 50 == 50     # Is 50 is equal to 50 ? Yes
True
>>>
```

- Beginners often confuse between == and = operators. Always remember = is an assignment operator and is used to assign a value to the variable. On the other hand, == is a equality operator and is used to test whether two values are equal or not.

Logical Operators #

- Logical operators are used to combine two or more boolean expressions and tests whether they are true or false. Expressions containing logical operators are known as Logical expressions. The following table lists logical operators available in Python.

Operator	Description
and	AND operator
or	OR operator
not	NOT operator

- The and and or are binary operators, while not is unary.

and Operator

- The and operator returns a bool value True if both operands are true. Otherwise, it returns False.
- **Syntax:** operand_1 and operand_2.
- The truth table for and operator is as follows:

operand_1	**operand_2**	**Result**
False	False	False
False	True	False
True	False	False
True	True	True

- Here are some examples:

Expression	**Intermediate Expression**	**Result**
(10>3) and (15>6)	True and True	True
(1>5) and (43==6)	False and False	False
(1==1) and (2!=2)	True and False	False

```
>>>
>>> (10>3) and (15>6)  # both conditions are true so, the result is true
True
>>>
>>> (1>5) and (43==6)  # both conditions are false so, the result is false
False
>>>
>>> (1==1) and (2!=2)  # one condition is false(right operand) so, the result is false
False
>>>
```

- The precedence of relational operators (i.e >, >=, <, <=, == and !=) is greater than that of and operator, so parentheses in the above expressions is not necessary, it is added here just to make the code more readable. For example:

```
>>>
>>> (10>3) and (15>6)
True
>>>
>>> 10 > 3 and 15 > 6  # this expression is same as above
True
>>>
```

- As you can see, the expression (10>3) and (15>6) is much more clearer than 10 > 3 and 15 > 6.
- In and operator, if the first operand is evaluated to False, then the second operand is not evaluated at all. For example:

```
>>>
>>> (10>20) and (4==4)
False
>>>
```

- In this case, (10>20) is False, and so is the whole logical expression. Hence there is no need to evaluate the expression (4==4).

or operator #

- The or operator returns False when both operands are False. Otherwise, it returns True. It's syntax is:
- **Syntax**: operand_1 or operand_2
- The truth table for or operator is as follows:

operand_1	operand_2	Result
False	False	False
False	True	True
True	False	True
True	True	True

- Here are some examples:

Expression	Intermediate Expression	Result
(100<200) or (55<6)	True or False	True
(11>55) or (6==6)	False or True	True
(1>12) or (2==3)	False or False	False
(10<22) or (20>3)	True or True	True

```
>>>
>>> (100<200) or (55<6)
True
>>>
>>> (11>55) or (6==6)
True
>>>
>>> (1>12) or (2==3)
```

```
False
>>>
>>> (10<22) or (20>3)
True
>>>
```

- In or operator, if the first operand is evaluated to True, then the second operand is not evaluated at all. For example:

```
>>>
>>> (100>20) or (90<30)
True
>>>
```

- In this case, (100>20) is True, and so is the whole logical expression. Hence there is no need to evaluate the expression (90<30).
- The precedence of or operator is less than that of and operator.

<u>not Operator</u> #

- The not operator negates the value of the expression. In other words, if the expression is True, then not operator returns False and if it is False, it returns True. Unlike the other two logical operators, the not is a unary operator. The precedence of not operator is higher than that of and operator and or operator. Its syntax is:
- **Syntax**: not operand
- The truth table for not operator is as follows:

operand	**Result**
True	False
False	True

- Here are some examples:

Expression	**Intermediate Expression**	**Result**
not (200==200)	not True	False
not (10<=5)	not False	True

```
>>>
>>> not (200==200)
False
>>>
>>> not (10<=5)
True
>>>
```

Python Bitwise Operators

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

  a = 0011 1100

  b = 0000 1101

  ----

  a&b = 0000 1100

  a|b = 0011 1101

  a^b = 0011 0001

  ~a = 1100 0011

- There are following Bitwise operators supported by Python language

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed

		binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Ternary Operator in Python

- Ternary operators also known as conditional expressions are operators that evaluate something based on a condition being true or false. It was added to Python in version 2.5. It simply allows to test a condition in a **single line** replacing the multiline if-else making the code compact.
- **Syntax :**

```
[on_true] if [expression] else [on_false]
```

1. **Simple Method to use ternary operator:**

```
# Program to demonstrate conditional operator
a, b = 10, 20

# Copy value of a in min if a < b else copy b
min = a if a < b else b

print(min)
```

Output:
```
10
```

2. **Direct Method by using tuples, Dictionary and lambda**

```
# Python program to demonstrate ternary operator
a, b = 10, 20

# Use tuple for selecting an item
print( (b, a) [a < b] )

# Use Dictionary for selecting an item
```

```
print({True: a, False: b} [a < b])

# lamda is more efficient than above two methods
# because in lambda  we are assure that
# only one expression will be evaluated unlike in
# tuple and Dictionary
print((lambda: b, lambda: a)[a < b]())
```

Run on IDE

```
Output:
10
10
10
```

3. **Ternary operator can be written as nested if-else:**

```
# Python program to demonstrate nested ternary operator
a, b = 10, 20

print ("Both a and b are equal" if a == b else "a is greater than b"
        if a > b else "b is greater than a")
```

Run on IDE

4. **Above approach can be written as:**

```
# Python program to demonstrate nested ternary operator
a, b = 10, 20

if a != b:
   if a > b:
      print("a is greater than b")
   else:
      print("b is greater than a")
else:
   print("Both a and b are equal")
```

Run on IDE

Output: b is greater than a

Increment and Decrement Operators in Python

- If you're familiar with Python, you would have known Increment and Decrement operators (both pre and post) are not allowed in it.
- Python is designed to be consistent and readable. One common error by a novice programmer in languages with ++ and -- operators is mixing up the differences (both in

precedence and in return value) between pre and post increment/decrement operators. Simple increment and decrement operators aren't needed as much as in other languages.

- You don't write things like :

```
for (int i = 0; i < 5; ++i)
```

In Python, instead we write it like

```
# A Sample Python program to show loop (unlike many
# other languages, it doesn't use ++)
for i in range(0, 5):
  print(i)
```

Run on IDE
Output:

```
0
1
2
3
4
```

We can almost always avoid use of ++ and --. For example, **x++** can be written as **x += 1** and **x--** can be written as **x -= 1**.

POSSIBLE QUESTIONS

UNIT – IV

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. What is Python?
2. Write a short note on Python Interpreter?
3. Define Identifiers
4. List some keywords in Python Programming
5. List the Operators in Python Programming
6. How to create a simple calculator using python?

PART – C (6 MARKS)

1. Explain the process of Python Interpreter
2. Explain how the Python is used as a calculator.
3. Write in detail (i) Identifier (ii) Keywords.
4. Explain the process of Strings and its Operations
5. Explain Arithmetic Operators with suitable Example.
6. Explain Logical Operators with suitable Example.
7. Explain Relational Operators with suitable Example.
8. Write in detail (i) Ternary Operatory (ii) Assignment Operator
9. Explain bitwise Operators with suitable Example.
10. Write in detail (i) Literals (ii) Increment/ Decrement Operators

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

(For the Candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - IV : (Objective Type Multiple choice Questions each Question carries one Mark)

PROGRAMMING IN PYTHON [16CAU501B]

PART - A (Online Examination)

Questions	**Opt1**	**Opt2**	**Opt3**	**Opt4**	**Key**
identifiers?	yes	no	machine dependent	independent	yes
identifier?	31 characters	63 characters	79 characters	any length.	any length.
Which of the following is an invalid variable?	my_string_1	1st_string	__	foo	1st_string
Why are local variable names beginning with an underscore discouraged?	they are used to indicate a private variables of a class	they confuse the interpreter	they are used to indicate global variables	they slow down execution	they are used to indicate a private variables of a class
Which of the following is not a keyword?	eval	assert	nonlocal	pass	eval
Which of the following is true for variable names in Python?	unlimited length	all private members must have leading and trailing underscores	underscore and ampersand are the only two special characters allowed	limited length	unlimited length
Which of the following is an invalid statement?	abc = 1,000,000	a b c = 1000 2000 3000	a,b,c = 1000, 2000, 3000	a_b_c = 1,000,000	a b c = 1000 2000 3000
Which of the following cannot be a variable?	__init__	in	it	on	in

Which is the correct operator for power(xy)?	X^y	X**y	X^^y	X*y	X**y
Which one of these is floor division?	/	//	%	**	//
What is the order of precedence in python? i) Parentheses ii) Exponential iii) Multiplication iv) Division v) Addition vi) Subtraction	i,ii,iii,iv,v,vi	ii,i,iii,iv,v,vi	ii,i,iv,iii,v,vi	i,ii,iii,iv,vi,v	i,ii,iii,iv,v,vi
What is answer of this expression, 22 % 3 is?	7	1	0	5	1
Operators with the same precedence are evaluated in which manner?	Left to Right	Right to Left	Can't say	Right	Left to Right
Which one of the following have the same precedence?	Addition and Subtraction	Multiplication and Division	Both Addition and Subtraction AND Multiplication and Division	Addition and Multiplication	Addition and Subtraction
Which one of the following have the highest precedence in the expression?	Exponential	Addition	Multiplication	Parentheses	Parentheses
What is the result of the snippet of code shown below if x=1? x<<2	8	1	2	4	4
The one's complement of 110010101 is:	001101010	110010101	1101011	110010100	001101010
Bitwise _________ gives 1 if either of the bits is 1 and 0 when both of the bits are 1.	OR	AND	XOR	NOT	XOR
Which of the following expressions can be used to multiply a given number 'a' by 4?	a<<2	a<<4	a>>2	a>>4	a<<2
What arithmetic operators cannot be used with strings ?	+	*	–	/	–

What is the output when following code is executed ? >>> str1 = 'hello' >>> str2 = ',' >>> str3 = 'world' >>> str1[-1:]	olleh	hello	h	o	o
What is the output when following code is executed ? >>>print r"\nhello"	a new line and hello	\nhello	the letter r and then hello	error	\nhello
What is “Hello”.replace(“l”, “e”)	Heeeo	Heelo	Heleo	Hello	Heeeo
To retrieve the character at index 3 from string s=”Hello” what command do we execute (multiple answers allowed) ?	s[].	s.getitem(3)	s.__getitem__(3)	s.getItem(3)	s.__getitem__(3)
What function do you use to read a string?	input(“Enter a string”)	eval(input(“Enter a string”))	enter(“Enter a string”)	eval(enter(“Enter a string”))	input(“Enter a string”)
What is the output of the following? print("abc DEF".capitalize())	abc def	ABC DEF	Abc def	Abc Def	Abc def
What is the output of the following? print("xyyzxyzxzxyy".count('yy'))	2	0	error	xyz	2
What is the output of the following? print("Hello {name1} and {name2}".format(name1='foo', name2='bin'))	Hello foo and bin	Hello {name1} and {name2}	Error	Hello and	Hello foo and bin
What is the output of the following? print('cd'.partition('cd'))	(‘cd’)	(”)	(‘cd’, ”, ”)	(”, ‘cd’, ”)	(”, ‘cd’, ”)
What is the output of the following? print('abef'.partition('cd'))	(‘abef’)	(‘abef’, ‘cd’, ”)	(‘abef’, ”, ”)	error	(‘abef’, ”, ”)
What is the output of the following? print('{0:.2}'.format(1/3))	0.333333	0.33	0.333333:.2	error	0.33
What is the output of the following? print('ab'.isalpha())	TRUE	FALSE	None	error	TRUE
What is the output of the following? print('cd'.partition('cd'))	(‘cd’)	(”)	(‘cd’, ”, ”)	(”, ‘cd’, ”)	(”, ‘cd’, ”)
What is the two’s complement of -44?	1011011	11010100	11101011	10110011	11010100

What is the value of the expression: ~100?	101	-101	100	-100	-101
Any odd number on being AND-ed with ________ always gives 1. Hint: Any even number on being AND-ed with this value always gives 0.	10	2	1	0	1
What is the value of this expression: bin(10-2)+bin(12^4)	0b10000	0b10001000	0b1000b1000	0b10000b1000	0b10000b1000
What is the output of the code show below if a=10 and b =20? a=10 b=20 a=a^b b=a^b a=a^b print(a,b)	10 20	10 10	20 10	20 20	20 10
Consider the expression given below. The value of X is: X = 2+9*((3*12)-8)/10	30	30.8	28.4	27.2	27.2
Which of the following expressions involves coercion when evaluated in Python?	4.7 – 1.5	7.9 * 6.3	1.7 % 2	3.4 + 4.6	1.7 % 2
Which among the following list of operators has the highest precedence? +, -, **, %, /, <<, >>, |	<<, >>	**	|	%	**
Which of the following expressions is an example of type conversion?	4.0 + float(3)	5.3 + 6.3	5.0 + 3	3 + 7	3 + 7
Which of the following expressions results in an error?	float('10')	int('10')	float('10.8')	int('10.8')	int('10.8')
What is the value of the expression: 4+2**5//10	3	7	77	0	7
What is the output of the following? print("ab\tcd\tef".expandtabs())	ab cd ef	abcdef	ab\tcd\tef	ab tcd ef	ab cd ef

What is the output of the following? print("ab\tcd\tef".expandtabs('+'))	ab+cd+ef	ab++++++++cd++++++++ef	ab cd ef	TypeError, an integer should be passed as an argument.	TypeError, an integer should be passed as an argument.
What is the output of the following? print("Hello {} and {}".format('foo', 'bin'))	Hello foo and bin	Hello {} and {}	Error	Hello and	Hello foo and bin
________is aspecial symbol that represents a simple computation like addition, multiplication, or string concatenation.	operator	operand	expression	evaluate	operator
which one of the values on which an operator operates?	operator	operand	expression	evaluate	operand
_______ is the set of rules governing the order in which expressions involving multiple operators and operands are evaluated.	operator	expression	rules of precedence	evaluate	rules of precedence
________is a reserved word that is used by the compiler to parse a program; you cannot use keywords like if, def, and while as variable names.	keyword	operand	expression	evaluate	keyword
________ is to join two operands end-to-end	composition	concatenate	comment	statement	concatenate
________ is to simplify an expression by performing the operations in order to yield a single value.	keyword	operand	expression	evaluate	evaluate
____________ is a combination of variables, operators, and values that represents a single result value.	keyword	operand	expression	evaluate	expression
_________ is a part of a string specified by a range of indices.	index	slice	compound	expression	slice
_______ is a compound data types whose elements can be assigned new values	index	slice	mutable	expression	mutable

_____________ is a variable or value used to select a member of an ordered set, such as a character from a string	index	slice	compound	expression	index
_________ is to iterate through the elements of a set, performing a similar operation on each.	index	slice	traverse	expression	traverse
What is the output of the following? print('{:,}'.format('1112223334'))	1112223334	111,222,333,4	1112223334	Error	Error
What is the output of the following? print('1Rn@'.lower())	n	1rn@	rn	r	1rn@

UNIT – V

SYLLABUS

Creating Python Programs: Input and Output Statements-Control statements (Branching, Looping, Conditional Statement, Exit function, Difference between break, continue and pass.). Defining Functions-Default arguments.

CREATING PYTHON PROGRAMS

- Python is considered an interpreted language because Python programs are executed by an interpreter. There are two ways to use the interpreter: command line mode and script mode. In command-line mode, you type Python programs and the interpreter prints the result:

```
$ python
Python 2.4.1 (#1, Apr 29 2005, 00:28:56)
Type "help", "copyright", "credits" or "license" for more information.
>>> print 1 + 1
2
```

- The first line of this example is the command that starts the Python interpreter. The next two lines are messages from the interpreter. The third line starts with >>>, which is the prompt the interpreter uses to indicate that it is ready. We typed print 1 + 1, and the interpreter replied 2. Alternatively, you can write a program in a file and use the interpreter to execute the contents of the file. Such a file is called a script. For example, we used a text editor to create a file named latoya.py with the following contents:

```
print 1 + 1
```

- By convention, files that contain Python programs have names that end with .py.
- To execute the program, we have to tell the interpreter the name of the script:

```
$ python latoya.py 2
```

- In other development environments, the details of executing programs may differ. Also, most programs are more interesting than this one.

INPUT AND OUTPUT STATEMENTS

- Python provides numerous built-in functions that are readily available to us at the Python prompt.

- Some of the functions like input() and print() are widely used for standard input and output operations respectively. Let us see the output section first.

Python Output Using print() function

- We use the print() function to output data to the standard output device (screen).
- We can also output data to a file, but this will be discussed later. An example use is given below.

```
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
```

- In the second print() statement, we can notice that a space was added between the stringand the value of variable a.This is by default, but we can change it.
- The actual syntax of the print() function is

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

- Here, objects is the value(s) to be printed.
- The sep separator is used between the values. It defaults into a space character.
- After all values are printed, end is printed. It defaults into a new line.
- The file is the object where the values are printed and its default value is sys.stdout(screen). Here is an example to illustrate this.

```
print(1,2,3,4)
# Output: 1 2 3 4
print(1,2,3,4,sep='*')
# Output: 1*2*3*4
print(1,2,3,4,sep='#',end='&')
# Output: 1#2#3#4&
```

Output formatting

- Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method. This method is visible to any string object.

```
>>> x = 5; y = 10
>>> print('The value of x is {} and y is {}'.format(x,y))
```

```
The value of x is 5 and y is 10
```

- Here the curly braces {} are used as placeholders. We can specify the order in which it is printed by using numbers (tuple index).

```
print('I love {0} and {1}'.format('bread','butter'))
# Output: I love bread and butter
print('I love {1} and {0}'.format('bread','butter'))
# Output: I love butter and bread
```

- We can even use keyword arguments to format the string.

```
>>> print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name
= 'John'))
Hello John, Goodmorning
```

- We can even format strings like the old sprintf() style used in C programming language. We use the % operator to accomplish this.

```
>>> x = 12.3456789
>>> print('The value of x is %3.2f' %x)
The value of x is 12.35
>>> print('The value of x is %3.4f' %x)
The value of x is 12.3457
```

Python Input

- Up till now, our programs were static. The value of variables were defined or hard coded into the source code.

- To allow flexibility we might want to take the input from the user. In Python, we have the input() function to allow this. 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.

```
>>> int('10')
10
>>> float('10')
10.0
```

- This same operation can be performed using the eval() function. But it takes it further. It can evaluate even expressions, provided the input is a string

```
>>> int('2+3')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '2+3'
>>> eval('2+3')
5
```

Python Import

- When our program grows bigger, it is a good idea to break it into different modules.

- A module is a file containing Python definitions and statements. Python modules have a filename and end with the extension .py.

- Definitions inside a module can be imported to another module or the interactive interpreter in Python. We use the import keyword to do this.

- For example, we can import the math module by typing in import math.

```
import math
print(math.pi)
```

- Now all the definitions inside math module are available in our scope. We can also import some specific attributes and functions only, using the from keyword. For example:

```
>>> from math import pi
>>> pi
3.141592653589793
```

- While importing a module, Python looks at several places defined in sys.path. It is a list of directory locations.

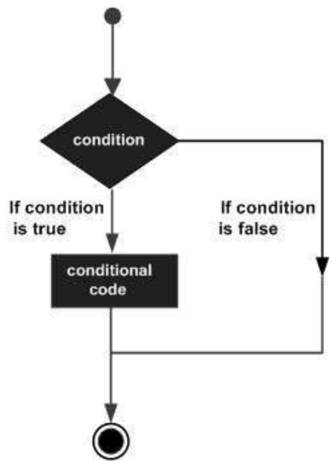
```
>>> import sys
>>> sys.path
['',
 'C:\\Python33\\Lib\\idlelib',
 'C:\\Windows\\system32\\python33.zip',
 'C:\\Python33\\DLLs',
 'C:\\Python33\\lib',
 'C:\\Python33',
 'C:\\Python33\\lib\\site-packages']
```

CONTROL STATEMENTS

- A program’s control flow is the order in which the program’s code executes. The control flow of a Python program is regulated by conditional statements, loops, and function calls.

Conditional / Decision Making

- Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.
  - Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.
  - Following is the general form of a typical decision making structure found in most of the programming languages −



- Python programming language assumes any non-zero and non-null values as TRUE, and if it is either zero or null, then it is assumed as FALSE value.
- Python programming language provides following types of decision making statements. Click the following links to check their detail.

Sr.No.	Statement & Description
1	**if statements** An **if statement** consists of a boolean expression followed by one or more statements.
2	**if...else statements** An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE.
3	**nested if statements** You can use one **if** or **else if** statement inside another **if** or **else if**statement(s).

- Let us go through each decision making briefly −

Single Statement Suites

- If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.
- Here is an example of a **one-line if** clause −

```
#!/usr/bin/python


var = 100
```

```
if ( var == 100 ) : print "Value of expression is 100"

print "Good bye!"
```

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

```
Value of expression is 100
Good bye!
```

if-else statement

- It is also called a two-way selection statement, because it leads the program to make a choice between two alternative courses of action.
- Here's the syntax for the if statement:

```
if expression:

   statement(s)

elif expression:

   statement(s)

elif expression:

   statement(s)

...

else:

   statement(s)
```

- The elif and else clauses are optional. Note that unlike some languages, Python does not have a switch statement, so you must use if, elif, and elsefor all conditional processing.
- Here's a typical if statement:

```
if x < 0: print "x is negative"

elif x % 2: print "x is positive and odd"

else: print "x is even and non-negative"
```

- When there are multiple statements in a clause (i.e., the clause controls a block of statements), the statements are placed on separate logical lines after the line containing the clause's keyword (known as the *header line* of the clause) and indented rightward from the header line. The block terminates when the indentation returns to that of the clause header (or further left from there). When there is just a single simple statement, as here, it can follow the : on the same logical line as the header, but it can also be placed on a separate logical line, immediately after the header line and indented rightward from it. Many Python practitioners consider the separate-line style more readable:

```
if x < 0:
   print "x is negative"
elif x % 2:
   print "x is positive and odd"
else:
   print "x is even and non-negative"
```

- You can use any Python expression as the condition in an if or elif clause. When you use an expression this way, you are using it in a Boolean context. In a Boolean context, any value is taken as either true or false. As we discussed earlier, any non-zero number or non-empty string, tuple, list, or dictionary evaluates as true. Zero (of any numeric type), None, and empty strings, tuples, lists, and dictionaries evaluate as false. When you want to test a value x in a Boolean context, use the following coding style:

```
if x:
```

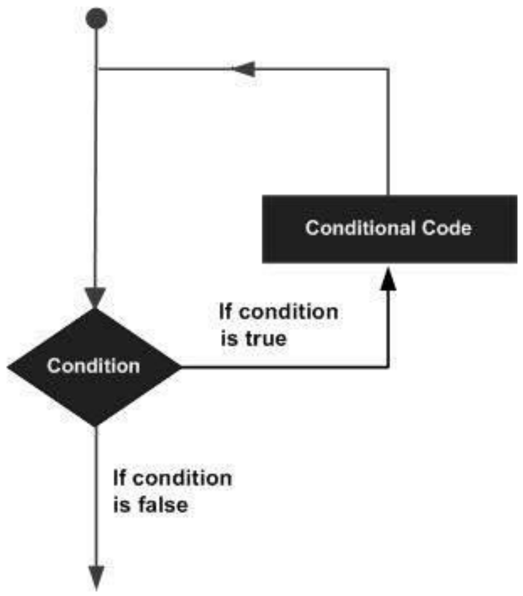
- This is the clearest and most Pythonic form. Don't use:

```
if x is True:
if x =  = True:
if bool(x):
```

- There is a crucial difference between saying that an expression “returns True" (meaning the expression returns the value 1 intended as a Boolean result) and saying that an expression “evaluates as true” (meaning the expression returns any result that is true in a Boolean context). When testing an expression, you care about the latter condition, not the former.
- If the expression for the if clause evaluates as true, the statements following the if clause execute, and the entire if statement ends. Otherwise, the expressions for any elif clauses are evaluated in order. The statements following the first elif clause whose condition is true, if any, are executed, and the entire if statement ends. Otherwise, if an else clause exists, the statements following it are executed.

LOOPING

- In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.
- Programming languages provide various control structures that allow for more complicated execution paths.
- A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −



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

Sr.No.	Loop Type & Description
1	**while loop** Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	**for loop** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	**nested loops** You can use one or more loop inside any another while, for or do..while loop.

The while Statement

- The while statement in Python supports repeated execution of a statement or block of statements that is controlled by a conditional expression. Here's the syntax for the while statement:

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

- A while statement can also include an else clause and break and continuestatements, as we'll discuss shortly.
- Here's a typical while statement:

```
count = 0
while x > 0:
   x = x // 2          # truncating division
   count += 1
print "The approximate log2 is", count
```

- First, expression, which is known as the loop condition, is evaluated. If the condition is false, the while statement ends. If the loop condition is satisfied, the statement or statements that comprise the loop body are executed. When the loop body finishes executing, the loop condition is evaluated again, to see if another iteration should be performed. This process continues until the loop condition is false, at which point the while statement ends.
- The loop body should contain code that eventually makes the loop condition false, or the loop will never end unless an exception is raised or the loop body executes a break statement. A loop that is in a function's body also ends if a return statement executes in the loop body, as the whole function ends in this case.

The for Statement

- The for statement in Python supports repeated execution of a statement or block of statements that is controlled by an iterable expression. Here's the syntax for the for statement:

```
for target in iterable:
   statement(s)
```

- Note that the in keyword is part of the syntax of the for statement and is functionally unrelated to the in operator used for membership testing. A forstatement can also include an else clause and break and continuestatements, as we'll discuss shortly.
- Here's a typical for statement:

```
for letter in "ciao":
   print "give me a", letter, "..."
```

- iterable may be any Python expression suitable as an argument to built-in function iter, which returns an iterator object (explained in detail in the next section). target is normally an identifier that names the control variable of the loop; the for statement successively rebinds this variable to each item of the iterator, in order. The statement or statements that comprise the loopbody execute once for each item in iterable (unless the loop ends because an exception is raised or a break or return statement is executed).
- A target with multiple identifiers is also allowed, as with an unpacking assignment. In this case, the iterator's items must then be sequences, each with the same length, equal to the number of identifiers in the target. For example, when d is a dictionary, this is a typical way to loop on the items in d:

```
for key, value in d.items( ):
   if not key or not value: del d[key]    # keep only true keys and values
```

- The items method returns a list of key/value pairs, so we can use a for loop with two identifiers in the target to unpack each item into key and value.

- If the iterator has a mutable underlying object, that object must not be altered while a for loop is in progress on it. For example, the previous example cannot use iteritems instead of items. iteritems returns an iterator whose underlying object is d, so therefore the loop body cannot mutate d (by deld[key]). items returns a list, though, so d is not the underlying object of the iterator and the loop body can mutate d.
- The control variable may be rebound in the loop body, but is rebound again to the next item in the iterator at the next iteration of the loop. The loop body does not execute at all if the iterator yields no items. In this case, the control variable is not bound or rebound in any way by the for statement. If the iterator yields at least one item, however, when the loop statement terminates, the control variable remains bound to the last value to which the loop statement has bound it. The following code is thus correct, as long as someseqis not empty:

```
for x in someseq:
   process(x)
print "Last item processed was", x
```

Iterators

- An iterator is any object i such that you can call i .next( ) without any arguments. i .next( ) returns the next item of iterator i, or, when iterator ihas no more items, raises a StopIteration exception. When you write a class (see Chapter 5), you can allow instances of the class to be iterators by defining such a method next. Most iterators are built by implicit or explicit calls to built-in function iter, covered in Chapter 8. Calling a generator also returns an iterator, as we’ll discuss later in this chapter.
- The for statement implicitly calls iter to get an iterator. The following statement:

```
for x in c:
   statement(s)
```

- is equivalent to:

```
_temporary_iterator = iter(c)

while True:

   try: x = _temporary_iterator.next( )

   except StopIteration: break

   statement(s)
```

- Thus, if iter(c) returns an iterator i such that i .next( ) never raisesStopIteration (an infinite iterator), the loop for x in c: never terminates (unless the statements in the loop body contain suitable break or return statements or propagate exceptions). iter(c), in turn, calls special method c .__iter__( ) to obtain and return an iterator on c. We'll talk more about the special method __iter__ in Chapter 5.
- Iterators were first introduced in Python 2.2. In earlier versions, for x in S: required S to be a sequence that was indexable with progressively larger indices 0, 1, ..., and raised an IndexError when indexed with a too-large index. Thanks to iterators, the for statement can now be used on a container that is not a sequence, such as a dictionary, as long as the container is iterable (i.e., it defines an __iter__ special method so that function iter can accept the container as the argument and return an iterator on the container). Built-in functions that used to require a sequence argument now also accept any iterable.

range and xrange

- Looping over a sequence of integers is a common task, so Python provides built-in functions range and xrange to generate and return integer sequences. The simplest, most idiomatic way to loop *n* times in Python is:

```
for i in xrange(n):
```

```
statement(s)
```

- range(x) returns a list whose items are consecutive integers from 0(included) up to x (excluded). range(x,y) returns a list whose items are consecutive integers from x (included) up to y (excluded). The result is the empty list if x is greater than or equal to y. range(x,y,step) returns a list of integers from x (included) up to y (excluded), such that the difference between each two adjacent items in the list is step. If step is less than 0, range counts down from x to y. range returns the empty list when x is greater than or equal to y and step is greater than 0, or when x is less than or equal to y and step is less than 0. If step equals 0, range raises an exception.
- While range returns a normal list object, usable for all purposes, xrangereturns a special-purpose object, specifically intended to be used in iterations like the for statement shown previously. xrange consumes less memory thanrange for this specific use. Leaving aside memory consumption, you can use range wherever you could use xrange.

Loop Control Statements

- Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- Python supports the following control statements. Click the following links to check their detail.

Sr.No.	Control Statement & Description
1	**break statement** Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	**continue statement** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

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

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

The Continue statement

- The continue statement is used to tell Python to skip the rest of the statements in the current loop block and to *continue* to the next iteration of the loop

Continue Statement

- It returns the control to the beginning of the loop.

```
# Prints all letters except 'e' and 's'
for letter in 'geeksforgeeks':
    if letter == 'e' or letter == 's':
        continue
    print 'Current Letter :', letter
    var = 10
Output:
Current Letter : g
Current Letter : k
Current Letter : f
Current Letter : o
Current Letter : r
Current Letter : g
Current Letter : k
```

The break statement

- The break statement is used to *break* out of a loop statement i.e. stop the execution of a looping statement, even if the loop condition has not become False or the sequence of items has been completely iterated over.

- An important note is that if you *break* out of a for or while loop, any corresponding loop else block is **not** executed.

Break Statement

- It brings control out of the loop

```
for letter in 'geeksforgeeks':

    # break the loop as soon it sees 'e'
    # or 's'
    if letter == 'e' or letter == 's':
        break

print 'Current Letter :', letter
```

Output:

```
Current Letter : e
```

Pass Statement

- We use pass statement to write empty loops. Pass is also used for empty control statement, function and classes.

```
# An empty loop
for letter in 'geeksforgeeks':
    pass
print 'Last Letter :', letter
```

Output:

```
Last Letter : s
```

exit() function

- To stop code execution in Python you first need to import the *sys* object. After this you can then call the exit() method to stop the program running. It is the most reliable, cross-platform way of stopping code execution. Here is a simple example.

```
import sys
sys.exit()
```

- You can also pass a string to the exit() method to get Python to spit this out when the script stops. This is probably the preferred way of doing things as you might otherwise not realize where the script stopped. Obviously you wouldn't just stop the script running arbitrarily, but you might want to prevent it from running if certain conditions haven't been met. Here is another example that stops execution if the length of an array is less than 2.

```
import sys

listofitems = []

# Code here that does something with listofitems

if len(listofitems) < 2:
  sys.exit('listofitems not long enough')
```

FUNCTION
Introduction

- Functions are reusable pieces of programs. They allow you to give a name to a block of statements and you can run that block using that name anywhere in your program and any number of times. This is known as *calling* the function. We have already used many built-in functions such as the len and range.
- Functions are **def**ined using the def keyword. This is followed by an *identifier* name for the function followed by a pair of parentheses which may enclose some names of variables and the line ends with a colon. Next follows the block of statements that are part of this function. An example will show that this is actually very simple:

DEFINING A FUNCTION

Example Defining a function

```
#!/usr/bin/python
# Filename: function1.py

def sayHello():
        print 'Hello World!' # block belonging to the function
# End of function

sayHello() # call the function

Output
$ python function1.py
Hello World!
```

How It Works

- We define a function called sayHello using the syntax as explained above. This function takes no parameters and hence there are no variables declared in the parentheses. Parameters to functions are just input to the function so that we can pass in different values to it and get back corresponding results.

Function Parameters

- A function can take parameters which are just values you supply to the function so that the function can *do* something utilising those values. These parameters are just like variables except that the values of these variables are defined when we call the function and are not assigned values within the function itself.
- Parameters are specified within the pair of parentheses in the function definition, separated by commas. When we call the function, we supply the values in the same way. Note the terminology used - the names given in the function definition are called *parameters* whereas the values you supply in the function call are called *arguments*.

Using Function Parameters

Example Using Function Parameters

```
#!/usr/bin/python
# Filename: func_param.py

def printMax(a, b):
        if a > b:
                print a, 'is maximum'
        else:
                print b, 'is maximum'

printMax(3, 4) # directly give literal values

x = 5
y = 7

printMax(x, y) # give variables as arguments
```

Output

```
$ python func_param.py
4 is maximum
```

```
7 is maximum
```

How It Works

- Here, we define a function called printMax where we take two parameters called a and b. We find out the greater number using a simple if..else statement and then print the bigger number.
- In the first usage of printMax, we directly supply the numbers i.e. arguments. In the second usage, we call the function using variables. printMax(x, y) causes value of argument x to be assigned to parameter a and the value of argument y assigned to parameter b. The printMax function works the same in both the cases.

Local Variables

- When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function i.e. variable names are *local* to the function. This is called the *scope* of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

Using Local Variables

Example Using Local Variables

```
#!/usr/bin/python
# Filename: func_local.py

def func(x):
        print 'x is', x
        x = 2
        print 'Changed local x to', x

x = 50
func(x)
print 'x is still', x
```

Output

```
$ python func_local.py
x is 50
Changed local x to 2
x is still 50
```

How It Works

- In the function, the first time that we use the *value* of the name x, Python uses the value of the parameter declared in the function.
- Next, we assign the value 2 to x. The name x is local to our function. So, when we change the value of x in the function, the x defined in the main block remains unaffected.
- In the last print statement, we confirm that the value of x in the main block is actually unaffected.

Using the global statement

- If you want to assign a value to a name defined outside the function, then you have to tell Python that the name is not local, but it is *global*. We do this using the global statement. It is impossible to assign a value to a variable defined outside a function without the global statement.
- You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function). However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is. Using the global statement makes it amply clear that the variable is defined in an outer block.

Example Using the global statement

```
# Filename: func_global.py

def func():
        global x

        print 'x is', x
        x = 2
        print 'Changed global x to', x

x = 50
func()
print 'Value of x is', x
```

Output

```
$ python func_global.py
x is 50
Changed global x to 2
Value of x is 2
```

How It Works

- The global statement is used to declare that x is a global variable - hence, when we assign a value to x inside the function, that change is reflected when we use the value of x in the main block.
- You can specify more than one global variable using the same global statement. For example, global x, y, z.

DEFAULT ARGUMENT VALUES

- For some functions, you may want to make some of its parameters as *optional* and use default values if the user does not want to provide values for such parameters. This is done with the help of default argument values. You can specify default argument values for parameters by following the parameter name in the function definition with the assignment operator (=) followed by the default value.

- Note that the default argument value should be a constant. More precisely, the default argument value should be immutable - this is explained in detail in later chapters. For now, just remember this.

Using Default Argument Values

Example Using Default Argument Values

```
#!/usr/bin/python
# Filename: func_default.py

def say(message, times = 1):
        print message * times

say('Hello')
say('World', 5)
```

Output

```
$ python func_default.py
Hello
WorldWorldWorldWorldWorld
```

How It Works

- The function named say is used to print a string as many times as want. If we don't supply a value, then by default, the string is printed just once. We achieve this by specifying a default argument value of 1to the parameter times.
- In the first usage of say, we supply only the string and it prints the string once. In the second usage of say, we supply both the string and an argument 5 stating that we want to *say* the string message 5 times.

Keyword Arguments

- If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called *keyword arguments* - we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.
- There are two *advantages* - one, using the function is easier since we do not need to worry about the order of the arguments. Two, we can give values to only those parameters which we want, provided that the other parameters have default argument values.

Using Keyword Arguments

Example Using Keyword Arguments

```
#!/usr/bin/python
# Filename: func_key.py

def func(a, b=5, c=10):
        print 'a is', a, 'and b is', b, 'and c is', c

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Output

```
$ python func_key.py
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

How It Works

- The function named func has one parameter without default argument values, followed by two parameters with default argument values.
- In the first usage, func(3, 7), the parameter a gets the value 3, the parameter b gets the value 5 and c gets the default value of 10.
- In the second usage func(25, c=24), the variable a gets the value of 25 due to the position of the argument. Then, the parameter c gets the value of 24 due to naming i.e. keyword arguments. The variableb gets the default value of 5.
- In the third usage func(c=50, a=100), we use keyword arguments completely to specify the values. Notice, that we are specifying value for parameter c before that for a even though a is defined before cin the function definition.

The return statement

- The return statement is used to *return* from a function i.e. break out of the function. We can optionally *return a value* from the function as well.

Using the literal statement

Example Using the literal statement

```
#!/usr/bin/python
# Filename: func_return.py

def maximum(x, y):
        if x > y:
                return x
        else:
                return y

print maximum(2, 3)
```

Output

```
$ python func_return.py
3
```

How It Works

- The maximum function returns the maximum of the parameters, in this case the numbers supplied to the function. It uses a simple if..else statement to find the greater value and then returns that value.
- Note that a return statement without a value is equivalent to return None. None is a special type in Python that represents nothingness. For example, it is used to indicate that a variable has no value if it has a value of None.
- Every function implicitly contains a return None statement at the end unless you have written your own return statement. You can see this by running print someFunction() where the function someFunction does not use the return statement such as:

```
def someFunction():
    pass
```

- The pass statement is used in Python to indicate an empty block of statements.

POSSIBLE QUESTIONS

UNIT – V

PART – A (20 MARKS)

(Q.NO 1 TO 20 Online Examinations)

PART – B (2 MARKS)

1. Define Function
2. Which function is used to print the statement in python?
3. List the Looping statements
4. Write the Syntax of if statement
5. Which function is used to read the input from the user?
6. Write the Syntax of function

PART – C (6 MARKS)

1. Explain the process of Input and Output Statements.
2. Discuss in detail about Conditional / Decision Making Statement.
3. Explain how to create a function with suitable example
4. Explain Looping Statement with example
5. Write in detail (i) break (ii) continue.
6. Write a python program to multiply two matrices
7. Describe the Default arguments of function
8. Write in detail (i) pass (ii) exit().
9. Write a python program to find Armstrong number in an interval
10. Difference between break, and continue statement

KARPAGAM ACADEMY OF HIGHER EDUCATION

Coimbatore – 641 021.

(For the Candidates admitted from 2016 onwards)

DEPARTMENT OF COMPUTER SCIENCE, CA & IT

UNIT - V : (Objective Type Multiple choice Questions each Question carries one Mark)

PROGRAMMING IN PYTHON [16CAU501B]

PART - A (Online Examination)

Questions	**Opt1**	**Opt2**	**Opt3**	**Opt4**	**Key**
the standard output device (screen).	input()	print()	output()	printf()	print()
widely used for standard input and output operations	printf()	nt()	input() and print()	input() and output()	int()
In Python, ___________ is a group of related statements that perform a specific task.	function	method	data	print	function
___________ through which we pass values to a function	method	data	print	parameter	parameter

What is the output of the following? x = ['ab', 'cd']
for i in x:
   i.upper()
print(x)
	[‘ab’, ‘cd’].	[‘AB’, ‘CD’].	[None, None].	[‘abv’, ‘cd’].	[‘ab’, ‘cd’].

What is the output of the following? i = 1
while True:
   if i%2 == 0:
      break
   print(i)
   i += 2
	1	1 2	1 2 3 4 5 6 …	1 3 5 7 9 11 …	1 3 5 7 9 11 …

What is the output of the following? i = 0
while i < 5:
    print(i)
    i += 1
    if i == 3:
        break
else:
    print(0)
	0 1 2 0	0 1 2	error	0 1 2 1	0 1 2

Which of the following is the use of function in python?	Functions are reusable pieces of programs	Functions don’t provide better modularity for your application	you can’t also create your own functions	All of the mentioned	Functions are reusable pieces of programs
Which keyword is use for function?	Fun	Define	Def	Function	Def

What is the output of the below program? def sayHello():
    print('Hello World!')
sayHello()
sayHello()
	Hello World!
	Hello World!
		‘Hello World!’
		‘Hello World!’
			Hello
			Hello
				Hello
				Hello W
					Hello World!
					Hello World!

Which of the following functions is a built-in function in python?	seed()	sqrt()	factorial()	print()	print()
What is the output of the expression: round(4.576)	4.5	5	4	4.6	5
The function pow(x,y,z) is evaluated as:	(x**y)**z	(x**y) / z	(x**y) % z	(x**y)*z	(x**y) % z
What is the output of the expression? round(4.5676,2)?	4.5	4.6	4.57	4.56	4.57
What is the output of the following function? any([2>8, 4>2, 1>2])	Error	TRUE	FALSE	4>2	TRUE
What is the output of the function: all(3,0,4.2)	Error	TRUE	FALSE	0	Error

What are the outcomes of the following functions?
chr(‘97’)
chr(97)
	a
	Error
		‘a’
		a
			Error
			a
				Error
				Error
					Error
					a

What is the output of the following function? complex(1+2j)	Error	1	2j	1+2j	1+2j

What is the output of the following? x = 123 for i in x: print(i)	1 2 3	123	error	1234	error
What is the output of the following? d = {0: 'a', 1: 'b', 2: 'c'} for i in d: print(i)	0 1 2	a b c	0 a 1 b 2 c	01abc2	0 1 2
What is the output of the following? for i in range(2.0): print(i)	0.0 1.0	0 1	error	0 1 00	error
What is the output of the following? for i in range(int(2.0)): print(i)	0.0 1.0	0 1	error	0 1 00	0 1
_____________ is a statement that executes a function. It consists of the name of the function followed by a list of arguments enclosed in parentheses.	function	function call	return value	argument	function call
________ is a value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.	function	function call	return value	argument	argument
____________ is the result of a function. If a function call is used as an expression, the return value is the value of the expression.	function	function call	return value	argument	return value
________ is an explicit statement that takes a value of one type and computes a corresponding value of another type.	type conversion	type coercion	module	dot notation	type conversion
_________ is a type conversion that happens automatically according to Python’s coercion rules.	type conversion	type coercion	module	dot notation	type coercion
_________ is a file that contains a collection of related functions and classes.	type conversion	type coercion	module	dot notation	module

________ is the syntax for calling a function in another module, specifying the module name followed by a dot (period) and the function name.	type conversion	type coercion	module	dot notation	dot notation
_________ is a named sequence of statements that performs some useful operation	function	function call	function definition	argument	function
_____________ is a statement that creates a new function, specifying its name, parameters, and the statements it executes.	function	function call	function definition	argument	function definition
_____________ is the order in which statements are executed during a program run.	function	function call	flow of execution	parameter	flow of execution
____________ is a name used inside a function to refer to the value passed as an argument.	function	function call	flow of execution	parameter	parameter
______________ is a variable defined inside a function.	variable	local variable	global variable	parameter	local variable
___________ is a list of the functions that are executing, printed when a runtime error occurs.	frame	traceback	parameter	flow of execution	traceback
_________ is a group of consecutive statements with the same indentation.	recursion	nesting	block	body	block
___________ is the block in a compound statement that follows the header.	recursion	nesting	block	body	body
__________ is one program structure within another, such as a conditional statement inside a branch of another conditional statement.	recursion	nesting	block	body	nesting
__________ is the process of calling the function that is currently executing	recursion	nesting	block	body	recursion
____________ is a statement that consists of a header and a body. The header ends with a colon (:).	flow of execution	conditional statement	compound statemen	statement	compound statemen
_____________ is a statement that controls the flow of execution depending on some condition.	flow of execution	conditional statement	compound statemen	statement	conditional statement

_____________ is a function that calls itself recursively without ever reaching the base case. Eventually, an infinite recursion causes a runtime error.	recursion	infinite recursion	block	body	infinite recursion
____________ is the boolean expression in a conditional statement that determines which branch is executed.	flow of execution	conditional statement	compound statemen	Condition	Condition

What is the output of the following? x = 'abcd'
for i in x:
    print(i)
    x.upper()
	a B C D	a b c d	A B C D	error	a b c d

What is the output of the following? x = 'abcd' for i in x: print(i.upper())	a b c d	A B C D	a B C D	error	A B C D
What is the output of the following? x = 'abcd' for i in range(len(x)): print(i)	a b c d	0 1 2 3	error	1 2 3 4	0 1 2 3
What is the output of the following? x = 'abcd' for i in range(len(x)): x[i].upper() print (x)	abcd	A B C D	error	a B C D	abcd
What is the output of the following code? def foo(k): k[0] = 1 q = [0] foo(q) print(q)y	[0].	[1].	[1, 0].	[0, 1].	[1].
How are keyword arguments specified in the function heading?	one star followed by a valid identifier	one underscore followed by a valid identifier	two stars followed by a valid identifier	two underscores followed by a valid identifier	two stars followed by a valid identifier
How many keyword arguments can be passed to a function in a single function call?	zero	one	zero or more	one or more	zero or more

What is the output of the following code? def foo():
    return total + 1
total = 0
print(foo())
	0	1	error	0.1	1

What is the type of sys.argv?	set	list	tuple	string	list
How are variable length arguments specified in the function heading?	one star followed by a valid identifier	one underscore followed by a valid identifier	two stars followed by a valid identifier	two underscores followed by a valid identifier	one star followed by a valid identifier
How are default arguments specified in the function heading?	identifier followed by an equal to sign and the default value	identifier followed by the default value within backticks (“)	identifier followed by the default value within square brackets ([])	identifier	identifier followed by an equal to sign and the default value
How are required arguments specified in the function heading?	identifier followed by an equal to sign and the default value	identifier followed by the default value within backticks (“)	identifier followed by the default value within square brackets ([])	identifier	identifier
Which of the following functions accepts only integers as arguments?	ord()	min()	chr()	any()	chr()
Which of the following functions will not result in an error when no arguments are passed to it?	min()	divmod()	all()	float()	float()
Which of the following functions does not throw an error?	ord()	ord(‘ ‘)	ord(”)	ord(“”)	ord(‘ ‘)
What is the output of the function: len(["hello",2, 4, 6])	4	3	Error	6	4
Suppose there is a list such that: l=[2,3,4]. If we want to print this list in reverse order, which of the following methods should be used?	reverse(l)	list(reverse[(l)])	reversed(l)	list(reversed(l))	list(reversed(l))

Register Number: ___________

[16CAU501B]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)
Coimbatore-641021.
(For the candidates admitted from 2016 onwards)
COMPUTER APPLICATIONS
FIRST INTERNAL EXAMINATION- JULY 2018
Fifth Semester
PROGRAMMING IN PYTHON

Date & Session: 13.07.2018 & FN **Duration : 2 Hours**
Class: III BCA **Maximum : 50 Marks**

PART-A (20 X 1 = 20 Marks)

Answer all the Questions

1. ________ is the process of formulating a problem, finding a solution, and expressing the solution.
   a. Problem solving b. Recover c. Format d. Retrieve
2. ________programming language is designedto be easy for humans to read and write.
   a. low level language b. high level language c. machine language d. assembly language
3. A programming language that is designed to be easy for a computer to execute; also called
   a. low level language b. high level language c. machine language d. assembly language
4. ________ is designed to be easy for humans to read and write.
   a machine language b. assembly language c. Both a& bd. high level language
5. ________ is to execute a program in a high-level language by translating it one line at a time.
   a. Compile b. Interpret c. Portability d. assembly
6. ________ is to translate a program written in a high-level language into a lowlevel language all at once, in preparation for later execution.
   a. Compile b. Interpret c. Portability d. assembly
7. ________ Program in a high-level language before being compiled.
   a. object code b. source code c. Executable d. script
8. ________ is the output of the compiler after it translates the program.
   a. object code b. source code c. Executable d. script
9. ________ is a set of instructions that specifies a computation.
   a. Algorithm b. Program c. Object d. Source
10. ________ is the structure of a program.
    a. Algorithm b. Program c. Syntax d. Source

11. ________ is an error in a program that makes it impossible to parse (and therefore impossible to interpret).
    a. Syntax error   b. runtime error   c. Exception   d. Semantic error
12. ________ is the meaning of a program.
    a. Algorithm   b. Program   c. Syntax   d. Semantics
13. ________ is an error in a program that makes it do something other than what the programmer intended.
    a. Syntax error   b. runtime error   c. Exception   d. Semantic error
14. Python was designed by ____________________
    a. John Chamber   b. Robert Gentleman   c. Guido van Rossum d. Ritchie
15. Which of the following data types is not supported in python?
    a. Number   b. String   c. List   d. Slice
16. ________________ are formal languages that havebeen designed to express computations
    a. Programming languages   b. Natural Languages
    c. Script Languages   d. Machine Languages
17. ________ are languages that are designed by people for specific applications.
    a. Programming languages   b. Natural Languages
    c. Script Languages   d. Machine Languages
18. ________ are the languages that people speak, such as English,Spanish, and French..
    a. Programming languages   b. Natural Languages
    c. Script Languages   d. Machine Languages
19. A ______ is a name that refers to a value.
    a. variable   b. datatype   c. Keyword   d. Syntax
20. Python is an __________language.
    a. Logical   b. Interpreted   c. Procedural   d. Structural

Part –B (3 x 2 = 6 Marks)
Answer all the Questions

21. What is a Debugging?
22. List the types of errors
23. Define Algorithm

Part –C (3 x 8 = 24 Marks)
Answer all the Questions

24. a. Explain the types of Errors in Programming
    (OR)
    b. Explain the Concept of Problem Solving
25. a. Discussabout Python Programming Language
    (OR)
    b. Discuss in detail about the Algorithm
26. a. Write a short note on Decision table
    (OR)
    b. Discuss about Flowchart with an example

Register Number: ____________
[16CAU501B]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)
Coimbatore-641021.
(For the candidates admitted from 2016 onwards)
COMPUTER APPLICATIONS
FIRST INTERNAL EXAMINATION- JULY 2018
Fifth Semester
PROGRAMMING IN PYTHON

Date & Session: 13.07.2018 & FN	**Duration : 2 Hours**
Class: III BCA	**Maximum : 50 Marks**

PART-A (20 X 1 = 20 Marks)

Answer all the Questions

1. ________ is the process of formulating a problem, finding a solution, and expressing the solution.

   a. Problem solving b. Recover c. Format d. Retrieve

2. ________programming language is designedto be easy for humans to read and write.

   a. low level language **b. high level language** c. machine language d. assembly language

3. A programming language that is designed to be easy for a computer to execute; also called

   a. low level language b. high level language **c. machine language** d. assembly language

4. ________ is designed to be easy for humans to read and write.

   a. Machine language b. assembly language c. Both a& b **d. high level language**

5. ________ is to execute a program in a high-level language by translating it one line at a time.

   a. Compile **b. Interpret** c. Portability d. assembly

6. ________ is to translate a program written in a high-level language into a low level language all at once, in preparation for later execution.

   a. Compile b. Interpret c. Portability d. assembly

7. ________ Program in a high-level language before being compiled.

a. object code **b. source code** c. Executable d. script

8. ________ is the output of the compiler after it translates the program.

a. object code b. source code c. Executable d. script

9. ________ is a set of instructions that specifies a computation.

a. Algorithm **b. Program** c. Object d. Source

10. ________ is the structure of a program.

a. Algorithm b. Program c. Syntax d. Source

11. ________ is an error in a program that makes it impossible to parse (and therefore impossible to interpret).

a. Syntax error b. runtime error c. Exception d. Semantic error

12. ________ is the meaning of a program.

a. Algorithm b. Program c. Syntax **d. Semantics**

13. ________ is an error in a program that makes it do something other than what the programmer intended.

a. Syntax error b. runtime error c. Exception d. Semantic error

14. Python was designed by _____________________

a. John Chamber b. Robert Gentleman **c. Guido van Rossum** d. Ritchie

15. Which of the following data types is not supported in python?

a. Number b. String c. List d. Slice

16. ________ are formal languages that have been designed to express computations.

a. Programming Languages b. Natural Languages

c. Script Languages d. Machine Languages

17. ________ are languages that are designed by people for specific applications

a. Programming Languages b. Natural Languages

c. Script Languages d. Machine Languages

18. ________ are the languages that people speak, such as English, Spanish, and French.

a. Programming Languages **b. Natural Languages**

c. Script Languages d. Machine Languages

19. A ________ is a name that refers to a value.

a. variable b. data type c. Keyword d. Syntax

20. Python is an ________________ language.

a. Logical **b. Interpreted** c. Procedural d. Structural

Part –B (3 x 2 = 6 Marks)

Answer all the Questions

21. What is a Debugging?

Answer:

Programming is a complex process, and because it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called bugs and the process of tracking them down and correcting them is called debugging.

22. List the types of errors

Answer:

Three kinds of errors can occur in a program:

- Syntax errors
- Runtime errors
- Semantic errors

23. Define Algorithm

Answer:

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Part –C (3 x 10 = 30 Marks)

Answer all the Questions

24. a. Explain the types of Errors in Programming

Answer:

Three kinds of errors can occur in a program:

- Syntax errors
- Runtime errors
- Semantic errors

Syntax errors:

- Python can only execute a program if the program is syntactically correct; otherwise, the process fails and returns an error message. Syntax refers to the structure of a program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. This sentence contains a syntax error. So does this one for most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e. e. cummings without spewing error messages. Python is not so forgiving. If there is a single syntax error anywhere in your program, Python will print an error message and quit, and you will not be able to run your program. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.
- Here are some ways to avoid the most common syntax errors:
  - 1. Make sure you are not using a Python keyword for a variable name.
  - 2. Check that you have a colon at the end of the header of every compound statement, including for, while, if, and def statements.
  - 3. Check that indentation is consistent. You may indent with either spaces or tabs but it’s best not to mix them. Each level should be nested the same amount.
  - 4. Make sure that any strings in the code have matching quotation marks.
  - 5. If you have multiline strings with triple quotes (single or double), make sure you have terminated the string properly. An unterminated string may cause an invalid token error at the end of your program, or it may treat the following part of the program as a string until it comes to the next string. In the second case, it might not produce an error message at all!
  - 6. An unclosed bracket—(, {, or [—makes Python continue with the next line as part of the current statement. Generally, an error occurs almost immediately in the next line.

    - 7. Check for the classic = instead of == inside a conditional. If nothing works, move on to the next section...

Runtime errors:

- The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called exceptions because they usually indicate that something exceptional (and bad) has happened. Runtime errors are rare in the simple programs you will see in the first few chapters, so it might be a while before you encounter one.
- **My program does absolutely nothing.**
    - This problem is most common when your file consists of functions and classes but does not actually invoke anything to start execution. This may be intentional if you only plan to import this module to supply classes and functions. If it is not intentional, make sure that you are invoking a function to start execution, or execute one from the interactive prompt. Also see the "Flow of Execution" section below.

Semantic errors:

- The third type of error is the semantic error. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do. The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.
- Try them out by writing simple test cases and checking the results.
    - In order to program, you need to have a mental model of how programs work. If you write a program that doesn't do what you expect, very often the problem is not in the program; it's in your mental model.
    - The best way to correct your mental model is to break the program into its components (usually the functions and methods) and test each component

independently. Once you find the discrepancy between your model and reality, you can solve the problem.

- Of course, you should be building and testing components as you develop the program. If you encounter a problem, there should be only a small amount of new code that is not known to be correct.

(OR)

b. Explain the Concept of Problem Solving

Answer:

CONCEPT OF PROBLEM SOLVING

- Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem solving skills.

PROBLEM DEFINITION

- The problem is *'I want a program which creates a backup of all my important files'*.
- Although, this is a simple problem, there is not enough information for us to get started with the solution. A little more **analysis** is required. For example, how do we specify which files are to be backed up? Where is the backup stored? How are they stored in the backup?
- After analyzing the problem properly, we **design** our program. We make a list of things about how our program should work. In this case, I have created the following list on how *I* want it to work. If you do the design, you may not come up with the same kind of problem - every person has their own way of doing things, this is ok.

1. The files and directories to be backed up are specified in a list.
2. The backup must be stored in a main backup directory.
3. The files are backed up into a zip file.
4. The name of the zip archive is the current date and time.
5. We use the standard **zip** command available by default in any standard Linux/Unix distribution. Windows users can use the Info-Zip program. Note that

you can use any archiving command you want as long as it has a command line interface so that we can pass arguments to it from our script.

THE SOLUTION

- As the design of our program is now stable, we can write the code which is an **implementation** of our solution.

FIRST VERSION

EXAMPLE: 10.1. A BACKUP SCRIPT - THE FIRST VERSION

```
#!/usr/bin/python
# Filename: backup_ver1.py

import os
import time

# 1. The files and directories to be backed up are specified in a list.
source = ['/home/swaroop/byte', '/home/swaroop/bin']
# If you are using Windows, use source = [r'C:\Documents', r'D:\Work'] or
something like that

# 2. The backup must be stored in a main backup directory
target_dir = '/mnt/e/backup/' # Remember to change this to what you will be using

# 3. The files are backed up into a zip file.
# 4. The name of the zip archive is the current date and time
target = target_dir + time.strftime('%Y%m%d%H%M%S') + '.zip'

# 5. We use the zip command (in Unix/Linux) to put the files in a zip archive
zip_command = "zip -qr '%s' %s" % (target, ' '.join(source))
```

```
# Run the backup
if os.system(zip_command) == 0:
    print 'Successful backup to', target
else:
    print 'Backup FAILED'
```

OUTPUT

```
$ python backup_ver1.py
Successful backup to /mnt/e/backup/20041208073244.zip
```

- Now, we are in the **testing** phase where we test that our program works properly. If it doesn't behave as expected, then we have to **debug** our program i.e. remove the *bugs* (errors) from the program.

How It Works

- You will notice how we have converted our *design* into *code* in a step-by-step manner.
- We make use of the os and time modules and so we import them. Then, we specify the files and directories to be backed up in the source list. The target directory is where store all the backup files and this is specified in the target_dir variable. The name of the zip archive that we are going to create is the current date and time which we fetch using the time.strftime() function. It will also have the .zipextension and will be stored in the target_dir directory.
- The time.strftime() function takes a specification such as the one we have used in the above program. The %Y specification will be replaced by the year without the cetury.

The %m specification will be replaced by the month as a decimal number between 01 and 12 and so on. The complete list of such specifications can be found in the [Python Reference Manual] that comes with your Python distribution. Notice that this is similar to (but not same as) the specification used in print statement (using the % followed by tuple).

- We create the name of the target zip file using the addition operator which *concatenates* the strings i.e. it joins the two strings together and returns a new one. Then, we create a string zip_command which contains the command that we are going to execute. You can check if this command works by running it on the shell (Linux terminal or DOS prompt).
- The **zip** command that we are using has some options and parameters passed. The -q option is used to indicate that the zip command should work **q**uietly. The -r option specifies that the zip command should work **r**ecursively for directories i.e. it should include subdirectories and files within the subdirectories as well. The two options are combined and specified in a shorter way as -qr. The options are followed by the name of the zip archive to create followed by the list of files and directories to backup. We convert the source list into a string using the join method of strings which we have already seen how to use.
- Then, we finally *run* the command using the os.system function which runs the command as if it was run from the *system* i.e. in the shell - it returns 0 if the command was successfully, else it returns an error number.
- Depending on the outcome of the command, we print the appropriate message that the backup has failed or succeeded and that's it, we have created a script to take a backup of our important files!

Note to Windows Users

You can set the source list and target directory to any file and directory names but you have to be a little careful in Windows. The problem is that Windows uses the backslash (\) as the directory separator character but Python uses backslashes to represent escape sequences!

So, you have to represent a backslash itself using an escape sequence or you have to use raw strings. For example, use 'C:\\Documents' or r'C:\Documents' but do **not** use'C:\Documents' - you are using an unknown escape sequence \D !

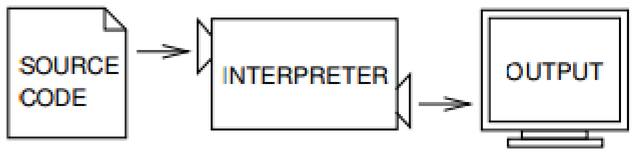
- Now that we have a working backup script, we can use it whenever we want to take a backup of the files. Linux/Unix users are advised to use the executable method as discussed earlier so that they can run the backup script anytime anywhere. This is called the **operation** phase or the **deployment** phase of the software.
- The above program works properly, but (usually) first programs do not work exactly as you expect. For example, there might be problems if you have not designed the program properly or if you have made a mistake in typing the code, etc. Appropriately, you will have to go back to the design phase or you will have to debug your program

25. a. Discuss about Python programming Language

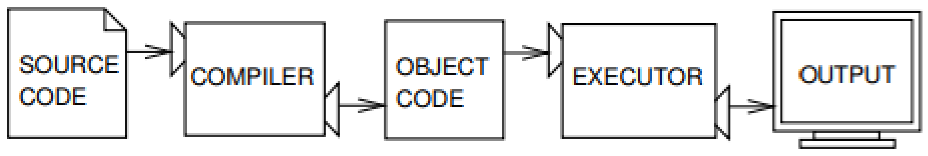
Answer:

The Python programming Language

- The programming language you will be learning is Python. Python is an example of a high-level language; other high-level languages you might have heard of are C, C++, Perl, and Java.
- As you might infer from the name "high-level language," there are also lowlevel languages, sometimes referred to as "machine languages" or "assembly 2 The way of the program languages." Loosely speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages.
- But the advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are portable, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.
- Due to these advantages, almost all programs are written in high-level languages. Low-level languages are used only for a few specialized applications. Two kinds of programs process high-level languages into low-level languages: interpreters and compilers. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.



- A compiler reads the program and translates it completely before the program starts running. In this case, the high-level program is called the source code, and the translated program is called the object code or the executable. Once a program is compiled, you can execute it repeatedly without further translation.



- Python is considered an interpreted language because Python programs are executed by an interpreter. There are two ways to use the interpreter: command line mode and script mode. In command-line mode, you type Python programs and the interpreter prints the result:

```
$ python
Python 2.4.1 (#1, Apr 29 2005, 00:28:56)
Type "help", "copyright", "credits" or "license" for more information.
>>> print 1 + 1
2
```

- The first line of this example is the command that starts the Python interpreter. The next two lines are messages from the interpreter. The third line starts with >>>, which is the prompt the interpreter uses to indicate that it is ready. We typed print 1 + 1, and the interpreter replied 2. Alternatively, you can write a program in a file and use the interpreter to execute the contents of the file. Such a file is called a script. For example, we used a text editor to create a file named latoya.py with the following contents:

```
print 1 + 1
```

- By convention, files that contain Python programs have names that end with .py.
- To execute the program, we have to tell the interpreter the name of the script:

```
$ python latoya.py 2
```

- In other development environments, the details of executing programs may differ. Also, most programs are more interesting than this one.
- Most of the examples in this book are executed on the command line. Working on the command line is convenient for program development and testing, because you can type

programs and execute them immediately. Once you have a working program, you should store it in a script so you can execute or modify it in the future.

(OR)

b. Discuss in detail about the Algorithms

Answer:

ALGORITHMS

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language. From the data structure point of view, following are some important categories of algorithms –
    - **Search** − Algorithm to search an item in a data structure.
    - **Sort** − Algorithm to sort items in a certain order.
    - **Insert** − Algorithm to insert item in a data structure.
    - **Update** − Algorithm to update an existing item in a data structure.
    - **Delete** − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

- Not all procedures can be called an algorithm. An algorithm should have the following characteristics –
    - **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps , and their inputs/outputs should be clear and must lead to only one meaning.
    - **Input** − An algorithm should have 0 or more well-defined inputs.
    - **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
    - **Finiteness** − Algorithms must terminate after a finite number of steps.
    - **Feasibility** − Should be feasible with the available resources.
    - **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm?

- There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

- As we know that all programming languages share basic code constructs like loops , flow-control , etc. These common constructs can be used to write an algorithm.
- We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.
- Example Let's try to learn algorithm-writing by using an example.
    - **Problem − Design an algorithm to add two numbers and display the result.**

        Step 1 − START

        Step 2 − declare three integers a, b & c

        Step 3 − define values of a & b

        Step 4 − add values of a & b

        Step 5 − store output of step 4 to c

        Step 6 − print c

        Step 7 − STOP

    - Algorithms tell the programmers how to code the program.
- Alternatively, the algorithm can be written as –

    Step 1 − START ADD

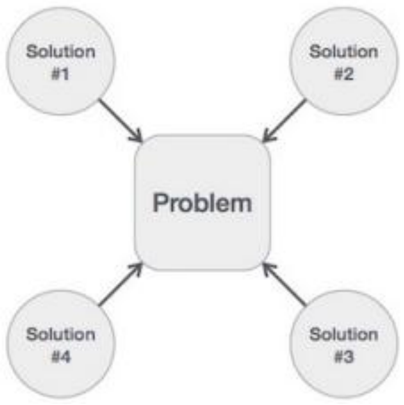
    Step 2 − get values of a & b

    Step 3 − c ← a + b

    Step 4 − display c

    Step 5 − STOP

- In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.
- Writing step numbers, is optional.
- We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



- ➢ Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.

26. a. Explain about Decision table

Answer:

DECISION TABLE

- ➢ A decision table is used to represent conditional logic by creating a list of tasks depicting business level rules. Decision tables can be used when there is a consistent number of a condition that must be evaluated and assigned a specific set of actions to be used when the conditions are finally met.
- ➢ Decision tables are fairly similar to decision trees except for the fact that decision tables will always have the same number of conditions that need to be evaluated and actions that must be performed even if the set of branches being analyzed is resolved to true. A decision tree, on the other hand, can have one branch with more conditions that need to be evaluated than other branches on the tree.
- ➢ Decision tables are a concise visual representation for specifying which actions to perform depending on given conditions. They are algorithms whose output is a set of actions. The information expressed in decision tables could also be represented as decision trees or in a programming language as a series of if-then-else and switch-case statements.

Example

- ➢ The limited-entry decision table is the simplest to describe. The condition alternatives are simple Boolean values, and the action entries are check-marks, representing which of the actions in a given column are to be performed.

- A technical support company writes a decision table to diagnose printer problems based upon symptoms described to them over the phone from their clients.
- The following is a balanced decision table (created by Systems Made Simple).

Printer troubleshooter

		Rules							
Conditions	Printer prints	No	No	No	No	Yes	Yes	Yes	Yes
	A red light is flashing	Yes	Yes	No	No	Yes	Yes	No	No
	Printer is recognized by computer	No	Yes	No	Yes	No	Yes	No	Yes
Actions	Check the power cable			✓					—
	Check the printer-computer cable	✓		✓					—
	Ensure printer software is installed	✓		✓		✓		✓	—
	Check/replace ink	✓	✓				✓		—
	Check for paper jam		✓		✓				—

- Of course, this is just a simple example (and it does not necessarily correspond to the reality of printer troubleshooting), but even so, it demonstrates how decision tables can scale to several conditions with many possibilities.

(OR)

b. Explain about Flowchart with an example

Answer:

FLOWCHARTING

Introduction to planning

- It is important to be able to plan code with the use of flowcharts.
- Even though you can code without a plan, it is not good practice to start coding without a guide to follow.

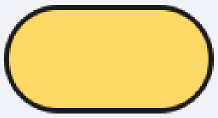
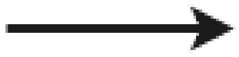
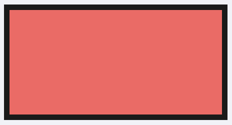
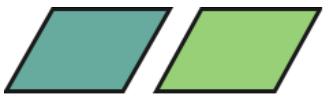
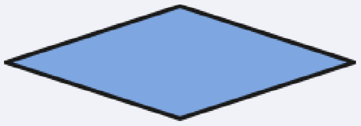
A good plan:

- creates a guide you can follow
- helps you plan efficient structure
- helps communicate to others what your code will do

- Poor planning can result in inefficient, unstructured code known as 'spaghetti code'.

FLOWCHARTS

- A standard way to plan code is with flowcharts.
- Specific parts of the flowchart represent specific parts of your code.

Symbol	Name	What it does in the code
	Start/End	Ovals show a start point or end point in the code
	Connection	Arrows show connections between different parts of the code
	Process	Rectangles show processes e.g. calculations (most things the computer does that does not involve an input, output or decision)
	Input/Output	Parallelograms show inputs and outputs (remember print is normally an input)
	Conditional/ Decision	Diamonds show a decision/conditional (this is normally if, else if/elif, while

and for)

revision

adding text to flowcharts

- There is no one right or wrong way to label flowcharts; you are presenting the structure of your code in a way that humans can understand. Only add extra details to parts of the flowchart when it is not obvious what they do.

Creating Flowcharts

There are many tools you can use to create flow charts.

Pseudo Code

- Pseudo code is an ordered version of your code written for human understanding rather than machine understanding.
- There is no one set way to write pseudo code.
- Good pseudo code should:
    - not be in a specific coding language
    - draft the structure of your code
    - be understandable to humans

e.g. pseudo code

```
if number <= 10 then
  ouput small number sentence
```

python code

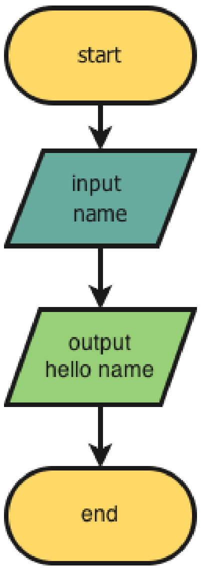
```
if number <=10:
  print("That's a small number!")
```

Note: Pseudo code may seem unnecessary but it is very useful to draft bits of code without worrying about the specifics of making it understandable to a computer.

Example 1

- This program asks the user their name then says "Hello [Name]":

flowchart for Example 1



Note: This flow chart only shows **ovals** and **parallelograms** because the code only has a **start** and **end** and one **input** and one **output**.

pseudo code for Example 1

```
input username
  output "Hello username"
```

Python code for Example 1

```
name= input("What is your name?")
  print("Hello "+ name)
```

Example 2

Description for Example 2

- This program asks the user to "Pick a number:" then prints 1 of 3 different outputs based on how big the number is.

Register Number: ____________

[16CAU501B]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

Coimbatore-641021.

(For the candidates admitted from 2016 onwards)

COMPUTER APPLICATIONS

SECOND INTERNAL EXAMINATION- AUGUST 2018

Fifth Semester

PYTHON PROGRAMMING

Date & Session: 14.08.2018 & FN	**Duration : 2 Hours**
Class: III BCA	**Maximum : 50 Marks**

PART-A (20 X 1 = 20 Marks)

Answer all the Questions

1. What error occurs when you execute? apple = mango
   a) SyntaxError b) NameError
   c) ValueError d) TypeError
2. Which of the following is not a complex number?
   a) k = 2 + 3j b) k = 2 + 3l
   c) k = complex(2, 3) d) k = 2 + 3J
3. What is the type of inf?
   a) Boolean b) Integer
   c) Float d) Complex
4. What do ~4 evaluate to?
   a) -5 b) -4
   c) -3 d) 3
5. Which of the following is a Python tuple?
   a) [1, 2, 3] b) (1, 2, 3)
   c) {1, 2, 3} d) {}
6. If a={5,6,7,8}, which of the following statements is false?
   a) print(len(a)) b) print(min(a))
   c) a.remove(5) d) a[2]=45
7. What is the output of this expression if x=22.19? >>> print("%5.2f"%x)
   a) 56 b) 56.24
   c) 56.23 d) 56.236
8. __________is a thing to which a variable can refer.
   a) number b) object
   c) list d) element
9. Operators with the same precedence are evaluated in which manner?
   a) Left to Right b) Right to Left
   c) Can't say d) Right
10. Python is an example of a ________________
    a) low level language b) high level language
    c) middle level language d) assembly language
11. _________ is an instruction that causes the Python interpreter to display a value.
    a) input statement b) print statement
    c) display d) statement

12. Evaluate the expression given below if A= 16 and B = 15. A % B // A

a) 0	b) 1
c) 1	d) -1

13. Which of the following operators has its associatively from right to left?

a) +	b) //
c) %	d) **

14. Which of these in not a core data type?

a) Lists	b) Dictionary
c) Tuples	d) Class

15. Which of the following will run without errors?

a) round(45.8)	b) round(6352.898,2,5)
c) round()	d) round(7463.123,2,1)

16. Is Python case sensitive when dealing with identifiers?

a) yes	b) no
c) machine dependent	d) machine independent

17. What is the maximum possible length of an identifier?

a) 31 characters	b) 63 characters
c) 79 characters	d) Identifiers can be of any length.

18. Which of the following is an invalid variable?

a) my_string_1	b) 1st_string
c) __	d) foo

19. What is the result of round(0.5) – round(-0.5)?

a) 1	b) 2
c) 0	d) -1

20. What does 3 ^ 4 evaluate to?

a) 81	b) 12
c) 0.75	d) 7

Part –B (3 x 2 = 6 Marks)
Answer all the Questions

21. What is meant by Programming?
22. Define Variables
23. List some features of python

Part –C (3 x 8 = 24 Marks)
Answer all the Questions

24. a. Difference between Top down and Bottom up programming methodologies

(OR)

b. Explain the Data types in python

25. a. Explain how to create a variable in python with example

(OR)

b. Write a python program to find the sum of natural number.

26. a. Explain the Structure of Python Programming

(OR)

b. Explain the Features of Python.

Register Number: ____________
[16CAU501B]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)
Coimbatore-641021.
(For the candidates admitted from 2016 onwards)
COMPUTER APPLICATIONS
SECOND INTERNAL EXAMINATION- AUGUST 2018
Fifth Semester
PYTHON PROGRAMMING

Date & Session: 14.08.2018 & FN — **Duration : 2 Hours**
Class: III BCA — **Maximum : 50 Marks**

PART-A (20 X 1 = 20 Marks)
Answer all the Questions

1. What error occurs when you execute? apple = mango
   a) SyntaxError **b) NameError**
   c) ValueError d) TypeError
2. Which of the following is not a complex number?
   a) k = 2 + 3j **b) k = 2 + 3l**
   c) k = complex(2, 3) d) k = 2 + 3J
3. What is the type of inf?
   a) Boolean b) Integer
   c) Float d) Complex
4. What do ~4 evaluate to?
   a) -5 b) -4
   c) -3 d) 3
5. Which of the following is a Python tuple?
   a) [1, 2, 3] **b) (1, 2, 3)**
   c) {1, 2, 3} d) {}
6. If a={5,6,7,8}, which of the following statements is false?
   a) print(len(a)) b) print(min(a))
   c) a.remove(5) **d) a[2]=45**
7. What is the output of this expression if x=22.19? >>> print("%5.2f"%x)
   a) 56 **b) 56.24**
   c) 56.23 d) 56.236
8. __________is a thing to which a variable can refer.
   a) number **b) object**
   c) list d) element
9. Operators with the same precedence are evaluated in which manner?
   a) Left to Right b) Right to Left
   c) Can’t say d) Right
10. Python is an example of a ________________
    a) low level language **b) high level language**
    c) middle level language d) assembly language

11. **_________** is an instruction that causes the Python interpreter to display a value.

a) input statement | **b) print statement**
c) display | d) statement

12. Evaluate the expression given below if A= 16 and B = 15. A % B // A

a) 0 | b) 1
c) 1 | d) -1

13. Which of the following operators has its associatively from right to left?

a) + | b) //
c) % | **d) ****

14. Which of these in not a core data type?

a) Lists | b) Dictionary
c) Tuples | **d) Class**

15. Which of the following will run without errors?

a) round(45.8) | b) round(6352.898,2,5)
c) round() | d) round(7463.123,2,1)

16. Is Python case sensitive when dealing with identifiers?

a) yes | b) no
c) machine dependent | d) machine independent

17. What is the maximum possible length of an identifier?

a) 31 characters | b) 63 characters
c) 79 characters | **d) Identifiers can be of any length.**

18. Which of the following is an invalid variable?

a) my_string_1 | **b) 1st_string**
c) __ | d) foo

19. What is the result of round(0.5) – round(-0.5)?

a) 1 | **b) 2**
c) 0 | d) -1

20. What does 3 ^ 4 evaluate to?

a) 81 | b) 12
c) 0.75 | **d) 7**

Part –B (3 x 2 = 6 Marks)
Answer all the Questions

21. What is meant by Programming?

Answer:

Programming is the process of taking an algorithm and encoding it into a notation, a programming language, so that it can be executed by a computer. Although many programming languages and many different types of computers exist, the important first step is the need to have the solution. Without an algorithm there can be no program.

22. Define Variables

Answer:

Names and Assignment

We used variables for the first time: a and b in the example. Variables are used to store data; in simple terms they are much like variables in algebra and, as mathematically-literate students, we hope you will find the programming equivalent fairly intuitive.

Variables have names like a and b above, or x or fred or z1. Where relevant you should give your variables a descriptive name, such as firstname or height 3.2. Variable

names must start with a letter and then may consist only of alphanumeric characters (i.e. letters and numbers) and the underscore character, ``_''. There are some reserved words which you cannot use because Python uses them for other things; these are listed in Appendix B.
We assign values to variables and then, whenever we refer to a variable later in the program, Python replaces its name with the value we assigned to it. This is best illustrated **by a simple example:**

```
>>> x = 5
>>> print x
5
```

23. List some features of python
Answer:
Features of python
- ➢ Simple
- ➢ Easy to learn
- ➢ Free and open source
- ➢ High Level Language
- ➢ Portable
- ➢ Interpreted
- ➢ Object Oriented
- ➢ Extensible
- ➢ Embeddable
- ➢ Extensive Libraries

Part –C (3 x 8 = 24 Marks)
Answer all the Questions

24. a. Difference between Top down and Bottom up programming methodologies
Answer:
Top-down Programming
- ➢ Top-down programming focuses on the use of modules. It is therefore also known as modular programming. The program is broken up into small modules so that it is easy to trace a particular segment of code in the software program. The modules at the top level are those that perform general tasks and proceed to other modules to perform a particular task. Each module is based on the functionality of its functions and procedures. In this approach, programming begins from the top level of hierarchy and progresses towards the lower levels. The implementation of modules starts with the main module. After the implementation of the main module, the subordinate modules are implemented and the process follows in this way. In top-down programming, there is a risk of implementing data structures as the modules are dependent on each other and they have to share one or more functions and procedures. In this way, the functions and procedures are globally visible. In addition to modules, the top-down programming uses sequences and the nested levels of commands.

Disadvantages of top-down programming

- Top-down programming complicates testing. Noting executable exists until the very late in the development, so in order to test what has been done so far, one must write stubs .
- Furthermore, top-down programming tends to generate modules that are very specific to the application that is being written, thus not very reusable.
- But the main disadvantage of top-down programming is that all decisions made from the start of the project depend directly or indirectly on the high-level specification of the application. It is a well-known fact that this specification tends to change over time. When that happens, there is a great risk that large parts of the application need to be rewritten.

How does top-down programming work?

- Top-down programming tends to generate modules that are based on functionality, usually in the form of functions or procedures. Typically, the high-level specification of the system states functionality. This high-level description is then refined to be a sequence or a loop of simpler functions or procedures, that are then themselves refined, etc.
- In this style of programming, there is a great risk that implementation details of many data structures have to be shared between modules, and thus globally exposed. This in turn makes it tempting for other modules to use these implementation details, thereby creating unwanted dependencies between different parts of the application.

Bottom-up Programming

- Bottom-up programming refers to the style of programming where an application is constructed with the description of modules. The description begins at the bottom of the hierarchy of modules and progresses through higher levels until it reaches the top. Bottom-up programming is just the opposite of top-down programming. Here, the program modules are more general and reusable than top-down programming.
- It is easier to construct functions in bottom-up manner. This is because bottom-up programming requires a way of passing complicated arguments between functions. It takes the form of constructing abstract data types in languages such as C++ or Java, which can be used to implement an entire class of applications and not only the one that is to be written. It therefore becomes easier to add new features in a bottom-up approach than in a top-down programming approach.

Advantages of bottom-up programming

- Bottom-up programming has several advantages over top-down programming .
- Testing is simplified since no stubs are needed. While it might be necessary to write test functions, these are simpler to write than stubs, and sometimes not necessary at all, in particular if one uses an interactive programming environment such as Common Lisp or GDB.
- Pieces of programs written bottom-up tend to be more general, and thus more reusable, than pieces of programs written top-down. In fact, one can argue that the purpose bottom-up programming is to create an application-specific language. Such a language is suitable for implementing an entire class of applications, not only the one that is to be written. This fact greatly simplifies maintenance, in particular adding new features to the application. It also makes it possible to delay the final decision concerning the exact functionality of the application. Being able

to delay this decision makes it less likely that the client has changed his or her mind between the establishment of the specifications of the application and its implementation.

(OR)

b. Explain the Data types in python

Answer:

Data Types

- Variables need not be numeric. There are several types. The most useful are described below:
- **Integer: Any whole number:**

```
>>> myinteger = 0
>>> myinteger = 15
>>> myinteger = -23
>>> myinteger = 2378
```

- **Float: A floating point number, i.e. a non-integer.**

```
>>> myfloat = 0.1
>>> myfloat = 2.0
>>> myfloat = 3.14159256
>>> myfloat = 1.6e-19
>>> myfloat = 3e8
```

- Note that although 2 is an integer, by writing it as 2.0 we indicate that we want it stored as a float, with the precision that entails.3.4 The last examples use exponentials, and in maths would be written and . If the number is given in exponential form it is stored with the precision of floating point whether or not it is a whole number.
- **String:** A string or sequence of characters that can be printed on your screen. They must be enclosed in either single quotes or double quotes--not a mixture of the two, e.g.

```
>>> mystring = "Here is a string"
>>> mystring = 'Here is another'
```

- **Arrays and Lists:** These are types which contain more than one element, analogous to vectors and matrices in mathematics. Their discussion is deferred until Section 3.10 ``Arrays''. For the time being, it is sufficient to know that a list is written by enclosing it in square brackets as follows: **mylist = [1, 2, 3, 5]**
- **If it is not sure what type a variable is, we can use the type () function to inspect it:**

```
>>> type(mystring)
<type 'str'>
```

- 'str' tells you it is a string. we might also get <type 'int'> (integer) and <type 'float'> (float) 3.5.
- **Tuple:** A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.

  Example

  Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

  Example

  We cannot change values in a tuple:

```
thistuple = ("apple", "banana", "cherry")
thistuple[1] = "blackcurrant" # test changeability
print(thistuple)
```

25. a. Explain how to create a variable in python with example

Answer:

- **Names and Assignment**
  - We used variables for the first time: a and b in the example. Variables are used to store data; in simple terms they are much like variables in algebra and, as mathematically-literate students, we hope you will find the programming equivalent fairly intuitive.
  - Variables have names like a and b above, or x or fred or z1. Where relevant you should give your variables a descriptive name, such as firstname or height 3.2. Variable names must start with a letter and then may consist only of alphanumeric characters (i.e. letters and numbers) and the underscore character, ``_". There are some reserved words which you cannot use because Python uses them for other things; these are listed in Appendix B.
  - We assign values to variables and then, whenever we refer to a variable later in the program, Python replaces its name with the value we assigned to it. This is best illustrated by a simple example:

```
>>> x = 5
>>> print x
5
```

  - You assign by putting the variable name on the left, followed by a single =, followed by what is to be stored. To draw an analogy, you can think of variables as named boxes. What we have done above is to label a box with an ``x", and then put the number 5 in that box.
  - There are some differences between the syntax 3.3 of Python and normal algebra which are important. Assignment statements read right to left only. x = 5 is fine, but 5 = x doesn't make sense to Python, which will report a SyntaxError. If you like, you can think of the equals sign as an arrow pointing from the number on the right, to the variable name on the left: $x \leftarrow 5$ and read the expression as ``assign 5 tox" (or, if you prefer, as ``x becomes 5"). However, we can still do many of things you might do in algebra, like:

```
>>> a = b = c = 0
```

  - Reading the above right to left we have: ``assign 0 to c, assign c to b, assign b to a".

```
>>> print a, b, c
0 0 0
```

  - There are also statements that are alegbraically nonsense, that are perfectly sensible to Python (and indeed to most other programming languages). The most common example is incrementing a variable:

```
>>> i = 2
>>> i = i + 1
>>> print i
3
```

- The second line in this example is not possible in maths, but makes sense in Python if you think of the equals as an arrow pointing from right to left. To describe the statement in words: on the right-hand side we have looked at what is in the box labelled i, added 1 to it, then stored the result back in the same box

(OR)

b. Write a python program to find the sum of natural number.

Answer:

# Python program to find the sum of natural numbers up to n where n is provided by user

```
# change this value for a different result
num = 16

# uncomment to take input from the user
#num = int(input("Enter a number: "))

if num < 0:
  print("Enter a positive number")
else:
  sum = 0
  # use while loop to iterate un till zero
  while(num > 0):
     sum += num
     num -= 1
  print("The sum is",sum)
```

Output

```
The sum is 136
```

Note:

- To test the program, change the value of num.
- Here, we store the number in num and display the sum of natural numbers up to that number. We use while loop to iterate until the number becomes zero.
- We could have solved the above problem without using any loops using a formula directly.
- Modify the above program to find the sum of natural numbers using the formula below.

n*(n+1)/2

For example, if n = 16, the sum would be (16*17)/2 = 136.

26. a. Explain the Structure of Python Programming

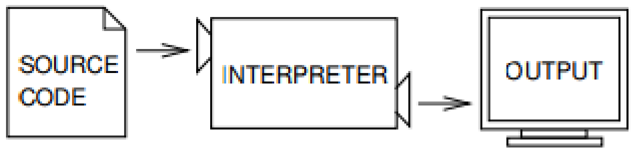
Answer:

<u>STRUCTURE OF PYTHON</u>

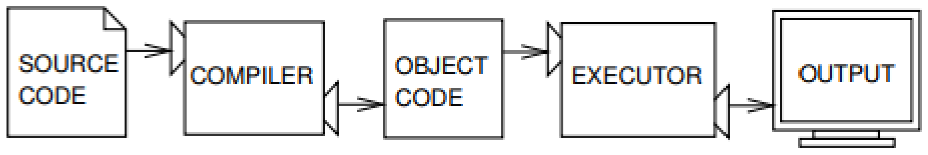
<u>The Python programming Language</u>

- The programming language you will be learning is Python. Python is an example of a high-level language; other high-level languages you might have heard of are C, C++, Perl, and Java.

- As you might infer from the name "high-level language," there are also lowlevel languages, sometimes referred to as "machine languages" or "assembly 2 The way of the program languages." Loosely speaking, computers can only execute programs written in low-level languages. Thus, programs written in a high-level language have to be processed before they can run. This extra processing takes some time, which is a small disadvantage of high-level languages.
- But the advantages are enormous. First, it is much easier to program in a high-level language. Programs written in a high-level language take less time to write, they are shorter and easier to read, and they are more likely to be correct. Second, high-level languages are portable, meaning that they can run on different kinds of computers with few or no modifications. Low-level programs can run on only one kind of computer and have to be rewritten to run on another.
- Due to these advantages, almost all programs are written in high-level languages. Low-level languages are used only for a few specialized applications. Two kinds of programs process high-level languages into low-level languages: interpreters and compilers. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.



- A compiler reads the program and translates it completely before the program starts running. In this case, the high-level program is called the source code, and the translated program is called the object code or the executable. Once a program is compiled, you can execute it repeatedly without further translation.



- Python is considered an interpreted language because Python programs are executed by an interpreter. There are two ways to use the interpreter: command line mode and script mode. In command-line mode, you type Python programs and the interpreter prints the result:

```
$ python
Python 2.4.1 (#1, Apr 29 2005, 00:28:56)
Type "help", "copyright", "credits" or "license" for more
information.
>>> print 1 + 1
2
```

- The first line of this example is the command that starts the Python interpreter. The next two lines are messages from the interpreter. The third line starts with >>>, which is the prompt the interpreter uses to indicate that it is ready. We typed print 1 + 1, and the interpreter replied 2. Alternatively, you can write a program in a file and use the interpreter to execute the contents of the file. Such a file is called a script. For example, we used a text editor to create a file named latoya.py with the following contents:

```
print 1 + 1
```

- By convention, files that contain Python programs have names that end with .py.
- To execute the program, we have to tell the interpreter the name of the script:

```
$ python latoya.py 2
```

- In other development environments, the details of executing programs may differ. Also, most programs are more interesting than this one.
- Most of the examples in this book are executed on the command line. Working on the command line is convenient for program development and testing, because you can type programs and execute them immediately. Once you have a working program, you should store it in a script so you can execute or modify it in the future.

(OR)

b. Explain the Features of Python.
Answer:

FEATURES OF PYTHON

- **Simple**
  - Python is a simple and minimalistic language. Reading a good Python program feels almost like reading English, although very strict English! This pseudo-code nature of Python is one of its greatest strengths. It allows you to concentrate on the solution to the problem rather than the language itself.
- **Easy to Learn**
  - As you will see, Python is extremely easy to get started with. Python has an extraordinarily simple syntax, as already mentioned.
- **Free and Open Source**
  - Python is an example of a FLOSS (Free/LibrÃ© and Open Source Software). In simple terms, you can freely distribute copies of this software, read it's source code, make changes to it, use pieces of it in new free programs, and that you know you can do these things. FLOSS is based on the concept of a community which shares knowledge. This is one of the reasons why Python is so good - it has been created and is constantly improved by a community who just want to see a better Python.

- **High-level Language**
    - When you write programs in Python, you never need to bother about the low-level details such as managing the memory used by your program, etc.
- **Portable**
    - Due to its open-source nature, Python has been ported (i.e. changed to make it work on) to many platforms. All your Python programs can work on any of these platforms without requiring any changes at all if you are careful enough to avoid any system-dependent features.
    - You can use Python on Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and even PocketPC !
- **Interpreted**
    - This requires a bit of explanation.
    - A program written in a compiled language like C or C++ is converted from the source language i.e. C or C++ into a language that is spoken by your computer (binary code i.e. 0s and 1s) using a compiler with various flags and options. When you run the program, the linker/loader software copies the program from hard disk to memory and starts running it.
    - Python, on the other hand, does not need compilation to binary. You just *run* the program directly from the source code. Internally, Python converts the source code into an intermediate form called bytecodes and then translates this into the native language of your computer and then runs it. All this, actually, makes using Python much easier since you don't have to worry about compiling the program, making sure that the proper libraries are linked and loaded, etc, etc. This also makes your Python programs much more portable, since you can just copy your Python program onto another computer and it just works!
- **Object Oriented**
    - Python supports procedure-oriented programming as well as object-oriented programming. In *procedure-oriented* languages, the program is built around procedures or functions which are nothing but reusable pieces of programs. In *object-oriented* languages, the program is built around objects which combine data and functionality. Python has a very powerful but simplistic way of doing OOP, especially when compared to big languages like C++ or Java.
- **Extensible**
    - If you need a critical piece of code to run very fast or want to have some piece of algorithm not to be open, you can code that part of your program in C or C++ and then use them from your Python program.
- **Embeddable**
    - You can embed Python within your C/C++ programs to give 'scripting' capabilities for your program's users.
- **Extensive Libraries**
    - The Python Standard Library is huge indeed. It can help you do various things involving regular expressions, documentation generation, unit testing, threading, databases, web browsers, CGI, ftp, email, XML, XML-RPC, HTML, WAV files, cryptography, GUI (graphical user interfaces), Tk, and

other system-dependent stuff. Remember, all this is always available wherever Python is installed. This is called the 'Batteries Included' philosophy of Python.

- Besides, the standard library, there are various other high-quality libraries such as wxPython, Twisted, Python Imaging Library and many more.