Unit 1: Introduction, Variables and Data Types

Unit at a glance:

1.1 Variables

Python codes are divided into identifiers / variables. A variable can be used to define a name or an identifier. Technically, all these names are called identifier. An identifier is a name used to identify a variable, function, class, module, or other object. Generlly, we declare variables / identifiers to write program code.

1.1.1 Declaring Variables

Variables are declared according to the Python character sets. A variable is declared to hold some value. Variables are examples of identifiers. Identifgiers are names given to identify something. Python has some rules about how identifiers or variables are formed. These are some basic rules to declare variables in Python.

- Variable names can be as short as a single letter either in uppercase or lowercase.
- Variable names must begin with a letter of the alphabet or an underscore ('_').
- After the first letter or underscore, they an contain letters, numbers and additional underscores. For example, totalCost, y2014, acc_bal, d_o_b, _sysVal, __my_name, __my__name, etc.
- Variable names are case-sensitive. For example, myname and myName are not the same names, they are two different variables. Note the lowercase n in the former and the uppercase N in the latter.
- Variable names should not be reserved words or keywords. That is, variables cannot ave the same name as a Python command or a library function. For example, while, if , etc. cannot be declared as variable name.
- No blank spaces are allowed between variable name. Also, it cannot use any special characters (such as @, $, %, etc.) other than underscore ('_').
- Variables or identifiers can be of unlimited length.

The following variable names are valid:

roll_no	Salary	sep98_score	index_age	Percentage
amount	dob	GrossPay	Cust_no	D_o_j
i	__my_name	Name_23	alb2_c3	

Some invalid variable names which produces syntax errors are as follows:

>>> 98Sep_score SyntaxError: invalid syntax	because variable start with a digit
>>> Your Age SyntaxError: invalid syntax	because variable contains space

>>> while SyntaxError: invalid syntax	because variable is a reserve word
>>> myname@ SyntaxError: invalid syntax	because variable contains a special character

1.1.2 Creating Variables

The assignment statement (=) assigns a value to a variable. In mathematics, the '=' operator has a different meaning. In an equation, the '=' operator is an equality operator. The left side of the equation is equal to the right side. The usual assignment operator is '='. The basic assignemtn statement has this form:

<variable>=<expr>

Here, <variable> is an identifier and <expr> is an expression. The semantics of the assignment is that the expression on the right side is evaluated to produce a value, which is then associated with the variable named on the left side.

For example,

```
>>>Num1=20
>>>Num2=40
>>>sumN=Num1+Num2
>>>sumN
60
```

Variables can also be assigned values in different forms. These are:

- **Assigning an expression:** An expression can be assigned into a variable. For example,

```
>>>x=10
>>>x=3.9*x*(1-x)
>>>x
-351.0
>>>Celsius=40
>>>Fahrenheit=9.0/50*Celsius+32
>>>Celsius, Fahrenheit
(40, 104.0)
```

- **A variable can be assigned many times:** It always retains the value of the most recent assignment. For example,

```
>>>Var1=7
>>>Var1
7
>>>Var1=0
>>>Var1
0
>>>Var1=Var1+10
>>>Var1
10
```

Notice that the above lines assigned many values to same variable and produces different result. But the last assigned value is the correct value for Var1. The last assignment shows how the current value of a variable can be used to update its value.

Storage location is a box in computer memory, where we put a value with the name of a variable. When the variable changes, the old value is erased and a new value is written.

Fig. 1 shows the effect of **`Var1=Var1+1`**

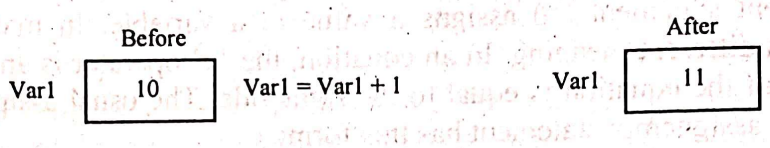


Fig: 1 Variable as box view of x = x + 1

- **Assigning a value to multiple variables:** In Python, we can assign a value to multiple variables. For example,

```
>>>a=b=c=d=5
>>>(a, b, c, d)
(5, 5, 5, 5)
```

Here, all the variables a, b, c and d are assigned a single value called 5.

- **Multiple assignments in a single line:** You can even assign values to multiple variables in asingle line. While assigning, the left and the right side must have the same number of elements. The general format is:

`<var1>,<var2>,<var3>,….<varN>=<expr1>,<expr2>,<expr3>,….<exprN>`

This is called simultaneous assignment. Sementically, this tells Python to evaluate all the expressions on the right-hand aide and then assign these values to the corresponding variables named on the left-hand side. For example,

```
>>>a1, a2, a3=10, 20, 30
>>>a1, a2, a3
(10, 20, 30)
```

Here, Python evaluates the entire right-hand side (**10, 20, 30**) of the = statement. Then, it matches values with destinations on the left-hand side (**a1, a2,** and **a3**), i.e., the rightmost value **30** is assigned to **a3, 20** to **a2** and **10** to **a1**. If the lists are of different lengths, an exception is raised and the program stops.

- **Using sequences to assign multiple values at once:** Using assignment operator, we can assign sequences like tuple and list values into multiple values. For example,

```
>>>days=(31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
>>>(Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov,
Dec)=days
>>>Jan
31
>>>Feb
28
```

From the above declaration, **days** is a tuple of twelve elements and **(Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec)** is a tuple of three variables. Assigning one to

the other assigns each of the values of days to each of the variables **(Jan, Feb, ...)** in order.

Let us take another example considering a list,

```
>>>WeekDay=[0, 1, 2, 3, 4, 5, 6]
>>>(Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday)=WeekDay
>>>Sunday
0
>>>Monday
1
```

1.2 Data Types in Python

A data type is just an interpretation applied to a string of bytes. On declaration point of view Python does declare any data type for any variable, but it creates a data type according to the value. Variables can hold values of different types called **data types.** The basic types are numbers and strings.

Python has a great set of useful **data types.** The data we manipulate in Python consists of objects and names. Each object in Python has three key attributes a **type**, a **value** and an id (i.e., memory address).

- An object represents some value. It may be a simple value like the number **20**, or a very complex value, like an object that describes the results of a student or status of an employee. For example, if you store a value say **20** to **Num**, then in Python we write:

  ```
  >>>Num=20
  ```

- Here, the object is **20** and its type is int. The object is now assigned into variable **Num** which now holds a data type called **int**.
- The type of an object determines what operations can be performed on it. For example, an object of Python's (integer) type holds a whole number and you can use operators like "+" and "–" to add and subtract objects of that type. For example, if you want to know the object type of **Num** (whose value is **20**), then in Python we write:

  ```
  >>>type(Num)
  <type 'int'>
  ```

  Notice that the object name of **Num** is defined as `<type 'int'>`, i.e., class int. The **type()** function tells the data type of the object Num.
- A Python name is like a luggage tag: The **id()** function represents the location in memory of the object and does not get change once it is created. For example, if you want to know the memory location of Num variable, then in Python we write:

  ```
  >>>id(Num)
  20380668
  ```

  Here, the memory location of **Num** object is **20380668**. Notice that the memory location (address) is not always fix in every computer, it may vary depending upon the memory space.

What does a data type mean?

Data type of an object determines what values it can have and what operations can be performed on it. Many current languages such as C and C++, a Python name is not associated with a type. a name is associated with an object and that object has a type. the association between a name and an object is called a binding. For example, after executing this statement,

```
Num=20
```

We say that the name is Num and the object 20, which has a type int.

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard types that are used to define the operations possible on them and the storage method for each of them.

Python has the following fundamental or base data types and sequence type data types:

- **Fundamental data types or native data types** like **int, float, complex,** etc.
- **Sequence types** like **string, list, tuple,** etc.
- **Set types** like **dictionary** or **mapping, set,** etc.

In Python, while the value that a variable points to has a type, the variable itself has no strict type in its definition. You can reuse the same variable to point to an object of a different type. Let us see the following examples,

```
>>>Val=10
>>>print "We assign a numeric value:", Val
```

We assign a numeric value: 10

Here, **Val** holds a integer type value. Also notice that we use the **print** statement to print the output.

```
>>>Val=10.5
>>>print "We assign a floating-point value:", Val
```

We assign a numeric value: 10.5

Here, **Val** holds a float type value.

```
>>>Val=2**32
>>>print "We assign a long value:", Val
```

We assign a long value: 4294967296

Here, **Val** holds a long type value.

```
>>>Val="Hello World!"
>>>print "We assign a string value:", Val
```

We assign a string value: Hello World!

Here, **Val** holds a string type value.

```
>>>Val=['A-10/130', 'Sangam Vihar', 'New Friends colony', 'New Delhi - 110004']
>>>print "We assign list of value:", Val
```

We assign list of values: ['A-10/130', 'Sangam Vihar', 'New Friends colony', 'New Delhi – 110004']

Here, **Val** holds list types values.

Similarly, you can assign any type of values into a single variable.

Short Answer Type Questions

A. Choose the correct answer from the given alternatives in each of the following:

1. What does pip stand for python? [WBSCTE 2022]
   (a) Unlimited length
   (b) All private members must have leading and trailing underscores
   (c) Preferred Installer Program
   (d) None of these

Answer: (c)

2. What will be the output of print(10, 20, 30, sep= "/") [WBSCTE 2022]
   (a) 10 20 30 (b) 10/20/30 (c) 10\20\30 (d) error

Answer: (b)

3. Which of the following is a Python tuple? [WBSCTE 2022]
   (a) {1, 2, 3} (b) {} (c) [1, 2, 3] (d) (1, 2, 3)

Answer: (d)

4. Suppose a list with name arr, contains 5 elements. You can get the 2nd element from the list using: [Model Question]
   (a) arr[-2] (b) arr[2] (c) arr[-1] (d) arr[1]

Answer: (d)

5. Which of these is not a core data type? [Model Question]
   (a) List (b) Tuple (c) Dictionary (d) Class

Answer: (d)

6. Which of these about a dictionary is false? [Model Question]
   (a) The values of a dictionary can be accessed using keys
   (b) The keys of a dictionary can be accessed using values
   (c) Dictionaries aren't ordered
   (d) Dictionaries are mutable

Answer: (b)

7. Which of the following data type is used to store values in Key & Value format? [Model Question]
   (a) List (b) Tuple (c) Dictionary (d) Set

Answer: (c)

[Model Question]

8. What is the output of following: set([1,1,2,3,4,2,3,4])
(a) [1,1,2,3,4,2,3,4] (b) {1,2,3,4}
(c) {1,1,2,3,4,2,3,4} (d) Invalid Syntax

Answer: (b)

[Model Question]

9. What datatype is the object below?
L = [1, 23, 'hello', 1].
(a) list (b) dictionary (c) array (d) tuple

Answer: (a)

B. Fill in the blanks in the following statements:

[WBSCTE 2022]

10. All keyword is Python are in ***lower case***.

11.
```
a = 5, b = 6
a, b = b, a
print(a,b)
```
6 5 is the final output. [WBSCTE 2022]

12.
```
a = 10<<12
b=34>>10, c = 7 | 13, d = 12 ^ 13
print(a, b, c, d)
```
40960 0 15 1 is the output. [WBSCTE 2022]

13. To find x^y, you will use ***x**y*** operator. [WBSCTE 2022]

14. A program in python is stored with a ***.py*** extension. [WBSCTE 2022]

15. The value of the expression set ([1,1,2,3]) is ***{1,2,3}***. [WBSCTE 2022]

16. The 13//(4+3%5) evaluates to ***1***. [WBSCTE 2022]

17. Import time
a= time.localtime()
c= time. asctime(a)
print(c)

Thu Jan 12 5:00:00 2023 is the output. [WBSCTE 2022]

18. IDLE stands for ***Integrated Development and Learning Environment***. [Model Question]

19. ***Variable*** is a named memory location that stores data and whose value may change during program execution. [Model Question]

20. A ***Datatype*** determines the type of data stored in a variable. [Model Question]

21. ***int, float, str and bool*** are the datatypes in Python. [Model Question]

C. Answer the following questions:

22. What is an Interpreted language? [WBSCTE 2022]

Answer:
An interpreted language is a programming language that is generally interpreted, without compiling a program into machine instructions.

23. How to comment multiple lines in python? [WBSCTE 2022]

Answer:
The simplest way of multiline commenting in python is to use consecutive single line comments using a hashtag. We can also add a multiline string (triple quotes) in our code, and place our comment inside it.

24. How will you capitalize the first letter of string? [WBSCTE 2022]

Answer:
To capitalize the first character of a string, We can use the charAt() to separate the first character and then use the toUpperCase() function to capitalize it.

25. What are Python's dictionaries? [WBSCTE 2022]

Answer:
Dictionaries are Python's implementation of a data structure that is more generally known as an associative array. A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value.

26. What is PEP 8? [WBSCTE 2022]

Answer:
PEP 8, sometimes spelled PEP8 or PEP-8, is a document that provides guidelines and best practices on how to write Python code.

27. What is the purpose of the PYTHONPATH environment variable? [WBSCTE 2022]

Answer:
Pythonpath is an environment variable that is used to specify the location of Python libraries. It is typically used by developers to ensure that their code can find the required Python libraries when it is run.

28. Is python a case-sensitive language? [WBSCTE 2022]

Answer:
Yes, Python is a case-sensitive language, i.e., it treats uppercase and lowercase characters differently.

29. What are membership operators? [WBSCTE 2022]

Answer:
Membership operators are operators used to validate the membership of a value. It test for membership in a sequence, such as strings, lists, or tuples.
in operator - The 'in' operator is used to check if a value exists in a sequence or not.
'not in' operator- Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

30. What is the difference between Python Arrays and lists? **[WBSCTE 2022]**
Answer:
List is used to collect items that usually consist of elements of multiple data types. An array is also a vital component that collects several items of the same data type. List cannot manage arithmetic operations. Array can manage arithmetic operations.

31. Write a Python code to add the values of two variables. **[WBSCTE 2022]**
Answer:
```
num1 = 15
num2 = 12
# Adding two nos
sum = num1 + num2
# printing values
print("Sum of {0} and {1} is {2}" .format(num1, num2, sum))
```

32. B = 3, A=2. Swap the values of these variables. **[WBSCTE 2022]**
Answer:
```
A = 2
B = 3
# Swapping of two variables
# without using third variable
A, B = B, A
print("Value of x:", A)
print("Value of y:", B)
```

33. How can the ternary operators be used in python? **[WBSCTE 2022]**
Answer:
Syntax:
```
[on_true] if [expression] else [on_false]
```
Example:
```
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

34. What is slicing? [WBSCTE 2022]

Answer:
Slicing in Python is a feature that enables accessing parts of sequences like strings, tuples and lists. You can also use them to modify or delete the items of mutable sequences such as lists.

35. What is Python? [Model Question]

Answer:
Python is an easy to learn, general-purpose, dynamic, interpreted, interactive, high-level, general purpose, multi-platform, powerful programming language. it has efficient high-level data structures and a simple but effective approach to object-oriented programming.

36. How can you start Python at windows command prompt? [Model Question]

Answer:
If you set the Python path for windows, then you can start Python with the following:

```
C:>python
```

37. How can you execute Python script at windows command prompt? [Model Question]

Answer:
A Python script can be executed at command line by invoking the interpreter on your application, as in the following:

```
C:>python script.py
```

38. What is IDLE? [Model Question]

Answer:
IDLE ("Interactive Development Environment") is an older-style environment for developing Python programs ("scripts") under windows and other operating systems

39. What is interactive mode? [Model Question]

Answer:
When commands are read from a "tty" (a simple command line interface program where key strokes are collected and interpreted), 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 ('...').

40. What is script? Where can you write the Python code? [Model Question]

Answer:
A script is a series of lines of Python code that will be executed all at once. Python code can be entered using a script.

41. Why a banner saying "RESTART" is always displayed in Python program execution? [Model Question]

Answer:
When you execute a program from the IDLE Editor, the interpreter gives a banner saying "RESTART", meaning that all the things you defined in any shell session so far are wiped clean and the program you are running starts afresh.

42. Python is an interpreted language. Justify. [Model Question]

Answer:
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 the hard disk to memory and starts running it.

Python, on the other hand, does not need the compilation and linking/loading steps. You just run the program directly from source code. Internally, Python converts the source program into an intermediate form called bytecodes and then translates this into the native language of your specific system and then runs it. all this actually makes Python much easier to use since you do not have to worry about compiling the program, making sure the proper libraries are linked and loaded, etc. This also makes your Python programs more portable since you can just copy the program to another computer and it just works!

43. Differentiate between interactive mode and script mode. [Model Question]

Answer:
In interactive mode, Python displays the results of expressions. In script mode, however, Python doesn't automatically display results. In order to see output from a Python script, we'll introduce the print statement. This statement takes a list of values and prints their string representation on the standard output file. The standard output is typically directed to the Terminal window.

44. What is a variable? [Model Question]

Answer:
A variable is an identifier that holds a value. In programming, we say that we assign a value to a variable. Technically speaking, a variable is a reference to a computer memory, where the value is stored.

45. Which of the following mathematical symbols are parts of the Python character set?

+, –, ×, ÷, =, ≠, <, or <

Answer: [Model Question]

Only, +, -, – and < mathematical symbols are part of the Python character set. In Python, the multiply operator is *, divide is /, not equal is != and less than or equal is ⇐.

46. What a variable can hold in Python? [Model Question]

Answer:
In Python language, a variable can hold a string, a number or various objects like a function or a class. Variables can be assigned different values over time.

47. State the basic rules to declare variables in Python. [Model Question]

Answer:
The rules are:

- Variables in Python can be created from alphanumeric characters and underscore (_) character.
- A variable cannot begin with a number.
- The variables are case sensitive.
- Variable names should not be reserved work or keyword.
- No special characters are used except underscore (_).
- Variables or identifiers can be of unlimited length.

48. How can we declare variables in Python? [Model Question]

Answer:
Python variables do not have to be explicitly declared to reserve memory space. The declaration happens automatically when you assign a value to a variable the equal sign (=) is used to assign values to variables.

49. What is a dynamic type? Give example. [Model Question]

Answer:
In Python, while the value that a variable points to has a type but the variable itself has no strict type in its definition. You can reuse the same variable to point to an object of a different type. This is called a dynamic type. For example,

```
>>>Val=10
>>>Val
10
>>>Val='ten'
>>>Val
'ten'
```

Here, variable Val is a dynamic data type.

50. Name the key attributes of a Python object. [Model Question]

Answer:
Each object in Python has three key attributes a type, a value and an id.

51. What is a keyword? [Model Question]

Answer:
A keyword is a reserved word in the Python programming language. Keywords are used to perform a specific task in a computer program.

52. How can you define the string literals in Python? [Model Question]

Answer:
String literals can be defined with any of the single quotes (‘), double quotes (“) or triple quotes (‘‘‘ or “““).

53. Explain the following command:
```
print "Hello World"
```
[Model Question]

Answer:
The print statement prints a string called “Hello World”. After the print command there is a set of parenthesis. Inside these parentheses, there exists what should be printed to the screen. If a print statement has quotes around text, the computer will print is out just as it is written.

54. When we write a statement like
```
a, b, c, d=1, 2, 3, 4
```
What does you call it? [Model Question]

Answer:
Multiple assignment.

55. Name five primitive data types in Python. [Model Question]

Answer:
Five primitive data types in Python are: Numbers, String, List, Tuple and Dictionary.

56. What does ‘immutable’ mean; which data type in python are immutable? [Model Question]

Answer:
An object whose state or value cannot be changed in place is said to be immutable type. So, a sequence that is immutable sequence is one that cannot change. For example, Strings and Tuples are immutable.

57. Differentiate between single quotes, double quotes and triple quotes. [Model Question]

Answer:
The differences are:

- If you quote with single quotes, you do not have to escape double quotes and vice versa.
- If you quote with triple quotes, you string can span multiple lines.

58. How can you create a complex literal in Python? [Model Question]

Answer:
Complex literals can be created by using the notation x+yj where x is the real component and y is the imaginary component. For example, >>>Ij*Ij which produces the result as: (–1+0j).

59. Name two sequence types and set types in Python. [Model Question]

Answer:
Two sequence types are: list and tuple.
Two set types are: dictionary (dict) and set.

60. How can you create an empty list in Python? [Model Question]

Answer:
To create an empty list, you can use empty square brackets or use the `list()` function with no arguments.

61. Is string a sequence type data type? [Model Question]

Answer:
Yes

62. How can you create an empty tuple in Python? [Model Question]

Answer:
To create an empty tuple, you can use and empty pair of parentheses, or with the `tuple()` built-in function.

63. What is the use of del statement? [Model Question]

Answer:
The `del` statement is used to delete the reference of an object. For example,

```
x=20
directions=('North', 'South', 'East', 'West')
del x, Directions
```

Here, the `del` statement deletes the reference of variable x and Direction tuple.

64. What will be the output of the following? [Model Question]

```
>>>sms='''Pleases,
"send me"
your Mail-id - and -
yours friend's address'''
>>>print sms
```

Answer:
The output is:
Please,
"send me"
your Mail-id - and -
yours friend's address

Long Answer Type Questions

1. a) Explain the basic data types available in Python with examples.
b) What is difference between list and tuples in Python? What are the key features of python? **[WBSCTE 2022]**

Answer:

a)

Data type	Description	Example
`str`	String	`str(), "Hello", 'Hello'`
`unicode`	Unicode, characters	`unicode(), u'hello', "world". encode('utf-8')`
`int`	Integer	`int(), 1, 55`
`float`	Decimal precision integers	`float(), 1.0, .032`
`bool`	Boolean Values	`bool(), True, False`
`list`	List of elements	`list(), [3, 'asd', True, 3]`
`dictionary`	Set of key: value pairs used to structure data	`dict(), {'element': 'Mn', 'Atomic Number': 25, 'Atomic Mass': 54.938}`
`set`	List of unique elements	`set(), [3, 4, 'hello']`
`tuple`	Organized list of elements	`tuple(), (2, 'Hello World!', 55.6, ['element 1'])`
`file`	A file object	`open('write_output.txt', 'w')`

b) 1st Part:

The main differences between lists and tuples are – Lists are enclosed in `brackets([])` and their elements and size can be changed, while tuples are enclosed in `parentheses(())` and cannot be updated. Tuples can be thought of as read-only lists.

2nd Part:

There are many features in Python, some of which are discussed below as follows:
The following are some of the features in Python that are discussed below:

1. Easy to Code: Python is a very high-level programming language, yet it is effortless to learn. Anyone can learn to code in Python in just a few hours or a few days. Mastering Python and all its advanced concepts, packages and modules might take some more time. However, learning the basic Python syntax is very easy, as compared to other popular languages like C, C++, and Java.

2. Easy to Read: Python code looks like simple English words. There is no use of semicolons or brackets, and the indentations define the code block. You can tell what the code is supposed to do simply by looking at it.

3. Free and Open-Source: Python is developed under an OSI-approved open source license. Hence, it is completely free to use, even for commercial purposes. It doesn't cost anything to download Python or to include it in your application. It can also be freely modified and re-distributed. Python can be downloaded from the official Python website.

4. Robust Standard Library: Python has an extensive standard library available for anyone to use. This means that programmers don't have to write their code for every single thing unlike other programming languages. There are libraries for image manipulation, databases, unit-testing, expressions and a lot of other functionalities. In addition to the standard library, there is also a growing collection of thousands of components, which are all available in the Python Package Index.

5. Interpreted: When a programming language is interpreted, it means that the source code is executed line by line, and not all at once. Programming languages such as C++ or Java are not interpreted, and hence need to be compiled first to run them. There is no need to compile Python because it is processed at runtime by the interpreter.

6. Portable: Python is portable in the sense that the same code can be used on different machines. Suppose you write a Python code on a Mac. If you want to run it on Windows or Linux later, you don't have to make any changes to it. As such, there is no need to write a program multiple times for several platforms.

7. Object-Oriented and Procedure-Oriented: A programming language is object-oriented if it focuses design around data and objects, rather than functions and logic. On the contrary, a programming language is procedure-oriented if it focuses more on functions (code that can be reused). One of the critical Python features is that it supports both object-oriented and procedure-oriented programming.

8. Extensible: A programming language is said to be extensible if it can be extended to other languages. Python code can also be written in other languages like C++, making it a highly extensible language.

9. Expressive: Python needs to use only a few lines of code to perform complex tasks. For example, to display Hello World, you simply need to type one line - print("Hello World"). Other languages like Java or C would take up multiple lines to execute this.

10. Support for GUI: One of the key aspects of any programming language is support for GUI or Graphical User Interface. A user can easily interact with the software using a GUI. Python offers various toolkits, such as Tkinter, wxPython and JPython, which allows for GUI's easy and fast development.

11. Dynamically Typed: Many programming languages need to declare the type of the variable before runtime. With Python, the type of the variable can be decided during runtime. This makes Python a dynamically typed language.
For example, if you have to assign an integer value 20 to a variable "x", you don't need to write int x = 20. You just have to write x = 15.

12. High-level Language: Python is a high-level programming language because programmers don't need to remember the system architecture, nor do they have to manage the memory. This makes it super programmer-friendly and is one of the key features of Python.

13. Simplify Complex Software Development: Python can be used to develop both desktop and web apps and complex scientific and numerical applications. Python's data analysis features help you create custom big data solutions without so much time and effort. You can also use the Python data visualization libraries and APIs to present data in a more appealing way. Several advanced software developers use Python to accomplish high-end AI and natural language processing tasks.

14. Other Advanced Programming Features: Python contains several advanced programming features such as generators (used to create iterators with a different approach that most other languages) and list comprehensions (used to create new lists from other iterables). Python also has automatic memory management eliminating the need to manually allocate and free memory in the code.

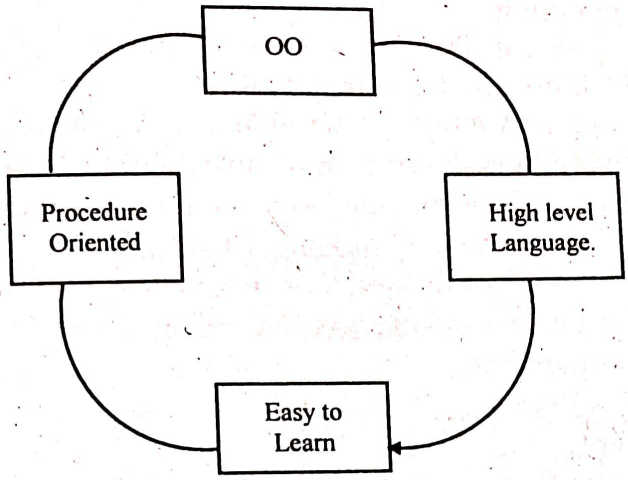
➲ **2. What is Python Programming language? Explain the features of Python. What are the applications using Python Programming Language?**

[Model Question]

Answer:

1st Part:

- Python was created by Guido Rossumin 1989 and is very easy to learn, read and write.
- Python Programming language is an interpreted, Object oriented, High Level Programming language with dynamic semantics.



2nd Part:

Features of Python Programming Language:

1. Easy to learn. It is loosely typed language.
2. Free and Open source.
3. It contains runtime tracking of data.
4. It is interpreted and dynamic type language.

3rd Part:

Applications of Python Programming Language:

Python is known for its general-purpose nature that makes it applicable in almost every domain of software development. Python makes its presence in every emerging field. It is the fastest-growing programming language and can develop any application. The applications are described in below:

1) Web Applications:

We can use Python to develop web applications. It provides libraries to handle internet protocols such as HTML and XML, JSON, Email processing, request, beautifulSoup, Feedparser, etc. One of Python web-framework named Django is used on **Instagram.** Python provides many useful frameworks, and these are given below:

- Django and Pyramid framework(Use for heavy applications)
- Flask and Bottle (Micro-framework)
- Plone and Django CMS (Advance Content management)

2) Desktop GUI Applications:

The GUI stands for the Graphical User Interface, which provides a smooth interaction to any application. Python provides a **Tk GUI library** to develop a user interface. Some popular GUI libraries are given below.

- Tkinter or Tk
- wxWidgetM
- Kivy (used for writing multitouch applications)
- PyQt or Pyside

3) Console-based Application:
Console-based applications run from the command-line or shell. These applications are computer program which are used commands to execute. This kind of application was more popular in the old generation of computers. Python can develop this kind of application very effectively. It is famous for having REPL, which means **the Read-Eval-Print Loop** that makes it the most suitable language for the command-line applications. Python provides many free library or module which helps to build the command-line apps. The necessary **IO** libraries are used to read and write. It helps to parse argument and create console help text out-of-the-box. There are also advance libraries that can develop independent console apps.

4) Software Development:
Python is useful for the software development process. It works as a support language and can be used to build control and management, testing, etc.

- **SCons** is used to build control.
- **Buildbot** and **Apache** Gumps are used for automated continuous compilation and testing.
- **Round** or **Trac** for bug tracking and project management.

➲ 3. Describe the process of downloading and installing python. **[Model Question]**

Answer:
Python produces different versions for different operating systems. Python is available for all major operating systems: Windows, Linux/Unix, OS/2, Mac, Amiga, etc. it can be confusing trying to find the correct version of Python. As per the Python development project, you may find different versions at: **http://www.python.org/download/** According to current update, in this text, we use **Python 2.7.1** for Windows version. Be careful to choose the version for your operating system and hardware.

Downloading Python:
Python can be downloaded and installed for free from:
http://www.python.org/download/
For additional help on Python:

- The standard Python documentation set contains a tutorial, a language reference, the standard library reference and documents on extending Python in C/C++. You can find it at: http://www.python.org/doc/
- Other Python tutorials:
  - Beginner's Guide to Python at: http://wiki.python.org/moin/BeginnersGuide
- Other Python resources:
  - Python documentation at: http://www.python.org/doc/
  - The Python home Web site at: http://www.python.org/
  - The whole Python FAQ at: https://docs.python.org/2/faq/

To download Python:

- Start a browser like Internet Explorer or Mozilla Firefox.

- Type http://www.python.org/download/ at the address bar. Notice that the download page for Python appears.

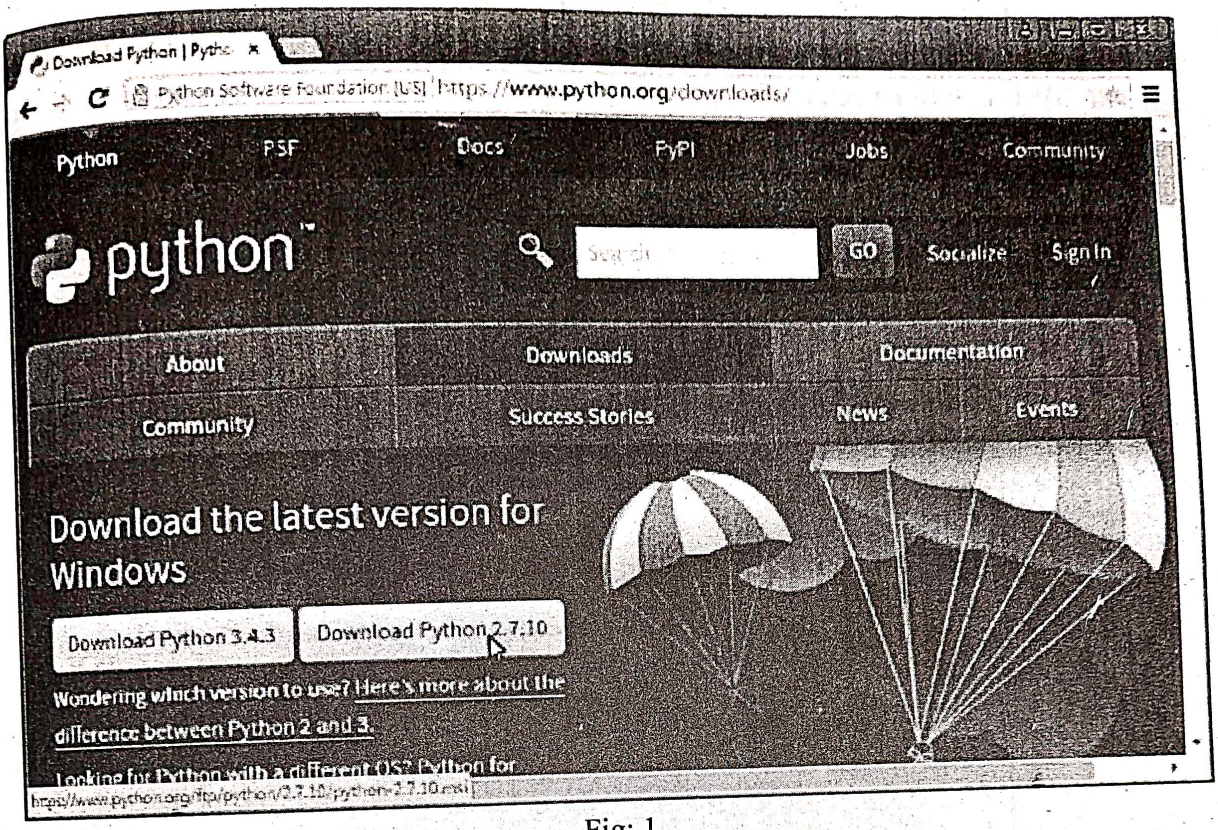


Fig: 1

- Click on the latest version of download link as shown in Fig. 1.
- Click the download link (called Python 2.7.10 Windows x86 MSI installer released on May 23, 2015) and see the dialog box to save the windows installer package as shown in Fig. 2.

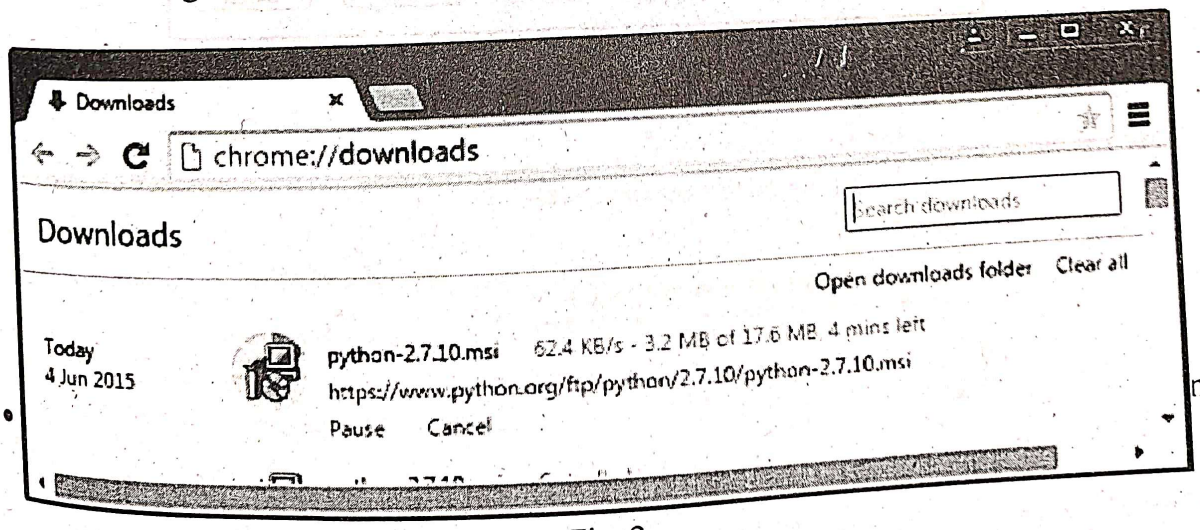


Fig: 2

Installing Python:

As we have already downloaded **Python 2.7.10 Windows x86 MSI installer**, now it is easy to install it in our own computer. To install Python:

- Open the Windows Download folder or the location where **Python 2.7.10 Windows x86 MSI installer** package is present.
- Double click on it.
- The Python installer will open as shown in Fig. 3.
- Select for whom you want to have access to Python. Just choose one and click Next.
- Next, choose a directory where Python should be installed. By default Python is installed in C:\ drive (see Fig. 4). You should really just leave it as it is and click Next.

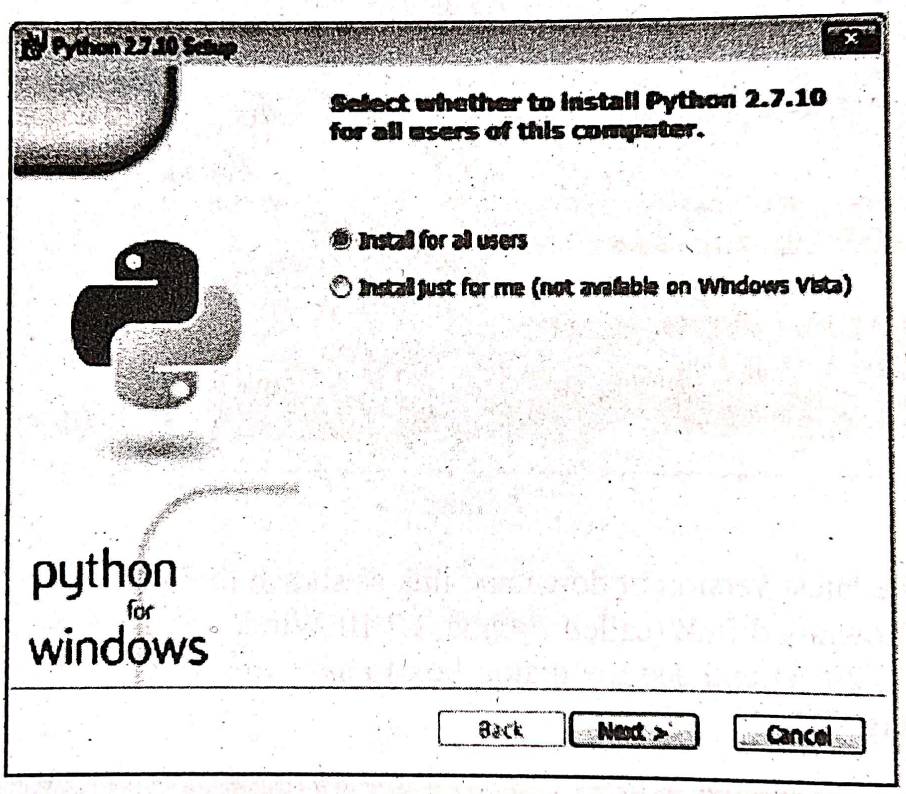


Fig: 3

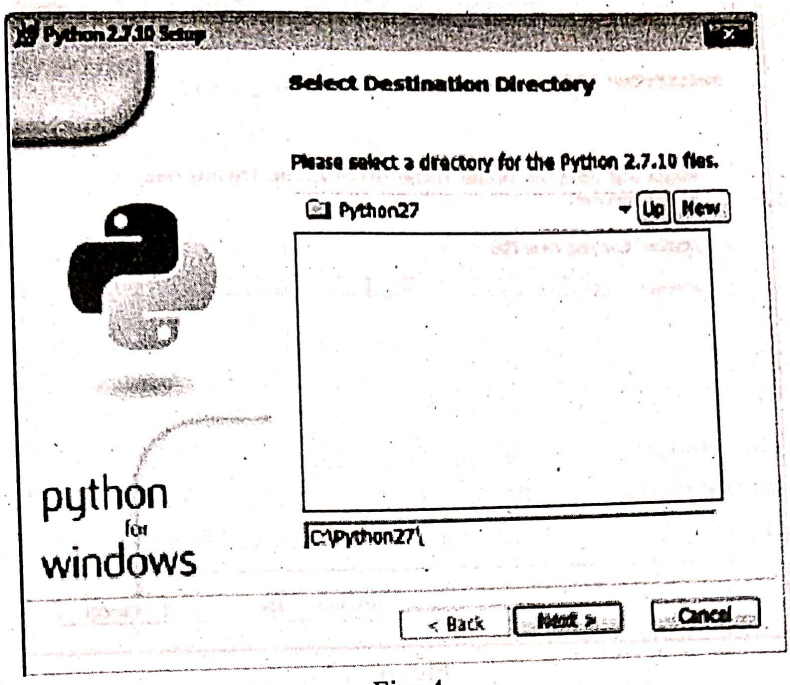


Fig: 4

- The next screen shows all default installation options (see Fig. 5). If you click Disk Usage or Advanced button, you may find different settings. Accept the default installation options on this screen and click Next.

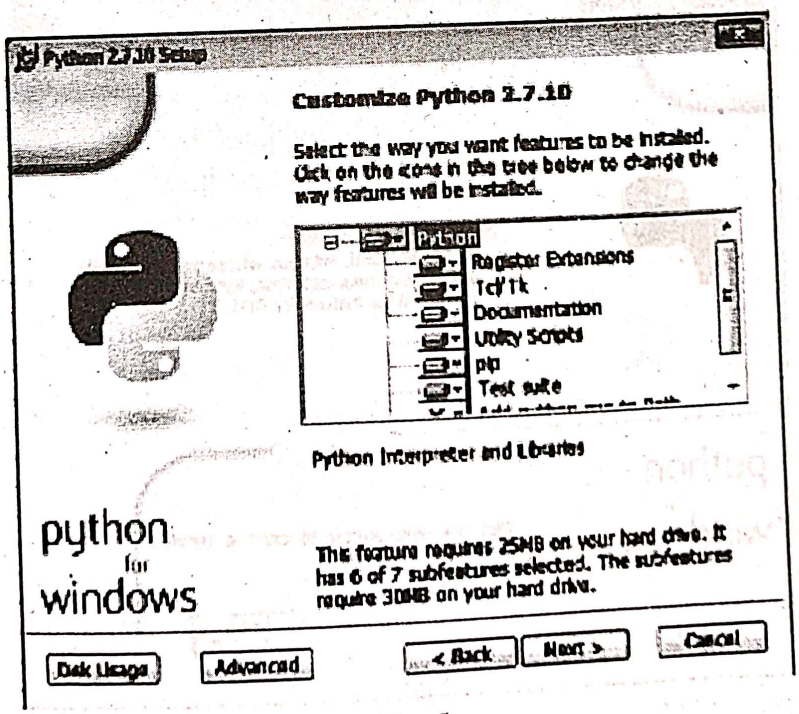


Fig: 5

- Python will begin to get installed on your system (see Fig. 6). In just a few minutes you will have the world's greatest language on your own computer!

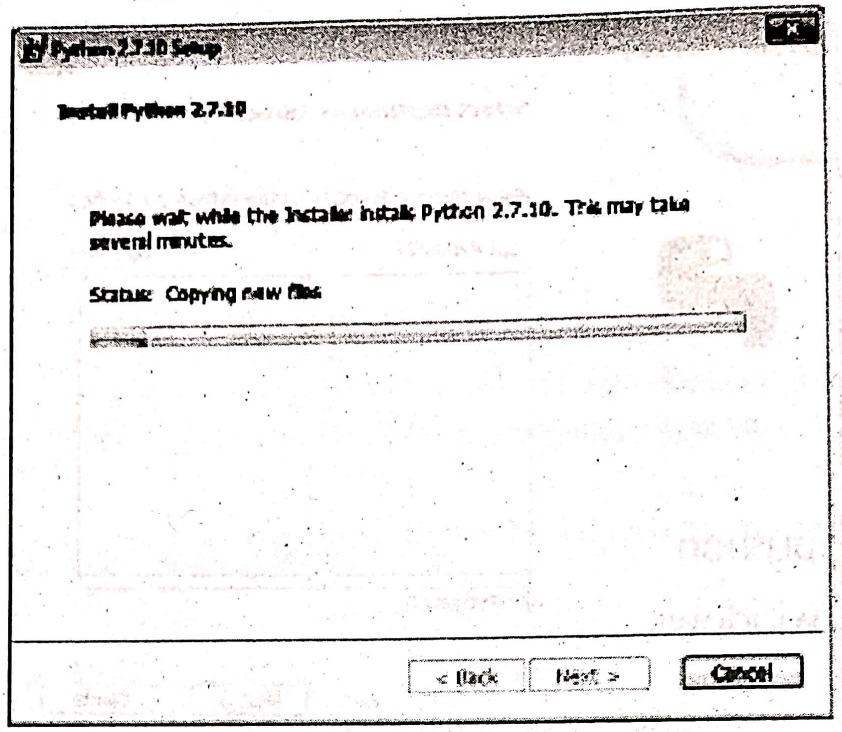


Fig: 6

- When the installer will complete, you will see the following screen. Click Finish (see Fig. 7).

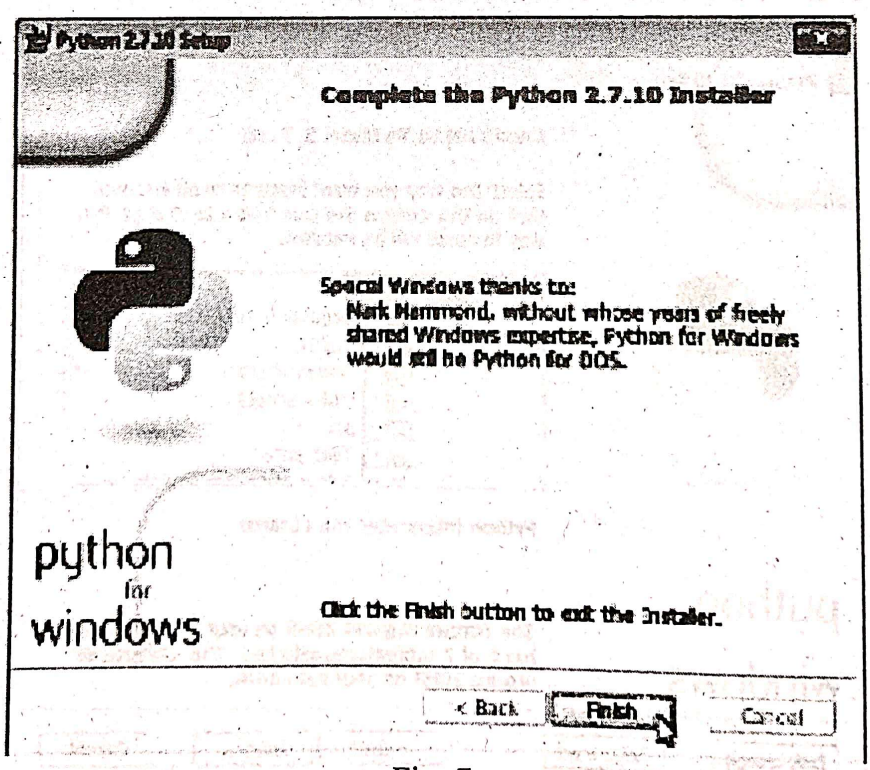


Fig: 7

4. Explain Numeric data type in python. [Model Question]

Answer:

Number data types store numeric values. This is also called as a numeric literal. They are immutable data types, which mean that changing the value of a number data type results in a newly allocated object. Python provides three basic types of numbers: plan integers, floating point numbers and complex numbers. Number objects are created when you assign a value to them. To assign a value to an integer variable, the syntax is:

```
<numeric literal name>=<numeric literal value>
```

For example,

```
Num1=1
Num2=10
```

Notice that the two variables **Num1** and **Num2** assign two integer values 1 and 10 respectively. You can also delete the reference to a number object. Python also provides a mechanism for removing variables using the **del** statement. The syntax of the **del** statement is:

```
del var1[, var2[, var3[…., varN]]]
```

Or

```
del(object, …)
```

You can delete a single object or multiple objects by using the **del** statement. For example,

```
del var
del var_a, var_b
```

Any numeric value can be assigned to a variable at any time, overwriting the previous value. Python supports three basic numeric data types:

- **int.** Whole numbers, i.e., signed integer
- **float.** Approximated real numbers
- **complex.** Imaginary numbers

The following Table shows some examples of numeric data types.

Table: Numeric data types

int	float	complex
10	0.0	3.14j
100	15.20	45.j
–786	–21.9	9.322e–36j
080	32.3+e18	.876j
–0490	–90.	–.6545+0j
–0×260	–32.54e100	3w+26j
0×69	70.2–E12	4.53e–7j

Integer Type:

A Python object of type **int** an integer, that is, a signed whole number (i.e., signed integer, 32 bits long). The range is a least –2,147,483,648 to 2,147,483,647 (approximately ±2 billion). To write an **int** constant, you may use several different formats.

- A zero is written as just.
- To write an integer in decimal (base 10), the first digit must not be zero. Examples: 17, 100000000000000.
- To write an integer in octal (base 8), precede it with "0o" and use the digits 0 to 7.0123 is octal and equal to 83 decimal and uses the digits 0 through 9, plus a, A, b, B, c, C, d, D, e, E, f and F. 0x2BC8 is hexadecimal and equal to 11208. Examples: 0o177,0o37.
  For example,

```
>>>0o37
37
>>>0o177
127
```

- To write an integer in hexadecimal (base 16), precede it with "0x" or "0X". Examples: x7f,X1000.
- To write an integer in binary (base 12), precede it with "0b" or "0B". Examples: b1001,B111110011101. Hex, octal and long notations can be combined 0x234C678D098BAL, for example, 6209769885267781. To produce a negative number, use the unary "-" operator before the number. Note that this is an operator and not part of the constant.

```
>>>100--5
105
>>>100-(-5)
105
```

Long Type:

When we want to store a number larger than 2,147,483,647, Python can do it, using the long data type. Python can manipulate long data types in the form of `int` type. For example,

```
>>>print 2**32
4294967296
>>>type(2**32)
<type 'long'>
```

Float Type:

A format for representing numbers with fractional parts is called floating-point numbers. Python cannot represent very large or very small numbers and the precision is limited to only about 14 digits. Floating-point numbers are stored with three parts:

- a sign: + or –
- a mantissa e.g., 6.02
- an exponent e.g., 23

To write a constant, use at least one digit, plus either a decimal point or an exponent or both.

- The decimal point present, must be preceded or followed by one or more digits. Examples:
  3.14

0.0
1.
.1
pi = 3.14
print pi

- If the exponent present, consists of either "e" or "E", optionally followed by a "+" or "–" sign, followed at least one digit, the resulting value is equal to the part before the exponent times ten to the power of the exponent, that is, a scientific notation. For example, carbon in 12 grams of carbon[12] and is written as 6.0221418×10^{23}. In Python that would be "6.0221418e23" which can be written as,

```
>>>c=6.0221418e23
>>>print "Carbon in 12 gram: ", c
Carbon in 12 gram: 6.0221418e+23
```

Complex Type:

Mathematically, a complex number is a number of the form A+Bi where i is the imaginary number, equal to the square root of –1. Complex numbers are also supported, imaginary numbers are written with a suffix of 'j' or 'J'. Complex numbers with a nonzero real component are written as '(real+imagj)'

For example,

3.14j as in imaginary number = $3.14\sqrt{-1}$

A complex number is created by adding a real and an imaginary number: $2+14j$. Note that Python always prints these in ()'s, for example $(2+14j)$.

Python displays complex numbers in parentheses when they have a nonzero real part. For example,

```
>>>print(2+3j)*(4+5j)
(-7+22j)
>>>5j
5j
>>>1+2.56j
(1+2.56j)
>>>(1+2.56j)*(-1-3.44j)
(7.8064-6j)
```

Unlike Python's other numeric types, complex numbers are a composite quantity made of two parts: **the real part** and **the imaginary part**, both of which are represented internally as values. You can retrieve the two components using attribute references. For a complex number C:

- C real is the real part.
- C.imag is the imaginary part.

```
>>>a=(1+2.56j)*(-1-3.44j)
>>>a
(7.8064-6j)
>>>a.real
```

```
7.8064
>>>a.imag
-6.0
```

➲ **5. How to get the current time?** [Model Question]

Answer:

The `localtime()` functions of the time module are used to get the current time tuple. Consider the following example.

Example:

```
1. import time;
2. #returns a time tuple
3. print(time.localtime(time.time()))
```

Output:

```
time.struct_time(tm_year=2019, tm_mon=12, tm_mday=18, ti_hour=15,
tm_min=1, tm_sec=32, tm_wday=1, tm_yday=352, tm_isdst=0)
```

➲ **6. Write short notes on the following:**

a) String type [Model Question]
b) List type [Model Question]
c) Tuple type [Model Question]
d) Dictionary Mapping type [Model Question]
e) Sets type [Model Question]
f) Arithmatic operator [Model Question]
g) Relational operator [Model Question]
h) Membership operator [Model Question]
i) Assignment operator [Model Question]
j) Logical operator [Model Question]
k) Identity operator [Model Question]

Answer:

a) String type comes under sequence type or collection type. a string literal or string in Python can be created using single quotes, double quotes and triple quotes. When we use triple quotes, strings can span several lines without using the escape character.

To store a string, the syntax is:

<name of string>='<value of string>'

For example,

```
>>>Bday="Happy birthday to you."
>>>Msg='Please send us your Email-ID.'
>>>GameStatus="""No win situation"""
>>>postcode='''011-2705'''
>>>print Bday, Msg, GameStatus, 'Post Code:', postcode
```

Happy Birthday to you. Please send us your Email-Id. No win situation Post Code: 011-2705

From above declarations, we store string value using single, double and triple quotes. The fact of the matter is that Python allows all of them. That is, when you begin a value with

double quotes, you must end it with double quotes. But the following declaration is wrong:

```
Bday='Happy birthday to you"
Msg="[please send us your main id'
GameStatus="""no win situation'"
Postcode="'91101-2509"'"
```

Subsets of strings can be taken using the slice operator (`[]` and `[:]`) with indexes starting at 0 in the beginning of the string and working their way from –1 at the end. The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example,

```
str1="Python in CBSE Schools"     # Assigns a string into variable str.
print str1                        # Prints the string str, i.e., Python in CBSE
                                  School Schools
print str1[0]                     # Prints first character of the string, i.e., P
print str[2:5]                    # Prints characters starting from 3rd to 5th, i.e.,
                                  tho
print str[2:]                     # Prints string starting from 3rd character, i.e.,
                                  thon in CBSE Schools
print str*2                       # Prints string two times, i.e., in CBSE
                                  SchoolsPython in CBSE Schools
print str+"-2014 onwards"         # Prints concatenated string, i.e.,
                                  # Python in CBSE Schools - 2014 onwards.
```

Let us assign multilane strings:

```
>>>String="'This is a multi-line string.
This is the second line.'"
>>print String
This is a multi-line string.
This is the second line.
```

From the above five lines, in first two lines we assign a triple-quoted string to the name **`String`** and then display its value using the **`print`** statement.

b) List is a sequence data type A list is, as the name suggests, a series of values. In Python, these values are assigned by placing them within square braces and separating them by commas. To some extent, lists are similar to arrays in C, C++ and Java. One difference between them is that all the items belonging to a list can be of dafferent data type. to declare a list the syntax is:

`<name of list>=[<value>, <value>, <value>]`

For example,

```
Address=['A-10/130', 'Sangam Vihar', 'New Friends colony', 'New
Delhi - 110004'
print Address
```

will print the complete list as,

['A-10/130', 'Sangam Vihar', 'New Friends colony', 'New Delhi - 110004']

Let us take another example to assign first 10 natural numbers and print the numbers in a list called Num as

```
Num=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print Num
```

which will print the complete list as,

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

From the previous two variables **Address** and **Num**, we notice that when we store string values (**Address**) inside a list, it should be placed in either single or double quotes with comma separator and for number (**Num**) only by simple comma separator.

The values stored in a list can be accessed using the slice operator ([] and [:]) with indexes starting a 0 in the beginning of the list and working their way to end–1. The plus (+) sign is the list concatenation operator and the asterisk (*) is the repetition operator.

For example, suppose, we have two lists as given below:

```
ListDays=['Days in year - 2014: ', 31, 28, 31, 30, 31, 30, 31,
31, 30, 31, 30, 31]
ListMonths=['January', 'February', 'March', 'April', 'May',
'June', 'July', 'August', 'September', 'October', 'November',
'December']
print ListDays          # Prints complete list
print ListDays[0]       # Prints first element of the list
print ListDays[1:3]     # Prints elements starting from 2nd till 3rd
print ListDays[1:]      # Prints elements starting from 2nd element
print ListMonths[0]*2   # Prints first element two times
print 'Months: ', ListMonths, 'with', ListDays # Prints
concatenated lists
```

This will produce following result respectively:

['Days in year - 2014: ', 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

Days in year - 2014:

[31, 28]

[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

Months: ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December'] with ['Days in year – 2014: ', 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

c) In Python, a tuple may be defined as a finite, static list of literals (numeric or string). A tuple is very similar to a list in that it contains a sequence of heterogeneous items. Unlike lists, however, tuples are enclosed within parentheses [( )]. For example, if you see a built-in type function like **max()** and **min()**, then it stores values in form of tuple. That is,

```
>>>max(1,2,3)
3
>>>min(19,9,99)
9
```

Both the build-in functions store values in form of tuples. The main differences between lists and tuples are:

- Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated.

- Tuples can be thought of as read-only lists. Like list, to initialize a tuple, one can enclose the values in parentheses and separates them by commas

For example,

```
Directions=('North', 'South', 'East', 'West')
Weekdays=('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday')
print Directions              # Prints complete list.
print Directions[0]           # Prints first element of the list.
print Directions[1:3]         # Prints elements starting from 2nd till 3rd.
print WeekDays[2:]            # Prints elements starting from 3rd element.
print Directions*2            # Prints list two times.
print WeekDays+Directions     # Prints concatenated lists.
```

This will produce following results respectively

```
('North', 'South', 'East', 'West')
North
('South', 'East')
('Tuesday', 'Wednesday', Thursday', 'Friday', 'Saturday')
('North', 'South', 'East', 'West', 'North' 'South', 'East',
'West')
('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday', 'North', 'South', 'East', 'West')
```

As per the tuples rule, the following is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with the lists:

```
Directions=('North', 'South', 'East', 'West')  # A tuple.
Marks=[80, 76, 87, 90, 66]                     # A list.
Directions[2]='North-West'                     # Invalid syntax with tuple.
Marks[2]=88                                    # Valid syntax with list.
```

From the above four command lines, from the last two, the second last line:

```
Directions[2]='North-West'
```

The above statement produces an error as: "Invalid syntax with tuple". Because, 'tuple' object does not support item assignment. But the last line does not produce any error because it is a list which can be updated.

d) All of the compound data types we have studied in detail so far strings, lists and tuples are sequence types, which use integers as indices to access the multiple values they contain. **Dictionaries** are a different kind of compound type. They are Python's **mapping type**. They are map **keys** which can be of any immutable type to values which can be of any type, just like the values of a list or tuple.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([ ]).

Initializing a dictionary, one offsets the keys and values in curly braces. Each key-value pair is separated from others by a comma. for each pair, the key and value are separated by a colon. The key of each member is offset in quotes. For example, a sample dictionary is as follows:

```
Book_Profile={"Author": "Reeta Sahoo & Gagan Sahoo",
```

```
"Title": "MyPython Programming for Class-XI",
"EYear": "2014",
"Price": 250
}
```

To assign or access directory values:

```
print Book_Profile                  # Prints complete dictionary.
print Book_Profile.keys()           # Prints all the keys.
print Book_Profile.values()         # Prints all the values.
```

This will produce following result:

```
{'Price': 250, 'Title': 'MyPython Programming for Class-XI',
'EYear': '2014', 'Author': 'Reeta Sahoo & Gagan Sahoo'}
['Price', 'Title', 'EYear', 'Author']
[250, 'MyPython Programming for Class-XI', '2014', 'Reeta Sahoo &
Gagan Sahoo']
```

You can assign the directory value to variable as:

```
Ath=Book_Profile["Author"]
print Ath
```

which will print: Reeta Sahoo & Gagan Sahoo

To insert or modify a member, one simply assigns the value:

```
Book_Profile["Publisher"]='New Saraswati House India (p) Ltd.'
# Inserts new member into directory.
Book_Profile["Author"]='Sahoo & Sahoo' # Updates existing member.
print Book_Profile                  # Prints complete dictionary.
```

This will produce the following result:

```
{'Publisher': 'New Saraswati House India (P) Ltd.', 'Price': 250,
'Title': 'MyPython Programming for Class-XI', 'EYear': '2014',
'Author': 'Sahoo & Sahoo'}
```

A directory can also be declared as follows:

```
>>>Subject={}                       # Declares an empty
dictionary.
>>>Subject['One']="This is English"  # Inserts a new member
as key One.
>>>Subject[2]= "This is Physics"     # Inserts another new
member as key 2.
>>>print Subject                     # Prints complete
dictionary.
{2: 'This is Physics', 'One': 'This is English'}
```

When we create a dictionary type, we may store duplicate entries in the dictionary. But when we operate the dictionary, it automatically removes the duplicate entries. For example,

```
>>>fruits={'apple', 'orange', 'apple', 'pear', 'orange',
'banana'}
>>>print fruits          # show that duplicates have been removed.
set(['orange', 'pear', 'banana', 'apple'])
```

```
>>>'orange' in fruits   # fast membership testing.
True
>>>'guava' in fruits
False
```

e) A set is an unordered collection with no duplicate elements i.e., set containing unique members. Basic uses include membership testing and eliminating duplicate entries. It is a lot faster than checking if an object is in a list. Set objects also support mathematical operations like union, intersection, difference and symmetric difference. Curly braces or the set() function can be used to create sets. To create a set:

```
>>>a=set('abracadabra')
>>>b=set('alacazam')
                                # prints unique letters in set a.
>>>a
set(['a', 'r', 'b', 'c', 'd'])
                                # letters in a but not in b.
>>>a-b
set(['r', 'b', 'd'])
                                # letters in either a or b.
>>>a|b
set(['a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'])
                                # letters in both a and b.
>>>a&b
set(['a', 'c'])
                                # letters in a or b but not both.
>>>a^b
set(['b', 'd', 'm'. 'l', 'r', 'z'])
>>>a
set(['a', 'r', 'b', 'c', 'd'])
>>>a.add('p')                   # Adds p into set.
>>>a
set(['a', 'c', 'b', 'd', 'p', 'r'])
```

f) Arithmetic Operator:

An arithmetic operator in Python that performs mathematical operations on data and returns manipulated value.

Operator symbol	Meaning	Brief description
+	Addition	Addition of given operands
–	Subtraction	Subtraction of given operands
*	Multiplication	Multiplication of given operands
/	Division	Division of given operands
%	Modulus	It will return the remainder between two operands
**	Exponentiation	It represents X to the power of Y
//	Floor division	Floor division is the same as division, except that it returns the largest possible integer

For example,

```
print(2+2)  # Output: 4
print(2-2)  # Output: 0
print(2*2)  # Output: 4
print(2/2)  # Output: 1.0
```

```
print(5%2) # Output 1 because it return remainder
print(2**3) # Output 8 because it represent 2 to the power of 8
print(22//10) # Output 2 because it's a floor division
```

g) Relational Operator

An operator in Python that performs operations on data and returns a result as **True** or **False**. It is also known as a comparison operator.

Operator	Description
>	It checks whether the left operand is greater than the right then returns true else false
<	It checks whether the left operand is lesser than the right then returns true else false
>=	It checks whether the left operand is greater than or equal to the right then return true else false
<=	It checks whether the left operand is less than or equal to the right then return true else false
==	It checks whether left and right operand are equal then return true else false
!=	It checks whether left and right operand are not equal then return true else false

For example:

```
print(2>2) # Output: False
print(2<2) # Output: False
print(2>=2) # Output: True
print(2<=2) # Output: True
print(2==2) # Output: True
print(2!=2) # Output: False
```

h) Membership Operator

As the name suggests, it checks whether the given value is part or member of a collection. Say in our day-to-day life, we say that I am a member of this society, club, community, etc. The same concept has been derived in Python to check.

Operator symbol	Description
In	It checks whether the left-side operand is a member of the right-side operand
Not in	It checks whether the left-side operand is a member of the right-side operand

Understand better with an example

```
x=10
y=200
varlist=[10,20,30,40,50]
print(x in varlist)
# Output: True
# Since x(10) is the member of a list hence it return True
print(y not in varlist)
```

```
# Output: True
# Since y(200) is not the member of a list hence it return True
```

i) Assignment Operator

As the name suggests, this operator assigns a value to a variable. Like the way we are using in mathematics and assigning some value to x variable say x=10

In Python, there are many Assignment operators

Operator	Description	Statement
=	It assigns a value from right side operand to left side operand	Z=a+b
+=	It first adds the value of the right-side operand and left side operand then assigns the result to the left side operand	x+=y is like x=x+y
-=	It first subtracts the value of the right-side operand and left-side operand then assign the result to the left-side operand	x-=y like x=x-y
*=	It first multiplies the value of the right-side operand and left-side operand then assign the result to the left side operand	x*=y is like x=x*y
/=	It first divides the value of the right-side operand and left side operand then assign the result to the left-side operand	x/=y is like x=x/y
%=	It first modulus the value of right-side operand and left-side operand then assign the result to the left-side operand	x%=y is like x=x%y
**=	It first exponent the value of right-side operand and left-side operand then assign the result to the left-side operand	x**=y is like x=x**y
//=	It first, floor division the value of right-side operand and left-side operand then assign the result to the left-side operand	x//=y is like x=x//y

Let's have a look into some example

```
x=5
y=2
x+=y # This is similar to x=x+y
print(x)
# Output: 7
```

j) Logical Operator

Logical operators are used with a conditional statement to give results in **True** or **False** to do any logical decision.

It is the same as the mathematics when we use **AND, OR, NOT** logical operators.

Operator	Description
and	It returns true if both sides of the operand are True
or	It returns true if either side of the operand are True
not	It reverses the result, if the operand is True it will return False and vice versa

Left expression	Logical operator	Right expression	Result
True	And	True	True
True	And	False	False
False	And	False	False
False	And	True	False
True	Or	True	True
True	Or	False	True
False	Or	True	True
False	Or	False	False
True	Not	NA	False
False	Not	NA	True

Key-points to remember

- If the first expression is false while using **and** operator, then further expressions will not be executed by Python.
- If the first expression is True while using **or** operator, then the further expressions will not be executed by Python.

Let's have a look at few examples

```
x=5
y=2
if 5>2 and 2>5: # first expression is true but second expression
is false
    print("True")
else:
    print("False")
# Since and logical operator have been used it return False
if 5>2 or 2>5: # first expression is true but second expression
is false
    print ("True")
# Since or logical operator have been used it return True
else:
    print("False")
if not 5>2: #This expression return True
    print("True")
else:
    print("False")
# Since not logical operator have been used it return False
```

Below is how Python executes the expression based on the logical operator and their expression result.

```
# First expression is False hence it won't execute further
if 2>5 and 5>2:
```

```
    print("True")
else:
    print("False")
# Output: False
#First expression is True.hence it won't execute further
if 5>2 or 2>5:
    print("True")
else:
    print("False")
# Output: True
```

Example of how to use not operator

```
if not 2>5 and 2>5:
    print("Inside if")
else:
    print("Inside else")
```

Output: Inside else

Explanation:

First, it executes not 2>5, it should be False but since **not** operator is being used it will return True. Since the first expression returns True hence it will check the second expression as well because of **and** operator.

Second expression, 2>5 return False

Finally, it becomes True **and** False hence logical operator returns False.

k) Identity Operator:

This operator compares two object's memory locations, whether they are the same.

Operator	Description
is	It returns **True** if the variable of both sides is referring to the same memory location otherwise **False**
is not	It returns **True** if the variable of both sides is referring to a different memory location otherwise **False**

Let's look at an example,

```
x=5
y=5
z=10
print(x is y) # x and y are pointing to same memory location
# Output: True
print(x is z) # x and z are not pointing to same memory location
# Output: False
print(x is not z)
# Output: True
```

Another example with the string data type,

```
x="Ajit"
y="Ajit"
print(x is y)
# Output: True
Print(x is not y)
s# Output: False
```

Unit 2: Control Structures

Unit at a glance:

2.1 Decision making and iteration keyword

Python contains multiple keywords used for loopoing, decision-making and controlling of execution statement order. Here's a list oof those keywords, along with a simple example of each one.

break: The `break` statement stops execution of a loop in Python. For example, you might want to stop a given loop when a condition is met. The following example will break out of a loop when the index value is odd.

```
index=0
while index<100:
    if index % 2==1:
        break
    print index
    index=index+1
```

This block of code will print out only the number 0, because as soon as the value of 1 is reached, it will break out of the loop as per the `break` statement.

continue: The `continue` statement allows a loop to pick up again at the top of the iteration, rather than executing any statements following the continue statement in the loop. For example, you might want to skip certain values in a loop. The following example will skip all odd values in a loop.

```
index=0
while index<10:
    index=index+1
    if index % 2==1:
        contunue
    print index
```

This block of code will print out every even number between 1 and 10. The `continue` statement skips all of the odd numbers.

elif: The `elif` statement is Python shorthand for "else if". An `elif` statement should follow an `if` statement and the code within the `elif` block will only be executed if the condition is Ture. This does not mean that every `if` statemnt must have a corresponding `elif`. Note that an elif statement must be at the same level as the `if` statement it follows. Finally, once any condition is evaluated to True, the entire block `if's` and `elif's` is exited. The following example shows how you might test for three different conditions.

```
x=input("Enter a value: ")
```

```
if x==1:
   print "You entered one"
elif x==2:
   print "You entered two"
elif x==3:
   print "You entered three"
```

When the user enters a value, the series of statements is interpreted. If the user entered, for example, a 2 at the prompt, the first `elif` would be evaluated to True and the block would exit.

else: The `else` statement corresponds to an `if` statement somewhere in an application. If the preceding `if` statemetn (or `elif` statement) is not true, the `else` statement code will be executed. The following example shows testing a single value and printing out some information, depending on whether the value is odd or even. The `else` statement can also be used with a `while` or `try` statement, as you will see later.

```
x=input("Enter a value: ")
if x % 2==0:
   print "Even"
elif:
   print "Odd"
```

for: The `for` statement allows you to iterate over a sequence of values. the following example prints out the elements of a set.

```
myList=[1, "Hello world", 2.5]
for i in myList:
   print i
```

if: The `if` statement allows you to evaluate an expression and take action, depending on whether the statement is True or False. The following example simply prints out a string if a variable is the proper value.

```
x=1
if x==1:
     print "Yes!", x
```

while: The `while` statement implements a looping mechanism that will continue to execute until a given condition is no longer true. The following example shows a loop that counts up to (but not including) 10.

```
index=0
while index<10:
      print index
      index=index+1
```

Short Answer Type Questions

A. Choose the correct answer from the given alternatives in each of the following:

1. What is the output of the following? **[WBSCTE 2022]**

```
x = ['ab', 'cd']
for i in x:
x.append(i.upper())
print(x)
```

(a) ['AB', 'CD'] (b) ['ab', 'cd', 'AB', 'CD']
(c) ['ab', 'cd'] (d) none of the mentioned

Answer: (d)

2. Which statement is used to terminate the execution of the nearest enclosing loop in which it appears? **[WBSCTE 2022]**

(a) Pass (b) break (c) continue (d) jump

Answer: (b)

3. What will be the output of Python code snippet x<<2 if x=1? **[WBSCTE 2022]**

(a) 4 (b) 2 (c) 1 (d) 8

Answer: (a)

4. Predict output for the snippet **[WBSCTE 2022]**

```
for j in range(1,5):
print(j)
```

(a) 1 2 3 4 (b) indentation error (c) j j j j (d) j j j j

Answer: (b)

5. What will be the output of Python code snippet x<<2 if x=1? **[WBSCTE 2022]**

(a) 4 (b) 2 (c) 1 (d) 8

Answer: (a)

6. What will be the output of the following Python expression if x = 56.236?

print("%.2f"%x) **[Model Question]**

(a) 56.236 (b) 56.23 (c) 56.000 (d) 56.24

Answer: (d)

7. Which module in the python standard library parses option received from the command line? [Model Question]

(a) getarg (b) getopt (c) main (d) os

Answer: (b)

8. What will be the output of the following Python code? [Model Question]

```
print("abc. DEF" .capitalize())
```

(a) Abc.def (b) abc.def (c) Abc.Def (d) ABC.DEF

Answer: (a)

9. What will be the output of the following Python code? [Model Question]

```
x=['ab', 'cd']
for i in x:
    i.upper()
print(x)
```

(a) ['ab', 'cd'] (b) ['AB', 'CD'] (c) [None, None] (d) none of these

Answer: (b)

10. What will be the output of the following Python code? [Model Question]

```
i=1
while True:
    if i%3==0:
     break
    print(i)
    i+=1
```

(a) 1 2 (b) 1 2 3 (c) error (d) none of these

Answer: (c)

11. What will be the output of the following Python code? [Model Question]

```
i=5
while True:
    if i%0011==0:
     break
    print(i)
    i+=1
```

(a) 5 6 7 8 9 10 (b) 5 6 7 8 (c) 5 6 (d) error

Answer: (b)

12. What will be the output of the following Python code? [Model Question]

```
i=1
while True:
    if i%2==0:
     break
    print(i)
    i+=2
```

(a) 1 (b) 1 2 (c) 1 2 4 5 6 ... (d) 1 3 5 7 9 11...

Answer: (d)

13. What will be the output of the following Python code? [Model Question]

```
True = False
while True:
    print(True)
    break
```

(a) True (b) False (c) None (d) none of these

Answer: (d)

B. Fill in the blanks in the following statements:

14. The ***decision making*** statement is a combination of an else statement and an if statement. [WBSCTE 2022]

15. ***Indentation*** is used to define a block of code in Python language. [WBSCTE 2022]

C. Answer the following questions:

16. What are python iterators? [WBSCTE 2022]

Answer:

Iterator in Python is an object that is used to iterate over iterable objects like lists, tuples, dicts, and sets.

17. What are Scope in Python? [WBSCTE 2022]

Answer:

Scope verifies which variable can be 'Seen'. The scope defines the set of rules which tell us how and where a variable can be searched.

18. Differentiate between continue and break statement. [WBSCTE 2022]

Answer:

Basis for comparison	Break	Continue
Use	It is used for the termination of all the remaining iterations of the loop.	It is used for the termination of the only current iteration of the loop.
Control after using break/continue statement	The line which is just after the loop will gain control of the program.	The control will pass to the next iteration of that current loop by skipping the current iteration.
Causes	It performs the termination of the loop.	It performs early execution of the next loop by skipping the current one.
Continuation	It stops the continuation of the loop.	It stops the execution of the current iteration.
Other	It can be used with labels and switches.	It can't be used with labels and switches.

19. How many times will 'Hello World' be printed in the following program? [Model Question]

```
count = 1
while count <= 10:
    print('Hello World')
    count = count + 3
```

Answer:

4

20. What is wrong with this Python loop: [Model Question]

```
n = 10
while n > 0 :
    print(n)
print('All done')
```

Answer:

It will run keep running forever.

21. What will the following code display? [Model Question]

```
sum = 0
for count in range(1, 6):
    sum = sum + count
print(sum)
```

Answer:

15

22. What will the following code display? [Model Question]

```
for number in range(6, 66, 6):
    print(number, end=' ')
```

Answer:

6 12 18 24 30 36 42 48 54 60

23. What will the following code display? [Model Question]

```
for number in range(10, 5, -1):
    print(number)
```

Answer:

10
9
8
7
6

[Model Question]

24. Write the output of following code.

```
val = 10
total = 0

for count in range(1,val,3):
    total = total + count
    if count % 2 == 0:
        print(count*10)
    else:
        print(count)
print (total)
```

Answer:

1
40
7
12

[Model Question]

25. Write the output of following code.

```
num = 532
r = 0
while num > 0:
    digit = num % 10
    r = digit + r * 10
    num = num // 10
print(r)
```

Answer:

235

[Model Question]

26. Write the output of following code.

```
for i in range(1,6):
    for j in range(1,i+1):
        print("*",end='')
    print()
```

Answer:

```
*
**
***
****
*****
```

Long Answer Type Questions

1. a) Explain different types of loops available in Python with suitable examples.
b) Write a python program to check whether the number given is a palindrome or not.
c) Write a Python program to calculate the length of a string.
d) Write a python function to print multiplication table from 1 to 10.
[WBSCTE 2022]

Answer:

a) There are different types of loops depending on the position at which condition is checked.

Infinite Loop: A loop becomes infinite loop if a condition never becomes `False`. This results in a loop that never ends. Such a loop is called an infinite loop. We can create an infinite loop using `while` statement. If the condition of `while` loop is always `True`, we get an infinite loop. We must use `while` loops with caution because of the possibility that the condition may never resolves to a `False` value.

Example Program:

```
count=1
while count==1:
  n=input("Enter a Number:")
  print("Number=", n)
```

This will continue running unless you give CTRL+C to exit from the loop.

Loops with condition at the top:

This is a normal `while` loop without `break` statements. The condition of the `while` loop is at the top and the loop terminates when this condition is `False`.

Loop with condition is the middle:

This kind of loop can be implemented using an infinite loop along with a conditional break in between the body of the loop Fig. 1 shows the flow chart of a loop with condition in the middle.

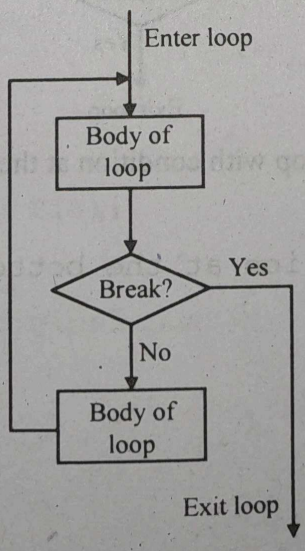


Fig: 1 Loop with condition in the middle

Example Program:

```
vowels='aeiou'
# Infinite Loop
while True:
    v=input("Enter a letter:")
    if v in vowels:
        print(v, "is a vowel")
        break
    print("This is not a vowel, Enter another letter")
print("End of Program")
```

Output:

Enter a letter: t
This is not a vowel, Enter another letter
Enter a letter: o
o is a vowel
End of Program

Loop with condition at the bottom:

This kind of loop ensures that the body of the loop is executed at least once. It can be implemented using an infinite loop along with a conditional break at the end. this is similar to the `do...while` loop in C. Fig. 2 shows the flow chart of loop with condition at the bottom.

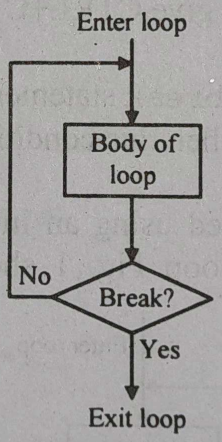


Fig: 2 Loop with condition at the bottom

Example Program:

```
# Demo of loop with condition at the bottom
choice=0
a, b=6, 3
while choice !=5:
 print("Menu")
 print("1. Addition")
 print("2. Subtraction")
 print("3. Multiplication")
 print("4. Division")
 print("5. Exit")
```

```
choice=int(input("Enter your choice:"))
if choice==1: print("Sum=", (a+b))
if choice==2: print("Difference=", (a-b))
if choice==3: print("Product=", (a*b))
if choice==4: print("Quotient=", (a/b))
if choice==5: break
print("End of Program")
```

Output:

Menu
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Exit

Enter your choice: 2
Difference=3
Menu
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Exit

Enter your choice: 5
End of Program

b)

```
print("Enter a Number \n")
num = int(input())
rev = 0

#Copying the original number
temp = num

#Finding Reverse
while temp > 0:
    rev = (rev*10) + (temp %10);
    temp = temp//10

#Comparing reverse with original number
if rev == num :
    print("Palindrome \n")
else:
    print("Not Palindrome")
```

Output:

Enter a Number

101
Palindrome

Enter a Number

145
Not Palindrome

c) Python Program to Calculate the Length of a String without Using Built-In `len()` Function:

```
1. def main():
2. user_string=input("Enter a string:")
3. count_character=0
4.     for each_character in user_string:
5.       count_character+=1
6.     print(f"The length of user entered string is
{count_character}")
7. if __name__== "__main__":
8.     main()
```

Output:
Enter a string: To answer before listening that is folly and shame
The length of user entered string is 50

d)

```
1.number=int(input("Enter the number of which the user wants to p
rint the multiplication table: "))
2.count=1
3.# we are using while loop for iterating the multiplication 10
times
4.print("The Multiplication Table of: ", number)
5.while count<=10:
6.   number=number*1
7.   print(number, 'x', i, '=', number*count)
8.   count+=1
```

➲ **2. Explain If-else Statement used in Python Programming language.**

[Model Question]

Answer:
Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.
In python, decision making is performed by the following statements.

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.
If - else Statement	The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.
Nested if Statement	Nested if statements enable us to use if ? else statement inside an outer if statement.

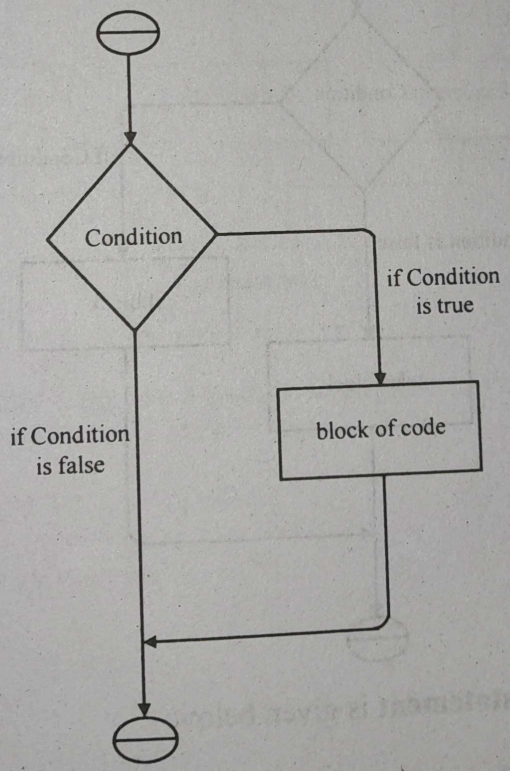
3. Explain the statements with flowchart and examples: **[Model Question]**

i) if Statement
ii) if–else Statement
iii) elif Statement

Answer:

i) if Statement:

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.



The syntax of the if-statement is given below:

if expression:
statement

Example: Program to print the largest of the three numbers.

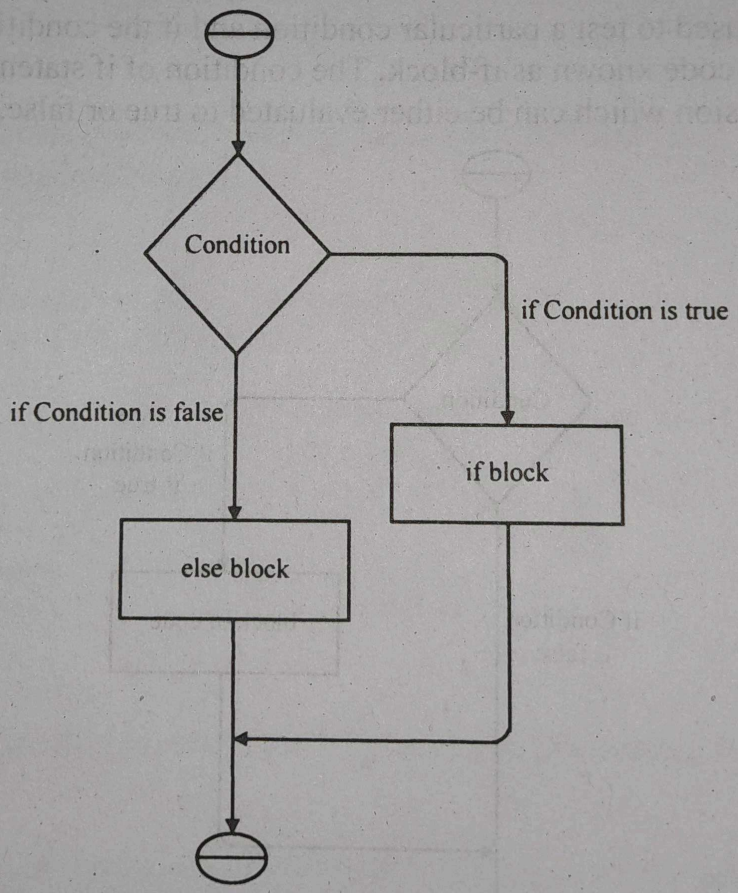
```
a = int(input("Enter a? "))
b = int(input("Enter b? "))
c = int(input("Enter c? "))
if a>b and a>c:
print("a is largest")
if b>a and b>c:
print("b is largest")
if c>a and c>b:
print("c is largest")
```

Output:

Enter a? 100
Enter b? 120
Enter c? 130
c is largest

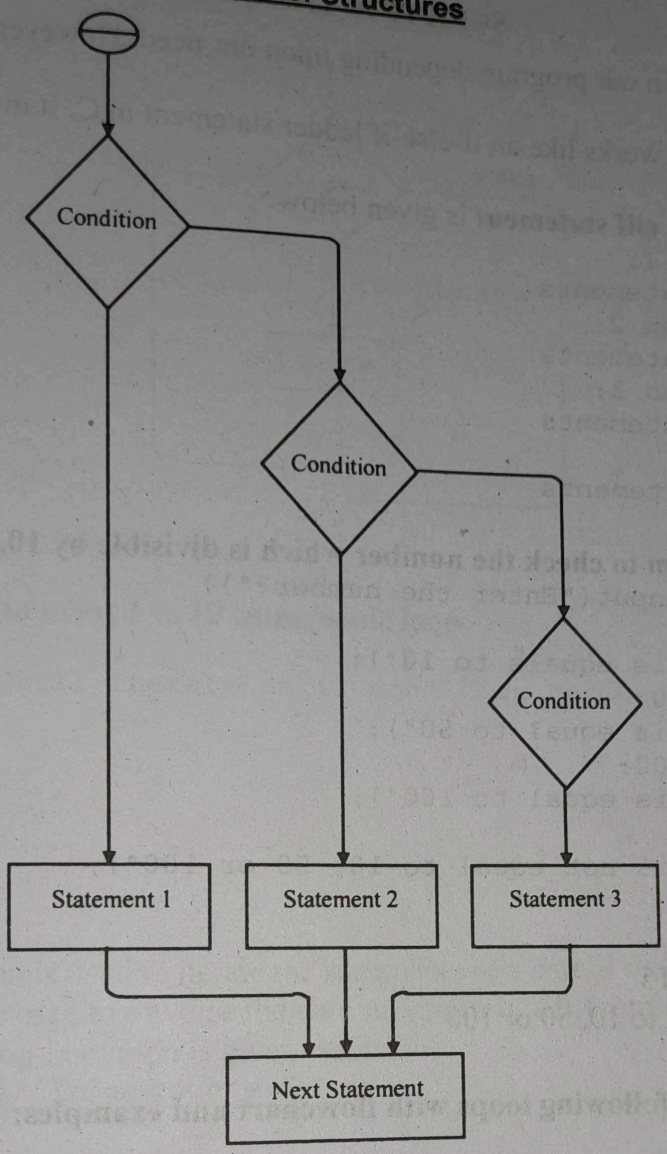
ii) if-else Statement:

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition. If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.



The syntax of the if-else statement is given below:

```
if condition:
#block of statements
else:
#another block of statements (else-block)
```



Example: Program to check whether a person is eligible to vote or not.

```
age = int (input("Enter your age? "))
if age>=18:
 print("You are eligible to vote !!");
else:
print("Sorry! you have to wait !!");
```

Output:

Enter your age? 90
You are eligible to vote !!

iii) elif Statement:

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number

of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

The syntax of the elif statement is given below:

```
if expression 1:
# block of statements
elif expression 2:
# block of statements
elif expression 3:
# block of statements
else:
# block of statements
```

Example: Program to check the number which is divisible by 10,50 and 100.

```
number = int(input("Enter the number?"))
if number==10:
print("number is equals to 10");
elif number==50:
print("number is equal to 50");
elif number==100:
print("number is equal to 100");
else:
print("number is not equal to 10, 50 or 100");
```

Output:

Enter the number? 15
number is not equal to 10, 50 or 100

➲ **4. Explain the following loops with flowchart and examples:** **[Model Question]**

i) While loop

ii) for loop

Answer:

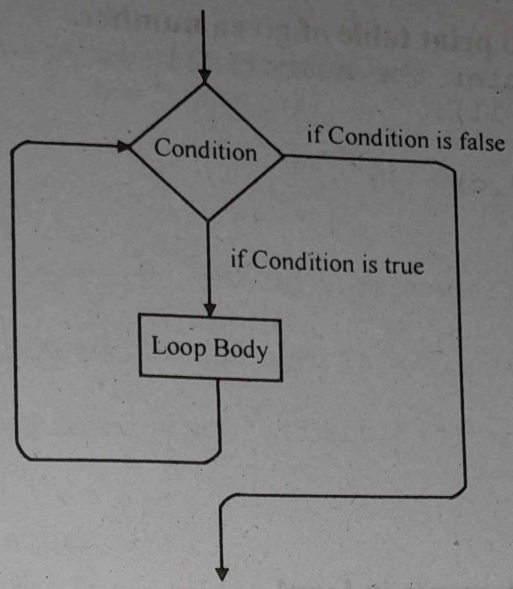
i) While loop:

The Python while loop allows a part of the code to be executed until the given condition returns false. It is also known as a pre-tested loop.It can be viewed as a repeating if statement. When we don't know the number of iterations then the while loop is most effective to use.

The syntax is given below:

```
while expression:
 statements
```

Flowchart:



Example: Program to print 1 to 10 using while loop

```
i=1
#The while loop will iterate until condition becomes false.
While(i<=10):
 print(i)
 i=i+1
```

Output:

1 2 3 4 5 6 7 8 9 10

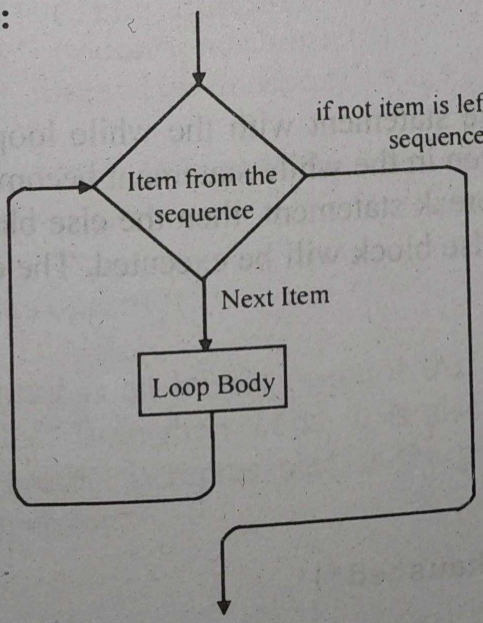
ii) For loop:

The for loop in Python is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

The syntax of for loop in python is given below:

```
for iterating_var in sequence:
statement(s)
```

The For loop flowchart:



Example: Program to print table of given number.

```
n = int(input("Enter the number "))
for i in range(1,11):
c = n*i
print(n,"*",i,"=",c)
```

Output:

Enter the number 10
10 * 1 = 10
10 * 2 = 20
10 * 3 = 30
10 * 4 = 40
10 * 5 = 50

➲ **5. Define 'Pass statements in loop'.** **[Model Question]**

Answer:

The pass statement is used to declare the empty loop. It is also used to define empty class, function, and control statement.

Example:

```
# An empty loop
str1 = 'javatpoint'
i = 0
while i < len(str1):
i += 1
pass
print('Value of i :', i)
```

Output: Value of i: 10

➲ **6. Define: Using else in while loop. Write a program to print Fibonacci numbers in given limit.** **[Model Question]**

Answer:

1st Part:

Python allows us to use the else statement with the while loop also. The else block is executed when the condition given in the while statement becomes false. Like for loop, if the while loop is broken using break statement, then the else block will not be executed, and the statement present after else block will be executed. The else statement is optional to use with the while loop.

Example:

```
i=1
while(i<=5):
print(i)
 i=i+1
else:
print("The while loop exhausted")
```

Output: 1 2

2nd Part:

A program to print Fibonacci numbers in given limit:

```
terms = int(input("Enter the terms "))
# first two initial terms
a = 0
b = 1
count = 0
# check if the number of terms is Zero or negative
if (terms <= 0):
   print("Please enter a valid integer")
elif (terms == 1):
   print("Fibonacci sequence upto",limit,":")
   print(a)
else:
   print("Fibonacci sequence:")
   while (count < terms) :
       print(a, end = ' ')
       c = a + b
       # updating values
       a = b
       b = c
    count += 1
```

➲ **7. Write a program to get the remainder of two negative number using while loop and modulus (%) operator in Python.** [Model Question]

Answer:

```
   while True:
    x = input(' Do you want to continue (Y / N)? ')
   if x.upper() != 'Y':
       break
 # input two integer number and store it into x and y
  x = int(input (' First number: '))
  y = int(input (' Second number: '))
print("Modulus of negative number is: ", x, "%", y, " = ", x % y,
 sep = " ")
print("Modulus of negative number is: ", y, "%", x, " = ", y % x,
 sep = " ")
```

➲ **8. What is modulus operator?** [Model Question]

Answer:

Python Modulus Operator is an inbuilt operator that returns the remaining numbers by dividing the first number from the second. It is also known as the **Python modulo**. In Python, the modulus symbol is represented as the percentage (%) symbol. Hence, it is called the remainder operator.

Syntax:

```
Rem = X % Y
```

Here X and Y are two integer numbers, and the modulus (%) is used in between to get the remainder where the first number (X) is divided by the second number (Y).
For example, we have two numbers, 24 and 5. And we can get the remainder by using the modulus or modulo operator between the numbers 24 % 5.

Example:

Write a program to get the remainder of two numbers using the while loop and modulus (%) operator in Python.

```
while True: # if the while condition is true if block is executed
  a = input ('Do you want to continue or not (Y / N)? ')
 if a.upper() != 'Y':  # If the user pass 'Y', the following stat
ement is executebreak
  a = int(input (' First number is: ')) # first number
  b = int(input (' Second number is: ')) # second number
 print(a, ' % ', b, ' = ', a % b) # perform a % b
 print(b, ' % ', a, ' = ', b % a) # perform b % a
```

Output:

```
Do you want to continue or not (Y/N)? Y
First number is: 37
Second number is: 5
37%5=2
5%37=5

Do you want to continue or not (Y/N)? Y
First number is: 37
Second number is: 5
24%5=4
5%24=5

Do you want to continue or not (Y/N)? Y
First number is: 37
Second number is: 5
28%5=3
5%28=5
```

➲ 9. Briefly explain the nested loops used in Python Programming Languages.

[Model Question]

Answer:

Python allows us to nest any number of for loops inside a **for** loop. The inner loop is executed n number of times for every iteration of the outer loop. The syntax is given below.

Syntax:

```
for iterating_var1 in sequence:  #outer loop
```

```
for iterating_var2 in sequence:  #inner loop
#block of statements
#Other statements
```

Program:

```
# User input for number of rows
rows = int(input("Enter the rows:"))
# Outer loop will print number of rows
for i in range(0,rows+1):
# Inner loop will print number of Astrisk
  for j in range(i):
  print("*",end = '')
   print()
```

Output:

Enter the rows: 5

```
*
**
***
****
```

➲ **10. Write a Python Program for guessing the number using while loop.** [Model Question]

Answer:

Guessing the number using while loop:

```
Import random
counter = 1
correct_num = random.randint(1,100)
User_input = int(input("Enter the number between 1 and 100"))
print(correct_num)
while user_input! = correct_num:
if user_input<correct_num:
print("guess a higher number")
counter = counter + 1
user_input = int(input("Enter the number between 1 and 100"))
else:
print("lower number")
counter = counter + 1
user_input = int(input("Enter the number between 1 and 100"))
print("Correct guess, the number of attempt is," counter)
```

➲ **11. What are the loops control statements used in Python Programming Languages?** [Model Question]

Answer:

We can change the normal sequence of **while** loop's execution using the loop control statement. When the while loop's execution is completed, all automatic objects defined in

that scope are demolished. Python offers the following control statement to use within the while loop.

1. Continue Statement - When the continue statement is encountered, the control transfer to the beginning of the loop.

Example:

```
# prints all letters except 'a' and 't'
i = 0
str1 = 'javatpoint'
while i < len(str1):
if str1[i] == 'a' or str1[i] == 't':
 i += 1
 continue
print('Current Letter :', a[i])
  i += 1
```

Output:

Current Letter: j
Current Letter: v
Current Letter: p
Current Letter: o
Current Letter: i
Current Letter: n

2. Break Statement - When the break statement is encountered, it brings control out of the loop.

Example:

```
# The control transfer is transferred
# when break statement soon it sees t
i = 0
str1 = 'EDUCATION'
while i < len(str1):
    if str1[i] == 'N':
        i += 1
        break
    print('Current Letter :', str1[i])
    i += 1
```

Output:

Current Letter: E
Current Letter: D
Current Letter: U
Current Letter: C

3. Pass Statement - The pass statement is used to declare the empty loop. It is also used to define empty class, function, and control statement. Let's understand the following example.

Example:

```
# An empty loop
str1 = 'POPULARPUB'
i = 0
while i < len(str1):
    i += 1
    pass
print('Value of i :', i)
```

Output: Value of i: 10

Unit 3: Functions, Modules and Packages

Unit at a glance:

3.1 Functions

Now that we have covered some of the elementary control flow structures in Python, we can start combining them into logical blocks to carry out specific tasks. In particular we can construct pieces of coded that can be repeated when necessary and whose outcome depends on the input parameters provided. In other words, we are talking about functions. A function is a good way to write code that can be repeated and whose outcome typically depends on inputs provided.

A function in Python has the following syntax:

```
def my_function(arg1, arg2=default2, … \
        argn=defaultn):
  '''Docstring (optional)'''
  instructions to be executed
  when executing the function
  return result # optional
```

The function definition starts with the word `def`. Remember that code needs to be indented.

Notice that the function definition starts with the reserved word `def` and the code inside the function is indicated with appropriate indentation.

The input parameters for the function are the dummy variables arg1, arg2, …, argn and as you can see it is possible to define default values for some of these parameters. Parameters with default values must be defined last in the argument list.

The second line in the function definition is called the **documentation string** and its purpose is to describe the actions that are performed by the function. Finally, notice that it is not necessary for a function to return a result.

A documentation string enables us to provide information about what a function does. Make sure you use it!

Let us define a function to calculate the area of a rectangle sides a and b:

```
def rect_area(a, b=1.0):
  '''Calculate the area of a rectangle'''
  return a*b
```

We are defining a default value for the parameter b.

Notice that the parameter b has been given the default value of 1. If we were to call this function with only one parameter, the function will know how to handle the calculations and use the default values when needed.

```
> c=rect_area(20, 2)\
> print(c)
40
```

We can use the function by calling it in the same any other in-built Python function is In the first line of code above, we are calling the `rect_area` function with two parameters, such that we assign the value 20 to a and override the default value of b with 2. As expected the area calculated is 40. Let us try providing only one single value to the function:

```
> c2=rect_area(42.4)
> print(c2)

42.4
```

Here we have only passed the value 42.4 to the function. In this case the value is assigned to a and the default value of b=1 is used in the calculation.

We can include control flow structures in our programmes to make them more useful and flexible. Let us for instance implement a simple function to calculate the factorial of a number:

```
def factorial(n):
  '''Return the factorial of n'''
  f=1
  if n<=1:
    return f
  else:
    while n>0:
      f*=n
      n-=1
    return f
```

A function can use any of the other control flow structures of the language.

`*=` and `-=` indicate repeated operations with the left-hand-side value.

When we pass a number smaller or equal than the one the function expects, it returns the value1 and when the number is greater than 1 the factorial is calculated with a `while` loop. Let us use the function:

```
> print(factorial(-3))
  1
> print(factorial(5))
  120
```

There may be times when it is more convenient to define a simple function on-the-fly, without having to resort of a full `def` structure. In these cases we can exploit the use of the so-called lambda functions:

A lambda function in Python is an anonymous function created at runtime.

```
lambda arg1, arg2, ... : statement
```

where, as before, `arg1, arg2, ...` are the input parameters and statement is the code to be executed with the input provided.

For example, if we needed to calculate the cube of a lit of numbers we could try the following code:

```
x=[1, 3, 6]
g=lambda n: n**3
```

In this case the object `g` is a lambda function that can be called as any other function in Python.

So far nothing too strange: We have initialised a list with the numbers 1, 3 and 6 and then defined a lambda function that calculates the cube of the argument n.
We can now apply this function, for example:

```
> [g(item) for item in x]
 [1, 27, 216]
```

Lambda functions may seem very simple, but it is that simplicity that provides their strength, as it shown above. This can be seen employed for instance in the implementations of PySpark, the Python API for Spark, an open-source cluster computing framework.
Lambda functions are very useful in frameworks such as Spark.

3.2 Scripts and Modules

With the flexibility provided by the possibility of controlling the flow of a set of instructions and the repeatability offered by constructing our own functions, it becomes imperative to be able to store programmes in a way that enable us to use and reuse code.
In Python we are able to do this by saving the instructions that make up a programme in a plain text file saved with the extension `.py`. Furthermore, if we use the interactivity provided by the iPython / Jupyter notebook, it is also possible to save our notebooks in a JSON formatted notebook with the extension `.ipynb`.
Python scripts have the extension `.py` whereas notebooks have the extension `.ipynb`.
It is then possible to execute a Python script from the command line by calling Python followed by the name of the script to be executed. For instance, we can create a script defining a `main` function and a call to it. We can save the function in a script called `firstscript.py` with the following contents:

```
def main():
  '''Print the square of a list of numbers from 0 to n'''
  n=input("Give me a positive number")
  x=range(int(n)+1)
  y=[item**2 for item in x]
  print(y)
main()
```

We are defining a main function in this programme and calling simply with the command main().
In this case we are asking the user for a number `n` with the command `input`. We then use this number to calculate a sequence given by the square of the numbers from 0 to `n` and assign it to the variable `y`. Finally we simply print the list stored in `y`.
Notice that we have used `n+1` for `x` range.
Remember that we have saved the script above, but we have not executed it. We can do this by typing the following command in a terminal in the appropriate path:

```
> python firstscript.py
Give me a positive number: 4
 [0, 1, 4, 9, 16]
```

In this case we have given the value `n=4` as an input.

This is perhaps not the most advanced algorithm to implement, but we can surely see the possibilities. In particular, we can see how we can create scripts to add further functionality to our code and as such the concept of a module becomes natural.

A module is a file or collection of files containing related Python functions and objects to achieve a defined task. These modules enable us to extend the capabilities of the language and create programmes that enable us to carry out specific tasks. Any user is able to create their own modules and packages and make them available to others. Some of these modules are readily available for us to be used and once appropriate installation is done all we need to do is import them whenever we need to use them.

A module is a file containing related Python functions to achieve a specific task.

For example, we can use the `math` module to access some common mathematical functions. Let us create for instance a script that implements a function to calculate the area of a circle. In this case we will need the mathematical constant π to carry out the calculations:

The `math` module contains some common mathematical functions.

```
import math
def area_circ(r):
  return math.pi*r**2
r=3
Area=area_circ(r)
print("The area of a circle with"\
"radius (0) is (1)".format(r, Area))
```

We can use the value of π with `math.pi`.

Running the programme will result in the following output:

```
> python area_circ.py

The area of a circle with radius 3 is 28.2743338
```

Notice that we need to tell the Python interpreter that the constant π is part of the `math` module by using the syntax `math.pi`. In the example above we are importing all the functions of the `math` module. This can be somewhat inefficient in cases where only specific functionality is needed. Instead we could have imported only the value of π as follows:

In some cases it may be more efficient to load only the needed functionality from a module.

```
from math import pi
```

Short Answer Type Questions

A. Choose the correct answer from the given alternatives in each of the following:

1. The input() function takes user's input as a [WBSCTE 2022]
   (a) integer (b) float (c) string (d) character

Answer: (c)

2. What will be the output of the following Python function? [WBSCTE 2022]
min(max(False,–3, –4), 2, 7)
(a) –3 (b) –4 (c) error (d) false
Answer: (d)

3. What will be the output of the Python expression round (4.576)? [WBSCTE 2022]
(a) 4 (b) 4.6 (c) 5 (d) 4.5
Answer: (c)

4. Which function is called an anonymous function? [WBSCTE 2022]
(a) Lambda (b) Map (c) Filter (d) Reduce
Answer: (a)

5. Which of the following functions can help us to find the version of python that we are currently working on? [WBSCTE 2022]
(a) sys.version(1) (b) sys.version(0)
(c) sys.version() (d) sys.version
Answer: (c)

6. Which keyword is used for function in Python language? [Model Question]
(a) Function (b) Def (c) Fun (d) Define
Answer: (b)

7. Python supports the creation of anonymous functions at runtime, using a construct called [Model Question]
(a) pi (b) anonymous
(c) lambda (d) none of these
Answer: (c)

8. Which of the following program can work with _____ parameters. [Model Question]

```
def f(x):
    def f1(*args, **kwargs):
    print ("Matrix")
    return x(*args, **kwargs)
    return f1
```

(a) any number of (b) 0 (c) 1 (d) 2
Answer: (a)

9. Which of the following function is built-in function in python? [Model Question]
(a) factorial( ) (b) print( ) (c) seed( ) (d) sqrt( )
Answer: (b)

10. Which function is called when the following Python program is executed? [Model Question]

```
f = foo( )
format(f)
```

(a) str( ) (b) format (c) __str__( ) (d) __format__( )

Answer: (c)

11. Which one of the following is the use of function in python? [Model Question]

(a) Functions don't provide better modularity for your application
(b) You can't also create your own functions
(c) Functions are reusable pieces of programs
(d) All of the mentioned

Answer: (c)

12. What are the two main types of functions in Python? [Model Question]

(a) System function
(b) Custom function
(c) Built-in function & user defined function
(d) User function

Answer: (c)

B. Fill in the blanks in the following statements:

13. The Python built-in function for viewing variable types is ***type()***. [WBSCTE 2022]

14. To convert '5' into int data type ***int()*** command is used. [WBSCTE 2022]

C. Answer the following questions:

15. What is a lambda function? [WBSCTE 2022]

Answer:
A lambda function is an anonymous function (i.e., defined without a name) that can take any number of arguments but, unlike normal functions, evaluates and returns only one expression.

16. Explain find() function. [WBSCTE 2022]

Answer:
The find() method returns the index of first occurrence of the substring (if found). If not found, it returns -1.

17. How can you generate random numbers? [WBSCTE 2022]

Answer:
To generate random number in Python, randint() function is used. This function is defined in random module.

18. Differentiate between globals() and locals(). [WBSCTE 2022]

Answer:
The python **globals()** function returns the dictionary of the current global symbol table.
A **Symbol table** is defined as a data structure which contains all the necessary information about the program. It includes variable names, methods, classes etc.
Python locals() function returns the dictionary of the current local symbol table.
Local symbol Table: This symbol table stores all information needed for the local scope of the program and this information is accessed using python built-in function locals().

Long Answer Type Questions

➲ **1. a) What is difference between module and package? Is pandas a module or package?**
b) Write a Python program to print the factorial of a number. **[WBSCTE 2022]**

Answer:
a) 1st Part: In Python, module is the way to structure program. Each Python program file is a module, which imports other modules like objects and attributes.
The folder of Python program is a package of modules. A package can have modules or subfolders.
2nd Part: Pandas is **a Python package** providing fast, flexible, and expressive data structures designed to make working with "relational" or "labeled" data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real-world data analysis in Python.

b) Factorial of a Number using Recursion

```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""
    if x == 1:
        return 1
    else:
        # recursive call to the function
        return (x * factorial(x-1))
# change the value for a different result
num = 7
# to take input from the user
# num = int(input("Enter a number: "))
# call the factorial function
result = factorial(num)
print("The factorial of", num, "is", result)
```

➲ **2. What do you mean by function? Explain its types and advantages.** **[Model Question]**

Answer:
The Function is a piece of code written to carry out a specific task. A function can be defined as the organized block of reusable code, which can be called whenever required. Python allows us to divide a large program into the basic building blocks known as a function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the Python program.
The Function helps to programmer to break the program into the smaller part. It organizes the code very effectively and avoids the repetition of the code. As the program grows, function makes the program more organized.
There are mainly two types of functions:

- **User-define functions:** The user-defined functions are those define by the **user** to perform the specific task.
- **Built-in functions:** The built-in functions are those functions that are **pre-defined** in Python.

Advantage of Functions in Python:
There are the following advantages of Python functions.

- Using functions, we can avoid rewriting the same code again and again in a program.
- We can call Python functions multiple times in a program and anywhere in a program.
- We can track a large Python program easily when it is divided into multiple functions.
- Reusability is the main achievement of Python functions.
- Function calling is always overhead in a Python program.

[Model Question]

➲ 3. How to create a function?

Answer:
Python provides the def keyword to define the function. The syntax of the define function is given below:
Syntax:

```
def my_function(parameters):
    function_block
return expression
```

The **def** keyword, along with the function name is used to define the function.

- The identifier rule must follow the function name.
- A function accepts the parameter (argument), and they can be optional.
- The function block is started with the colon (:), and block statements must be at the same indentation.
- The **return** statement is used to return the value. A function can have only one **return.**

[Model Question]

➲ 4. What is return function?

Answer:
The return statement is used at the end of the function and returns the result of the function. It terminates the function execution and transfers the result where the function is called. The return statement cannot be used outside of the function.
It can contain the expression which gets evaluated and value is returned to the caller function. If the return statement has no expression or does not exist itself in the function then it returns the **None** object.

➲ 5. What are the applications of lambda function? **[Model Question]**

Answer:
The lambda function is commonly used with Python built-in functions **filter()** function and **map()** function.

- **Use lambda function with filter():**

The Python built-in **filter() function** accepts a function and a list as an argument. It provides an effective way to filter out all elements of the sequence. It returns the new sequence in which the function evaluates to **True**. Consider the following example where we filter out the only odd number from the given list:

```
#program to filter out the tuple which contains odd numbers
lst = (10,22,37,41,100,123,29)
oddlist = tuple(filter(lambda x:(x%3 == 0),lst)) # the tuple cont
ains all
the items of the tuple for which the lambda function evaluates to
 true
print(oddlist)
```

Output: (37, 41, 123, 29)

- **Using lambda function with map():**

The **map() function** in Python accepts a function and a list. It gives a new list which contains all modified items returned by the function for each item. Consider the following example of **map()** function.

```
#program to filter out the list which contains odd numbers
lst = (10,20,30,40,50,60)
square_list = list(map(lambda x:x**2,lst)) # the tuple contains a
ll the
items of the list for which the lambda function evaluates to true
print(square_tuple)
```

Output: (100, 400, 900, 1600, 2500, 3600)

➲ 6. What is Type conversion in Python? **[Model Question]**

Answer:
Python defines type conversion functions to directly convert one data type to another.
There are two types of Type Conversion in Python:

1. Implicit Type Conversion:

In Implicit type conversion of data types in Python, the Python interpreter automatically converts one data type to another without any user involvement.

Example:

```
x = 10
print("x is of type:",type(x))
y = 10.6
print("y is of type:",type(y))
x = x + y
print(x)
print("x is of type:",type(x))
```

Output:

x is of type: <class 'int'>
y is of type: <class 'float'>
20.6
x is of type: <class 'float'>

Here we can see the type of 'x' got automatically changed to the "float" type from the "integer" type. this is a simple case of Implicit type conversion in python.

2. Explicit Type Conversion:

In Explicit Type Conversion in Python, the data type is manually changed by the user as per their requirement. Various forms of explicit type conversion are explained below:

1. **int(a, base)**: This function converts any data type to integer. 'Base' specifies the base in which string is if the data type is a string.
2. **float()**: This function is used to convert any data type to a floating-point number.
3. **ord()** : This function is used to convert a character to integer.
4. **hex()** : This function is to convert integer to hexadecimal string.
5. **oct()** : This function is to convert integer to octal string.
6. **tuple()** : This function is used to convert to a tuple.
7. **set()** : This function returns the type after converting to set.
8. **list()** : This function is used to convert any data type to a list type.
9. **dict()** : This function is used to convert a tuple of order (key,value) into a dictionary.
10. **str()** : Used to convert integer into a string.
11. **complex(real,imag)** : This function converts real numbers to complex(real,imag) number.

Example:

```
# Python code to demonstrate Type conversion
# using  tuple(), set(), list()
# initializing string
s = 'Matrix'
# printing string converting to tuple
c = tuple(s)
print ("After converting string to tuple : ",end="")
print (c)
# printing string converting to set
c = set(s)
print ("After converting string to set : ",end="")
print (c)
```

```
# printing string converting to list
c = list(s)
print ("After converting string to list : ",end="")
print (c)
```

Output:

After converting string to tuple : ('M', 'a', 't', 'r', 'i', 'x')
After converting string to set : {'t', 'a', 'r', 'i'}
After converting string to list : ['g', 'e', 'e', 'k', 's']

➲ 7. What are the types of function arguments in Python? Explain. **[Model Question]**

Answer:

There may be several types of arguments which can be passed at the time of function call.

a. Required arguments
b. Default arguments
c. Variable-length arguments
d. Keyword arguments

a. Required Arguments:

We can provide the arguments at the time of the function call. As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling. If either of the arguments is not provided in the function call, or the position of the arguments is changed, the Python interpreter will show the error.

Example:

```
def func(name):
    message = "Hi "+name
    return message
name = input("Enter the name:")
print(func(name))
```

Output:

Enter the name: John
Hi John

b. Default Arguments:

Python allows us to initialize the arguments at the function definition. If the value of any of the arguments is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

Example:

```
def printme(name,age=22):
    print("My name is",name,"and age is",age)
printme(name = "john")
```

Output:

My name is John and age is 22

c. Variable-length Arguments (*args):

In large projects, sometimes we may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to offer the comma-separated values which are internally treated as tuples at the function call. By using the variable-length arguments, we can pass any number of arguments.

However, at the function definition, we define the variable-length argument using the ***args** (star) as *<variable - name >.

Example:

```
def printme(*names):
    print("type of passed argument is ",type(names))
    print("printing the passed arguments...")
    for name in names:
        print(name)
printme("john","David","smith","nick")
```

Output:

type of passed argument is <class 'tuple'>
printing the passed arguments...
john
David
smith
nick

In the above code, we passed ***names** as variable-length argument. We called the function and passed values which are treated as tuple internally. The tuple is an iterable sequence the same as the list. To print the given values, we iterated ***arg names** using for loop.

d. Keyword arguments(kwargs):**

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

The name of the arguments is treated as the keywords and matched in the function calling and definition. If the same match is found, the values of the arguments are copied in the function definition.

Example:

```
#function func is called with the name and message as the keyword
 arguments
def func(name,message):
    print("printing the message with",name,"and ",message)
    #name and message is copied with the values John and hello re
spectively
    func(name = "John",message="hello")
```

Output: printing the message with John and hello

➲ **8. What is Lambda function? Illustrate with example.** **[Model Question]**

Answer:

Python Lambda function is known as the anonymous function that is defined without a name. Python allows us to not declare the function in the standard manner, i.e., by using the **def** keyword. Rather, the anonymous functions are declared by using the **lambda** keyword. However, Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.

The anonymous function contains a small piece of code. It simulates inline functions of C and C++, but it is not exactly an inline function. It can accept any number of arguments and has only one expression. It is useful when the function objects are required.The syntax to define an anonymous function is given below:

Syntax:

lambda arguments: expression

Example: Multiple arguments to Lambda function

```
# a and b are the arguments and a*b is the expression which gets
evaluated and returned.
x = lambda a,b: a*b
print("mul = ", x(20,10))
<p><strong>Output:</strong></p>
<div class="codeblock3"><pre>
mul =  200
</pre></div>
<h2 class="h2">Why use lambda function?</h2>
<p>The main role of the lambda function is better described in th
e scenarios when we use them anonymously inside another function.
 In Python, the lambda function can be used as an argument to the
 <strong>higher-
order functions</strong> which accepts other functions as argumen
ts.</p>
<p>Consider the following example:</p>
<h3 class="h3">Example 1:</h3>
<div class="codeblock"><textarea name="code" class="python">
#the function table(n) prints the table of n
def table(n):
    return lambda a:a*n # a will contain the iteration variable i
 and a multiple of n is returned at each function call
n = int(input("Enter the number:"))
b = table(n) #the entered number is passed into the function tabl
e. b will contain a lambda function which is called again and aga
in with the iteration variable i
for i in range(1,11):
    print(n,"X",i,"=",b(i)) #the lambda function b is called with
 the iteration variable i
```

Output:

Enter the number: 10

10×1 = 10

10×2 = 20

$10 \times 3 = 30$
$10 \times 4 = 40$
$10 \times 5 = 50$
$10 \times 6 = 60$
$10 \times 7 = 70$
$10 \times 8 = 80$
$10 \times 9 = 90$
$10 \times 10 = 100$

➲ 9. Write short note on Function composition in Python. **[Model Question]**

Answer:

Function composition is the way of combining two or more functions in such a way that the output of one function becomes the input of the second function and so on. For example, let there be two functions "F" and "G" and their composition can be represented as F(G(x)) where "x" is the argument and output of G(x) function will become the input of F() function.

Code Example:

```
# Function to add 2
# to a number
def add(x):
    return x + 2

# Function to multiply
# 2. to a number
def multiply(x):
    return x * 2

# Printing the result of
# composition of add and
# multiply to add 2 to a number
# and then multiply by 2
print("Adding 2 to 5 and multiplying the result with 2: ",
      multiply(add(5)))
```

Output:

Adding 2 to 5 and multiplying the result with 2: 14

➲ 10. What are python modules? What are packages? **[Model Question]**

Answer:

1st Part:

As programs become larger and more complex, it becomes challenging to track errors and analyze its functionality, but different Python modules can be packaged and easily inserted into programs. A module is a file containing Python definitions and statements. The advantages of using modules include:

- Functions and variables must be defined only once and then they can be used in many programs without writing it again
- Rewrites code
- Allows a program to be organized into several logical sections, each placed in a separate file
- Developers can easily share components

The Python language can be supplemented with many properties by importing modules that provide certain functions and classes that can perform certain tasks. The modules are added to the code by typing import followed by the module name you wish to use.

Writing Modules:

Any program you write and save can be imported as a module. For example, if you save a program with the name prog.py, it can be imported using the 'import' command. However, the "import" only occurs once. Python assumes that variables and functions are not changed and the module code merely serves to initialize these elements.

After importing a module the module is compiled and generates a .pyc file. Python only recompiles a program if the .py is newer than the .pyc file.

Initialization Codes:

In out modules we can define code that runs automatically when they are imported. If you run the program the first example, the message 'Booting strformat module', which we define in strformat appears.

The rule is simple: the code that is defined in the first level, i.e., outside the definition of classes and functions will be executed as startup module. However, there are some situations where we want our code to run only under special conditions. this is the case of the main modules. We just want our function main() executed if the module is the main. If it has been imported, the application should only be performed if main() is called explicitly. For this, we use the following code:

```
1. if __name__ == "__ main__":
2. main()
```

The "__name__" variable stores the name of the current module. In this case, the initialization code investigates whether the module is the main and performs accordingly. You will find this code very often. It may seem sort of common, but this is the correct way to do core modules.

2nd Part:

When our modules get larger we do not want to have three hundred fifty classes and functions into a single file. We'll want to separate into several modules. That's why packages exist.

Python packages are packages that can contain other modules. In terms of storage modules are structured in files, packages are structured in folders.

Python is a standard library, used for a variety of tasks but there are many implementations of the language such as:

- CPython is the original implementation,
- IronPython is the implementation,

- Stackless Python CPython is a variant
- Jython implementation is for developing in Java
- Pippy is the implementation on Palm
- PyPy is an implementation of Python optimized by JIT

➲ 11. Write about python packages. **[Model Question]**

Answer:

The packages in python facilitate the developer with the application development environment by providing a hierarchical directory structure where a package contains sub-packages, modules and sub-modules. The packages are used to categorize the application level code efficiently.

Let's create a package named Employees in your home directory. Consider the following steps.

a. Create a directory with name Employees on path/**home**.

b. Create a python source file with name ITEmployees.py on the path/**home/Employees**.

ITEmployees.py

```
1. def getITNames():
2.   List=["John", "David", "Nick", "Martin"]
3.   return List;
```

c. Similarly, create one more python file with name BPOEmployees.py and create a function getBPONames().

d. Now, the directory Employees which we have created in the first step contains two python modules. To make this directory a package, we need to include one more file here, that is __init__.py which contains the import statements of the modules defined in this directory.

__init__.py

```
1. from ITEmployees import getITNames
2. from BPOEmployees import get BPONames
```

e. Now, the directory **Employees** has become the package containing two python modules. Here we must notice that we must have to create __init__.py inside a directory to convert this directory to a package.

f. To use the modules defined inside the package Employees, we must have to import this in our python source file. Let's create a simple python source file at our home directory (/home) which uses the modules defined in this package.

Test.py

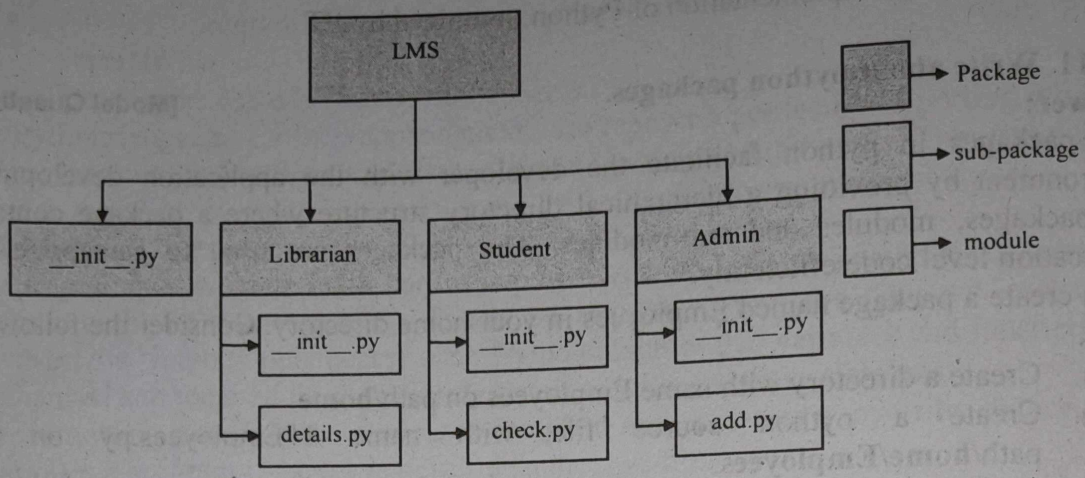
```
1. import Employees
2. print(Employees. getNames())
```

Output:

['John', 'David', 'Nick', 'Martin']

We can have sub-packages inside the packages. We can nest the packages up to any level depending upon the application requirements.

The following image shows the directory structure of an application Library management system which contains three sub-packages as Admin, Librarian and Student. The sub-packages contain the python modules.



12. How to load a module in python code. **[Model Question]**

Answer:

We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below:

1. The import statement
2. The from-import statement

The import statement:

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times, it has been imported into our file.

The syntax to use the import statement is given below:

Import module 1, module 2, …….. module n

Hence, if we need to call the function `displayMsg()` defined in the file file.py, we have to import that file as a module into our module as shown in the example below:

Example:

```
1. import file;
2. name=input("Enter the name?")
3. file.displayMsg(name)
```

Output:

Enter the name? John

The from-import statement

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using

'from<modulename> import<attribute>' statement. The syntax to use the from-import statement is given below.

from <module-name> import <name 1>, <name 2> .., ,name n>

Consider the following module named as calculation which contains three functions as summation, multiplication and divide.

calculation.py:

```
1. #place the code in the calculation.py
2. def summation(a, b):
3.   return a+b
4. def multiplication(a, b):
5.   return a*b;
6. def divide(a, b):
7.   return a/b;
```

Main.py:

```
1. from calculation import summation
2. #it will import only the summation() from calculation.py
3. a=int(input("Enter the first number"))
4. b=int(input("Enter the second number"))
5. print("Sum=", summation(a, b)) #we do not need to specify the
module name while accessing summation()
```

Output:

```
Enter the first number 10
Enter the second number 20
Sum=30
```

The from ... import statement is always better to use if we know the attributes to be imported from the module in advance. It doesn't let our code to be heavier. We can also import all the attributes from a module by using *.

Consider the following syntax.

from <module> import *

Unit 4: File I/O, Text Processing, Regular Expression

Unit at a glance:

4.1 Opening a file

Python provides the `open()` function which accepts two arguments, file name and access mode in thich the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

The syntax to use the `open()` function is given below:

```
file object=open(<file-name>, <access-mode>, <buffering>)
```

The files can be accessed using various modes like read, write, or append. The following are the details about the access moded to open a file.

SN	Access mode	Description
1	*r*	It opens the file to read-only. The file pointer exists at the beginning. The file is by default open in this mode if no access mode is passed.
2	*rb*	It opens the file to read-only in binary format. The file pointer exists at the beginning of the file.
3	*r+*	It opens the file to read and write both. The file pointer exists at the beginning of the file.
4	*rb+*	It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.
5	*w*	It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name. the file pointer exists at the beginning of the file.
6	*wb*	It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists with the same name. the file pointer exists at the beginning of the file.
7	*w+*	It opens the file to write and read both. it is different from r+ in the sense that it overwrites the previous file if one exists whereas r+ doesn't overwrite the previously written file. It creates a new fiel if no file exists. The file pointer exists at the beginning of the file.
8	*wb+*	It opens the file to write and read both in binary format. The file pointer exists at the beginning of the file.
9	*a*	It opens the file in the append mode. The file pointer exists at the end of the previously written file if exists any. It creates a

SN	Access mode	Description
		new file if no file exists with the same name.
10	*ab*	It opens the file in the append mode in binary format. The pointer exists at the end of the previously written file. It creates a new file in binary format if no file exists with the same name.
11	*a+*	It opens a file to append and read both. the file pointer remains at the end of the file if a file exists. It creates a new file if no file exists with the same name.
12	*ab+*	It opens a file to append and read both in binary format. The file pointer remains at the end of the file.

Let's look at the simple example to open a file named "file.txt" (stored in the same directory) in read mode and printing its content on the console.

Example:

```
1. #opens the file file.txt in read mode
2. fileptr=open("file.txt", "r")
3.
4. if fileptr:
5.     print("file is opened successfully")
```

Output:

<class '_io.TextIOWrapper'>
file is opened successfully

4.2 The close() method

Once all the operations are done on the file, we must close it through our python script using the `close()` method. Any unwritten information gets destroyed once the close() method is called on a file object.

We can perform any operation on the file externally in the file system is the file is opened in python, hence it is good practice to close the file once all the operations are done.

The syntax to use the `close()` method is given below:

```
fileobject.close()
```

consider the following example.

Example:

```
1. #opens the file file.txt in read mode
2. fileptr=open("file.txt", "r")
3.
4. if fileptr:
5.     print("file is opened successfully")
6.
7. #closes the opened file
```

```
8. fileptr.close()
```

4.3 Reading the file

To read a file using the python script, the python provides us the `read()` method. The `read()` method reads a string from the file. It can read the data in the text as well as binary format. The syntax of the `read()` method is given below:

```
fileobj.read(<count>)
```

Here, the count is the number of bytes to be read from the file starting from the beginning of the file. If the count is not specified, then it may read the content of the file until the end.

Consider the following example.

Example:

```
1. #open the file.txt in read mode. Causes error if no such file exists.
2. fileptr=open("file.txt", "r");
3.
4. #stores all the data of the file into the variable content
5. content=fileptr.read(9);
6.
7. #prints the type of the data stored in the file
8. print(type(content))
9.
10. #prints the content of the file
11. print(content)
12.
13. #closes the opened file
14. fileptr.close()
```

Output:

<class 'str'>
Hi, I am

4.4 Read Lines of the file

Python facilitates us to read the file line by line by using a function readline(). The readline() method reads the lines of the file from the beginning, i.e., if we use the `readline()` method two times, then we can get the first two lines of the file.

Consider the following example which contains a function `readline()` that reads the first line of our file "**file.txt**" containing three lines.

Example:

```
1. #open the file.txt in read mode. Causes error if no such file exists.
2. fileptr=open("file.txt", "r");
3.
```

```
#stores all the data of the file into the variable content
content=fileptr.readline();

#prints the type of the data stored in the file
print(type(content))

#prints the content of the file
print(content)

#closes the opened file
fileptr.close()
```

Output:

<class 'str'>
Hi, I am the file and being used as

4.5 Writing the file

To write some text to a file, we need to open the file using the open method with one of the following access modes.

a: It will append the existing file. the file pointer is at the end of the file. It creates a new file if no file exists.

w: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

Consider the following example.

Example:

```
#open the file.txt in append mode. Creates a new file if no such file exists.
fileptr=open("file.txt", "a");

#appending the content to the file
fileptr.write("Python is the modern day language. It makes things so simple.")

#closing the opened file
fileptr.close();
```

Now, we can see that the content of the file is modified.

File.txt:

Hi, I am the file and being used as
an example to read a
file in python.
Python is the modern day language. It makes things so simple.

4.6 Creating a new file

The new file can be created by using one of the following access modes with the function `open()`.

x: It creates a new file with the specified name. It causes an error a file exists with the same name.

a: It creates a new file with the specified name if no such file exists. It appends the content to the file if the file already exists with the specified name.

w: It creates a new file with the specified name if no such file exists. It overwrites the existing file.

Consider the following example.

Example:

```
1. #open the file.txt in read mode. Causes error if no such file
exists.
2. fileptr=open("file2.txt", "x");
3.
4. print(fileptr)
5.
6. if fileptr:
7.     print("File created successfully");
```

Output:

File created successfully

Short Answer Type Questions

A. Choose the correct answer from the given alternatives in each of the following:

1. The default access mode is [WBSCTE 2022]

   (a) r (b) w (c) rb (d) wb

Answer: (a)

2. Which one of the following is not a keyword in Python language? [WBSCTE 2022]

   (a) pass (b) eval (c) assert (d) nonlocal

Answer: (b)

B. Fill in the blanks in the following statements:

3. ***remove()*** method is used to delete a file. [WBSCTE 2022]

4. Open() function is used for ***open() internally stored files***. [WBSCTE 2022]

C. Answer the following questions:

5. What is the purpose of isatty()? [WBSCTE 2022]

Answer:

isatty() returns 1 if the given file descriptor is a terminal, or 0 otherwise.

6. What is the purpose of flush() ? [WBSCTE 2022]

Answer:

The flush() method of OutputStream class is used to flush the content of the buffer to the output stream.

7. Name two file build in functions. Give their syntax. [WBSCTE 2022]

Answer:

Python has a built-in function open() to open a file. Which accepts two arguments, file name and access mode in which the file is accessed. The function returns a file object which can be used to perform various operations like reading, writing, etc.

Syntax:

```
Fileobject=open(file-name, access-mode)
```

The `close()` method used to close the currently opened file, after which no more writing or Reading can be done.

Python automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax:

```
Fileobject.close()
```

Long Answer Type Questions

➲ **1. What are input and output files? How do use input and output files in Python?** [Model Question]

Answer:

1st Part:

File Input/Ouput (IO) requires 3 steps:

1. Open the file for read or write or both.
2. Read/Write data.
3. Close the file to free the resouces.

Python provides built-in functions and modules to support these operations.

2nd Part:

Refer to Unit at glance

➲ **2. How to loop through files?** [Model Question]

Answer:

Looping through the file:

By looping through the lines of the file, we can read the whole file.

Example:

```
1. #open the file.txt in read mode. Causes an error if no such file exists.
2.
3. fileptr=open("file.txt", "r");
4.
5. #running a for loop
6. for i in fileptr:
7.     print(i) #i contains each line of the file
```

Output:

```
Hi, I am the file and being used as
an example to read a
file in python.
```

➲ **3. What is file pointer? What do you understand by file pointer position and how to modify it?** **[Model Question]**

Answer:

1st Part:

A file pointer **stores the current position of a read or write within a file.** All operations within the file are made with reference to the pointer. The data type of this pointer is defined in stdio. h and is named FILE.

2nd Part:

File Pointer Positions:

Python provides the `tell()` method which is used to print the byte number at which the file pointer exists. Consider the following example.

Example:

```
1. #open the file file2.txt in read mode
2. fileptr=open("file2.txt", "r");
3.
4. #initially the filepointer is at 0
5. print("The filepointer is at byte:", fileptr.tell())
6.
7. #reading the content of the file
8. content=fileptr.read();
9.
10. #after the read operation file pointer modifies. tell() returns the location of the fileptr.
11.
```

```
12. print("After reading, the filepointer is at:", fileptr.tell())
```

Output:

The filepointer is at byte: 0
After reading, the filepointer is at 26

Modifying file pointer position:

In the real world applications, sometimes we need to change the file pointer location externally since we may need to read or write the content at various locations.

For this purpose, the python provides us the `seek()` method which enables us to modify the file pointer position externally.

The syntax to use the `seek()` method is given below:

```
<file-ptr>.seek(offset[, from)
```

The `seek()` method accepts two parameters:

offset: It refers to the new position of the file pointer within the file.

from: It indicates the reference position from where the bytes are to be moved. If it is set to 0, the beginning of the file is used as the reference position. if it is set to 1, the current position of the file pointer is used as the reference position. if it is set to 2, the end of the file pointer is used as the reference position.

Consider the following example:

Example:

```
1. #open the file file2.txt in read mode
2. fileptr=open("file2.txt", "r");
3.
4. #initially the filepointer is at 0
5. print("The filepointer is at byte:", fileptr.tell())
6.
7. #changing the file pointer location to 10.
8. fileptr.seek(10);
9.
10. #tell() returns the location of the fileptr.
11. print("After reading, the filepointer is at:", fileptr.tell())
```

Output:

The filepointer is at byte: 0
After reading, the filepointer is at 10

➲ **4. What are different file related methods available in python?** **[Model Question]**

Answer:

The file related methods

The file object provides the following methods to manipulate the files on various operating systems.

SN	Access mode	Description
1	`file.close()`	It closes the opened file. the file once closed, it can't be read or write any more.
2	`File.fush()`	It flushes the internal buffer.
3	`File.fileno()`	It returns the file descriptor used by the underlying implementation to request I/O from the OS.
4	`File.isatty()`	It returns true if the file is connected to a TTY device, otherwise returns false.
5	`File.next()`	It returns the next line from the file.
6	`File.read([size])`	It reads the file for the specified size.
7	`File.readline([size])`	It reads one line from the file and places the file pointer to the beginning of the new line.
8	`File.readlines([sizehint])`	It returns a list containing all the lines of the file. It reads the file until the EOF occurs using `readline()` function.
9	`File.seek(offset[,from)`	It modifies the position of the file pointer to a specified offset with the specified reference.
10	`File.tell()`	It returns the current position of the file pointer within the file.
11	`File.truncate([size])`	It truncates the file to the optional specified size.
12	`File.write(str)`	It writes the specified string to a file.
13	`File.writelines(seq)`	It writes a sequence of the strings to a file.

➲ 5. What are different file processing operations available in python os module?

[Model Question]

Answer:

Python os module:

The os module provides us the functions that are involved in file processing operations like renaming, deleting, etc.

Let's look at some of the os module functions.

Renaming the file:

The os module provides us the `rename()` method which is used to rename the specified file to a new name. The syntax to use the rename() method is given below:

```
rename(?current-name?, ?new-name?)
```

Example:

```
1. import os;
2.
3. #rename file2.txt to file3.txt
4. os.rename("file2.txt", "file3.txt")
```

Removing the file:

The os module provides us the `remove()` method which is used to remove the specified file. The syntax to use the `remove()` method is given below:

```
remove(?file-name?)
```

Example:

```
1. import os;
2.
3. #deleting the file named file3.txt
4. os.remove("file3.txt")
```

Creating the new directory:

The `mkdir()` method is used to create the directories in the current working directory. The syntax to create the new directory is given below:

```
mkdir(?directory name?)
```

Example:

```
1. import os;
2.
3. #creating a new directory with the name new
4. os.mkdir("new")
```

Changing the current working directory:

The `chdir()` method is used to change the current working directory to a specified directory.

The syntax to use the `chdir()` method is given below:

```
chdir("new-directory")
```

Example:

```
1. import os;
2.
3. #changing the current working directory to new
4.
5. os.chdir("new")
```

The getcwd() method:

This method returns the current working directory. The syntax to use the `getcwd()` method is given below:

```
os.getcwd()
```

Example:

```
1. import os;
2.
3. #printing the current working directory
4. print(os.getcwd())
```

Deleting directory:

The `rmdir()` method is used to delete the specified directory.

The syntax to use the `rmdir()` method is given below:

```
os.rmdir(?directory name?)
```

Example:

```
1. import os;
2.
3. #removing the new directory
4. os.rmdir("new")
```

➲ **6. How to write python output to the files?** **[Model Question]**

Answer:

Writing python output to the files:

In python, there are the requirements to write the output of a python script to a file.

The `check_call()` method of module **subprocess** is used to execute a python script and write the output of that script to a file.

The following example contains two python scripts. The script file1.py executes the script file.py and writes its output to the text file **output.txt**

file.py:

```
1. temperatures=[10, -20, -289, 100]
2. def c_to_f(c):
3.   if c<-273.15:
4.      return "That temperature doesn't make sense!"
5.   else:
6.      f=c*9/5+32
7.      return f
8. for t in temperatures:
9.   print(c_to_f(t))
```

file.py:

```
1. import subprocess
2.
3. with open("output.txt", "wb") as f:
4.   subprocess.check_call(["python", "file.py"], stdout=f)
```

Output:

```
50
-4
That temperature doesn't make sense!
212
```

7. Discuss few text processing methods in python. **[Model Question]**

Answer:

For simple text string operations such as string search and replacement, you can use the built-in string functions (e.g., str.replace(*old*, *new*)). For complex pattern search and replacement, you need to master regular expression (regex).

The built-in class str provides many member functions for text string manipulation. Suppose that *s* is a str object.

Strip whitespaces (blank, tab and newline)

- ***s*.strip()-> str**: Return a copy of the string *s* with leading and trailing whitespaces removed. Whitespaces includes blank, tab and newline.
- ***s*.strip([chars]) -> str**: Strip the leading/trailing characters given, instead of whitespaces.
- ***s*.rstrip(), *s*.lstrip() -> str**: Strip the right (trailing) whitespaces and the left (leading) whitespaces, respectively.

s.rstrip() is the most commonly-used to strip the trailing spaces/newline. The leading whitespaces are usually significant.

Uppercase/Lowercase

- ***s*.upper(), *s*.lower() -> str**: Return a copy of string *s* converted to uppercase and lowercase, respectively.
- ***s*.isupper(), *s*.islower() -> bool**: Check if the string is uppercase/lowercase, respectively.

Find

- ***s*.find(*key_str*, [*start*], [*end*]) -> int|-1**: Return the lowest index in slice *s*[*start*:*end*] (default to entire string); or -1 if not found.
- ***s*.index(*key_str*, [*start*], [*end*]) -> int|ValueError**: Similar to find(), but raises ValueError if not found.
- ***s*.startswith(*key_str*, [*start*], [*end*]), *s*.endsswith(*key_str*, [*start*], [*end*]) -> bool**: Check is the string begins or ends with *key_str*.

For examples,

```
>>> s = '/test/in.txt'
>>> s.find('in')
6
>>> s[0 : s.find('in')] + 'out.txt'
'/test/out.txt'
# You could use str.replace() described below
```

Find and Replace

- ***s*.replace(*old*, *new*, [*count*]) -> str**: Return a copy with all occurrences of *old* replaced by *new*. The optional parameter *count* limits the number of occurrences to replace, default to all occurrences.

str.replace() is ideal for simple text string replacement, without the need for pattern matching.

For examples,

```
>>> s = 'hello hello hello, world'
>>> help(s.replace)
>>> s.replace('ll', '**')
'he**o he**o he**o, world'
>>> s.replace('ll', '**', 2)
'he**o he**o hello, world'
```

Split into Tokens and Join

- ***s*.split([*sep*], [*maxsplit*=-1]) -> [str]**: Return a list of words using *sep* as delimiter string. The default delimiter is whitespaces (blank, tab and newline). The *maxSplit* limits the maximum number of split operations, with default -1 means no limit.
- ***sep*.join([str]) -> str**: Reverse of split(). Join the list of string with *sep* as separator.

For examples,

```
>>> 'apple, orange, pear'.split()      # default delimiter is whitespaces
['apple,', 'orange,', 'pear']
>>> 'apple, orange, pear'.split(',')   # Set the delimiter
['apple', 'orange', 'pear']
>>> 'apple, orange, pear'.split(', ', maxsplit=1)  # Set the split operation
['apple', 'orange, pear']
>>> ', '.join(['apple', 'orange, pear'])
'apple, orange, pear'
```

➲ 8. What is Regular Expression and re module? What do you know about pattern-string syntax? **[Model Question]**

Answer:

1st Part:

A Regular Expression (RE) is a string that represents a pattern. With RE functionality, ou can check any string with the pattern and see if any part of the string matches the pattern. The re module supplies Python's RE functionality. The compile function builds a RE object from a pattern string and optional flags. The methods of a RE object look for matches of the RE in a string or perform substitutions. Module re also exposes functions equivalent to a RE's methods, but with the RE's pattern string as the first argument.

2nd Part:

The pattern string representing a regular expression follows a specific syntax:

- Alphabetic and numeric characters stand for themselves. A RE whose pattern is a string of letters and digits matches the same string.

- Many alphanumeric characters acquire special meaning in a pattern when they are preceded by a backslash (\).
- Punctuation works the other way around: self-matching when escaped, special meaning when unescaped.
- The backslash character is matched by a repeated backslash (i.e., the pattern \\).

Since RE patterns often contain backslashes, you often specify them using raw-string syntax. Pattern elements (e.g., r'\t', equivalent to the non-raw string literal '\\t') do match the corresponding special characters (e.g., the tab character '\t'). Therefore, you can use raw-string syntax even when you do need a literal match for some such special character. Table 1 lists the special elements in RE pattern syntax. The exact meanings of some pattern elements change when you use optional flags, together with the pattern string, to build the RE object.

Table 1. RE pattern syntax:

Element	Meaning
.	Matches any character except \n (if DOTALL, also matches \n)
^	Matches start of string (if MULTILINE, also matches after \n)
$	Matches end of string (if MULTILINE, also matches before \n)
*	Matches zero or more cases of the previous RE; greedy (match as many as possible)
+	Matches one or more cases of the previous RE; greedy (match as many as possible)
?	Matches zero or one case of the previous RE; greedy (match one if possible)
*?, +?, ??	Nongreedy versions of *, +, and ? (match as few as possible)
{m, n}	Matches m to n cases of the previous RE (greedy)
{m, n}?	Matches m to n cases of the previous RE (nongreedy)
[…]	Matches any one of a set of characters contained within the brackets
|	Matches either preceding expression or following expression
(…)	Matches the RE within the parentheses and indicates a group
(?iLmsux)	Alternate way to set optional flags; no effect on match
(?:…)	Like (…), but does not indicate a group
(?P<id>…)	Like (…), but the group also gets the name id
(?P=id)	Matches whatever was previously matched by group named id
(?#...)	Content of parentheses is just a comment; no effect on match
(?=…)	Lookahead assertion: matches if RE… matches what comes next, but does not consume any part of the string
(?!…)	Negative lookahead assertion: matches if RE… does not match what comes next and does not consume any part of the string
(?<=…)	Lookbehind assertion: matches if there is a match ending at the current position for RE… (… must match a fixed length)
(?<!…)	Negative lookbehind assertion: matches if there is no match ending at

Element	Meaning
	the current position for RE... (... must match a fixed length)
\number	Matches whatever was previously matched by group numbered *number* (groups are automatically numbered from 1 to 99)
\A	Matches an empty string, but only at the start of the whole string
\b	Matches an empty string, but only at the start or end of a word (a maximal sequence of alphanumeric charactersl see also \w)
\B	Matches an empty string, but not at the start or end of a word
\d	Matches one digit, like the set [0-9]
\D	Matches one nondigit, like the set [^0-9]
\s	Matches a whitespace character, like the set [\t\n\r\f\v]
\S	Matches a nonwhitespace character, like the set [^\t\n\r\f\v]
\w	Matches one alphanumeric character; unless LOCALE or UNICODE is set, \w is like [a-zA-Z0-9_]
\W	Matches one nonalphanumeric character, the reverse of \w
\Z	Matches an empty string, but only at the end of the whole string
\\	Matches one backslash character

9. What are common Regular Expression idioms? **[Model Question]**

Answer:

`'.*'` as a substring of a regular expression's pattern string means "any number of repetitions (zero or more) of any character." In other words, `'.*'` matches any substring of a target string, including the empty substring. `'.+'` is similar, but matches only a nonempty substring. For example:

```
'pre.*post'
```

matches a string containing a substring 'pre' followed by a later substing 'post', even if the latter is adjacent to the former (e.g., it matches both 'prepost' and 'pre23post'). On the other hand:

```
'pre.+post'
```

Matches only if 'pre' and 'post' are not adjacent (e.g., it matches 'pre23post' but does nto match 'prepost'). Both patterns also match strings that continue after the 'post'. To constrain a pattern to match only strings that *end* with 'post', end the pattern with `\Z`. For example:

```
r'pre.*post\Z'
```

matches 'prepost', but not 'preposterous'. Note that you need to express the pattern with raw-string syntax (or escape the backslash \ by doubling it into \\), as it contains a backslash. Use raw-string syntax for all RE pattern literals, which ensures you'll never forget to escape a backslash.

Another frequently used element in RE patterns is \b, which matches a word boundary. If you want to match the word 'his' only as a whole word and not its occurrences as a substring in such words as 'this' and 'history', the RE pattern is:

```
r'\bhis\b'
```

with word boundaries both before and after. To match the beginning of any word starting with 'her', such as 'her' itself but also 'hermetic', but not words that just contain 'her' elsewhere, such as 'either' or 'there', use:

```
r'\bher'
```

with a word boundary before, but not after, the relevent string. To match the end of any word ending with 'its', such as 'its' itself but also 'fits', but not words that contain 'its' elsewhere, such as 'itsy' or 'jujitsu', use:

```
r'its\b'
```

with a word boundary after, but not before, the relevant string. To match whole words thus constrained, rather than just their beginning or end, add a pattern element `\w*` to match zero or more word characters. To match any full word starting with 'her', use:

```
r'\bher\w*'
```

to match any full word ending with 'its', use:

```
r'\w*its\b'
```

10. Describe Regular Expression object. [Model Question]

Answer:

A regular expression object *r* has the following read-only attrubutes that detail how *r* was built

`flags`

The `flags` argument passed to compile, or 0 when `flags` is omitted

`groupindex`

A dictionary whose keys are group names as defined by elements `(?P<id>)`; the corresponding values are the named groups' numbers

`pattern`

The pattern string from which *r* is compiled

These attributes make it easy to get back from a compiled RE object to its pattern string and flags, so you never have to store those separately.

A RE object *r* also supplies methods to locate matches for *r* within a string, as well as to perform substitutions on such matches.

`findall`

`r.findall(s)`

When *r* has no groups, `findall` returns a list of strings, each a substring of *s* that is a nonoverlapping match with *r*. For example, to print out all words in a file, one per line:

```
import re
reword=re.compile(r'\w+')
for aword in reword.findall(open('afile.txt').read()):
    print aword
```

When *r* has one group, `findall` also returns a list of strings, but each is the substring of *s* that matches *r*'s group. For example, to print only words that are followed by whitespace (not punctuation), you need to change only one statement in the example:

```
reword=re.compile('(\w+)\s')
```

When *r* has *n* groups (with $n > 1$), `findall` returns a list of tuples, one per nonoverlapping match with *r*. Each tuple has *n* items, one per group of *r*, the substring of

s matching the group. For example, to print the first and last word of each line that has at least two words:

```
import re
first_last=re.compile(r'^\W*(\w+)\b.*\b(\w+)\W*$', re.MULTILINE)
for first, last in \
first_last.findall(open('afile.txt').read()):
    print first, last
```

finditer

`r.finditer(s)`

`finditer` is like `findall` except that instead of a list of strings (or tuples), it returns an iterator whose items are match objects. In most cases, `finditer` is therefore more flexible and performs better than `findall`.

match

`r.match(s,start=0,end=sys.maxint)`

Returns an appropriate match object when a substring of *s*, string at index *start* and not reaching as far as index *end*, matches *r*. Otherwise, match returns None. Note that `match` is implicitly anchored at the starting position *start* in *s*. To search for a match with *r* at any point in *s* from *start* onward, call *r.search*, not *r.match*. For example, here's how to print all lines in a file that start with digits:

```
import re
digs=re.compile(r'\d+')
for line in open('afile.txt'):
if digs.match(line): print line,
```

search

`r.search(s,start=0,end=sys.maxint)`

Returns an appropriate match object for the leftmost substring of *s*, starting not before index *start* and not reaching as far as index *end*, that matches *r*. When no such substring exists, search returns `None`. For example, to print all lines containing digits, one simple approach is as follows:

```
import re
digs=re.compile(r'\d+')
for line in open('afile.txt'):
    if digs.search(line): print line,
```

split

`r.split(s,maxsplit=0)`

Returns a list *L* of the *splits* of *s* by *r* (i.e., the substrings of *s* separated by nonoverlapping, nonempty matches with *r*). For example, to eliminate all occurrences of substring 'hello' (in any mix of lowercase and uppercase) from a string, one way is:

```
import re
rehello=re.compile(r'hello', re.IGNORECASE)
astring=''.join(rehello.split(astring))
```

When *r* has *n* groups, *n* more items are interleaved in *L* between each pair of splits. Each of the *n* extra items is the substring of *s* that matches *r*'s corresponding group in that match, or `None` if that group did not participate in the match. For example here's one way to remove whitespace only when it occurs between a colon and a digit:

```
import re
re_col_ws_dig=re.compile(r'(:)\s+(\d)')
astring=''.join(re_col_ws_dig.split(astring))
```

If `maxsplit` is greater than 0, at most `maxsplit` splits are in *L*, each followed by *n* items as above, while the trailing substring of *s* after `maxsplit` matches of *r*, if any, is *L*'s last item. For example, to remove only the first occurrence of substring 'hello' rather than all of them, change the last statement in the first example above to:

```
astring=''.join(rehello.split(astring, 1))
```

sub

```
r.sub(repl,s,count=0)
```

Returns a copy of *s* where nonoverlapping matches with *r* are replaced by *repl*, which can be either a string or a callable object, such as a function. An empty match is replaced only when not adjacent to the previous match. when *count* is greater than 0, only the first *count* matches of *r* within *s* are replaced. When *count* equals 0, all matches of *r* within *s* are replaced. For example, here's another way to remove only the first occurrence of substring 'hello' in any mix of cases:

```
import re
rehello=re.compile(r'hello', re.IGNORECASE)
astring=rehello.sub('', astring, 1)
```

Without the final 1 argument to sub, the example removes all occurrences of 'hello'.

When *repl* is a callable object, *repl* must accept one argument (a match object) and return a string (or `None`, which is equivalent to returning the empty string '') to use as the replacement for the match. In this case, sub calls *repl*, with a suitable match-object argument, for each match with *r* that sub is replacing. For example,

to uppercase all occurrences of words starting with 'h' and ending with 'o' in any mix of cases:

```
import re
h_word=re.compile(r'\bh\w+o\b', re.IGNORECASE)
def up(mo): return mo.group(0).upper()
astring=h_word.sub(up, astring)
```

When *repl* is a string, sub uses *repl* itself as the replacement, except that it expands back references. A *back reference* is a substring of *repl* of the form `\g<id>`, where `id` is the name of a group in *r* (established by syntax `(?P<id>)` in *r*'s pattern string) or `\dd`, where `dd` is one or two digits taken as a group number. Each back reference, named or numbered, is replaced with the substring of *s* that matches the group of *r* that the back reference indicates. For example, here's a way to enclose every word in braces:

```
import re
grouped_word=re.compile('(\w+)')
astring=grouped_word.sub(r'{\1}', astring)
```

subn

`r.subn(repl,s,count=0)`

subn is the same as sub, except that subn returns a pair `(new_string, n)` where *n* is the number of substitutions that subn has performed. For example, one way to count the number of occurrences of substring 'hello' in any mix of cases is:

```
import re
rehello=re.compile(r'hello', re.IGNORECASE)
junk, count=rehello.subn('', astring)
print 'Found', count, 'occurrences of "hello"'
```

Unit 5: Frameworks

Unit at a glance:

5.1 Framework

The term framework can refer to a structure. It could be the structure of a system, a building, a project, or anything else.

You can now apply this definition in software engineering and refer to software frameworks as the structure for your software.

A software framework is a structure that you can use to build software. It acts as a foundation so you don't have to deal with creating unnecessary extra logic from scratch.

A framework is similar to a template in that you can modify it and add certain features and higher functionalities to create a complex and broad project that many people can use.

5.2 Why Use Software Frameworks?

Writing code is complex. And writing code that others can understand and manage is even more difficult because you must deal with syntax, declarations, performance, staying consistent, and other issues.

Software frameworks provide a template in which almost all general tasks have been handled. This allows you to focus on the core software development rather than the details of the process. Given that there is a set structure, it is simple to collaborate with others.

It's a good idea to use a software framework rather than re-inventing the wheel from scratch for numerous reasons. And perhaps the most important reason is that you won't have to write everything from scratch. This reduces the possibility of adding errors to your code.

Other reasons to use a framework include:

- It helps you avoid duplicate and redundant code.
- It makes it easier for developers who did not write the code to test and debug it.
- Frameworks are maintained by a group of people who test them so that you can use them with confidence.
- They help you write clean and secure code
- The time required to develop an application is significantly reduced because you can now focus on writing project-specific code.

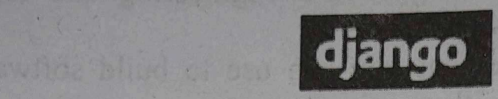
5.3 Web Development Frameworks

Web development involves creating a website that runs on the internet. This can be a single-page web application, a static website, a dynamic website, or anything else.

When creating these web pages, you usually have a frontend that the users see and a backend that controls and handles the logic (such as a database, API, and many other things).

5.4 What is Django?

Django was born in 2003 in a press agency of Lawrence, Kansas. It is a web framework that uses Python to create websites. Its goal is to write very fast dynamic websites. In 2005, the agency decided to publish the Django source code in the BSD license. In 2008, the Django Software Foundation was created to support and advance Django. Version 1.00 of the framework was released a few months later.



[ **Django's slogan**
The web framework for perfectionists with deadlines.]

Django's slogan is explicit. This framework was created to accelerate the development phase of a site, but not exclusively. Indeed, this framework uses the MVC pattern, which enables us to have a coherent architecture.

Until 2013, Django was only compatible with Python version 2x, but Django 1.5 released on February 26, 2013, points towards the beginning of Python 3 compatibility.
Today, big organizations such as the Instagram mobile website, Mozilla.org and Openstack.org are using Django.

5.5 Django – a web framework

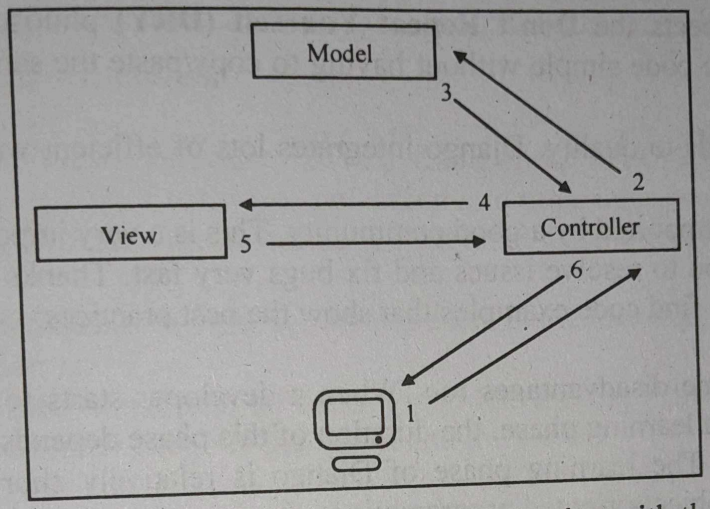
A framework is a set of software that organizes the architecture of an application and makes a developer's job easier. A framework can be adapted to different uses. It also gives practical tools to make a programmer's job faster. Thus, some features that are regularly used on a website can be automated, such as database administration and user management.
Once a programmer handles a framework, it greatly improves their productivity and the code quality.

5.6 The MVC framework

Before the MVC framework existed, web programming mixed the database access code and the main code of the page. This returned an HTML page to the user. Even if we are storing CSS and JavaScript files in external files, server-side language codes are stored in one file that is shared between at least three languages: Python, SQL and HTML.
The MVC pattern was created to separate logic from representation and have an internal architecture that is more tangible and real. The **Model-View=Controller (MVC)** represents the three application layers that the paradigm recommends:

Models: These represent data organization in a database. In simple words, we can say that each model defines a table in the database and the relations between other models. It's thanks to them that every bit of data is stored in the database.

- **Views:** These contain all the information that will be sent to the client. They make views that the final HTML document will generate. We can associate the HTML code with the views.
- **Controllers:** These contain all the actions performed by the server and are not visible to the client. The controller checks whether the user is authenticated or it can generate the HTML code from a template.



The following are the steps that are followed in an application with the MVC pattern:

1. The client sends a request to the server asking to display a page.
2. The controller uses a database through models. It can create, read, update, or delete any record or apply any logic to the retrieved data.
3. The model sends data from the database; for example, it sends a product list if we have an online shop.
4. The controller injects data into a view to generate it.
5. The view returns its content depending on the data given by the controller.
6. The controller returns the HTML content to the client.

The MVC pattern enables us to get coherence for each project's worker. In a web agency where there is a web designer and there are developers, the web designer is the head of the views. Given that views contain only the HTML code, the web designer will not be disturbed by the developer's code. Developers edit their models and controllers.

Django, in particular, uses an MVT pattern. In this pattern, views are replaced by templates and controllers are replaced by views. In the rest of this book, we will be using MVT patterns. Hence, our HTML code will be templates and our Python code will be views and models.

5.7 Why use Django

The following is a nonexhaustive list of the advantages of using Django:

- Django is published under the BSD license, which assures that web applications can be used and modified freely without any problems; it's also free.

- Django is full customizable. Developers can adapt to it easily by creating modules or overridden framework methods.
- This modularity adds other advantages. There are a lot of Django modules that you can integrate into Django. You can get some help with other people's work because you will often find high-quality modules that you might need.
- Using Python in this framework allows you to have benefits from all Python libraries and assures a very good readability.
- Django is a framework whose main goal is perfection. It was specifically made for people who want clear code and a good architecture for their applications. It totally respects the **Don't Repeat Yourself (DRY)** philosophy, which means keeping the code simple without having to copy/paste the same parts in multiple places.
- With regards to quality, Django integrates lots of efficient ways to perform unit tests.
- Django is supported by a good community. This is a very important asset because it allows you to resolve issues and fix bugs very fast. Thanks to the community, we can also find code examples that show the best practices.

Django has got some disadvantages too. When a developer starts to use a framework, he/she begins with a learning phase. the duration of this phase depends on the framework and the developer. The learning phase of Django is relatively short if the developer knows Python and object-oriented programming.

Short Answer Type Questions

A. Choose the correct answer from the given alternatives in each of the following:

1. DRY principle makes the code **[WBSCTE 2022]**

(a) Reusable
(b) Loop forever
(c) Bad and repetitive
(d) Complex

Answer: (a)

B. Fill in the blanks in the following statements:

2. Django supports the ***Model-View-Controller (MVC) design*** pattern. **[WBSCTE 2022]**

C. Answer the following questions:

3. Is Django better than Flask? **[WBSCTE 2022]**

Answer:

Flask provides support for API, while Django doesn't have any support for API. Flask does not support dynamic HTML pages, and Django offers dynamic HTML pages. Flask

is a Python web framework built for rapid development, whereas Django is built for easy and simple projects.

4. How do you insert a static file to a template in Django? [WBSCTE 2022]

Answer:

Static files can be added in either in the app directory with a directory called static or in the root with a directory called static.

Long Answer Type Questions

1. How Django is structured? [Model Question]

Answer:

Django is a Model-View-Controller (MVC) framework. MVC is a software design pattern that aims to separate a software application into three interconnected parts:

1. The **Model** provides the interface with the database containing the application data;
2. The **view** decides what information to present to the user and collects information from the user; and
3. The **controller** manages business logic for the application and acts as an information broker between the model and the view.

Django uses slightly different terminology in its implementation of MVC (Fig. 1). In Django:

1. The **model** is functionally the same. Django's Object-Relational Mapping (ORM more on the ORM later) provides the interface to the application database;
2. The **template** provides display logic and is the interface between the user and you Django application; and
3. The **view** manages the bulk of the application's data processing, application logic and messaging.

The MVC design pattern has been used for both desktop and web applications for many years, so there are variations on this theme – of which Django is no exception. If you wish to dig deeper into the MVC design pattern, be warned people can be passionate about what is a different interpretation of the same thing. To borrow a quote from the Django development team:

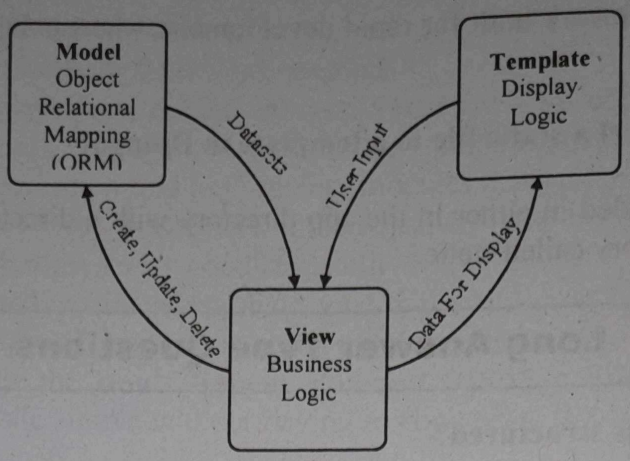


Fig: 1 A pictorial description of the Model-Template-View (MTV) pattern, Django's implementation of the MVC design pattern.

➲ 2. What is Django model? **[Model Question]**

Answer:

Django's models provide an Object-relational Mapping (ORM) to the underlying database. ORM is a powerful programming technique that makes working with data and relational databases much easier.

Most common databases are programmed with some form of Structured Query Language (SQL), however, each database implements SQL differently. SQL can be complicated and challenging to learn. An ORM tool, on the other hand, provides a simple mapping between an object (the 'O' in ORM) and the underlying database. This means the programmer need not know the database structure, nor does it require complex SQL to manipulate and retrieve data (Fig. 1).

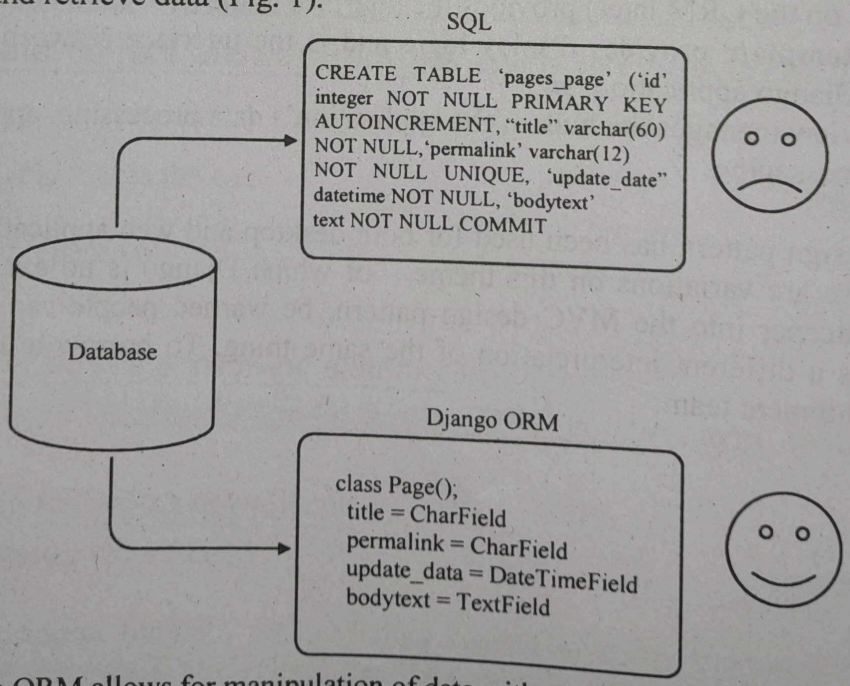


Fig: 1 An ORM allows for manipulation of data without having to write complex SQL

In Django, the model is the object mapped to the database. When you create a model, Django creates a corresponding table in the database (Fig. 2), without you having to write a single line of SQL. Django prefixes the table name with the name of your Django application (more on Django applications later).

The model also links related information in the database. In Fig. 3, a second model is created to keep track of a user's course enrolments. Repeating all the user's information in the `yourapp_Course` table would be against good design principles, so we instead create a *relationship* (the 'R' in ORM) between the `yourapp_Course` table and the `yourapp_UserProfile` table.

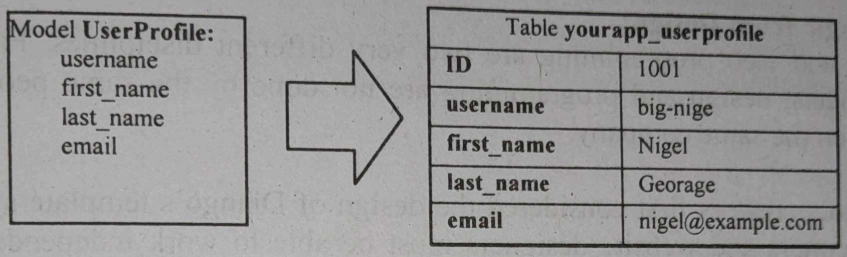


Fig: 2 Creating a Django model creates a corresponding table in the database

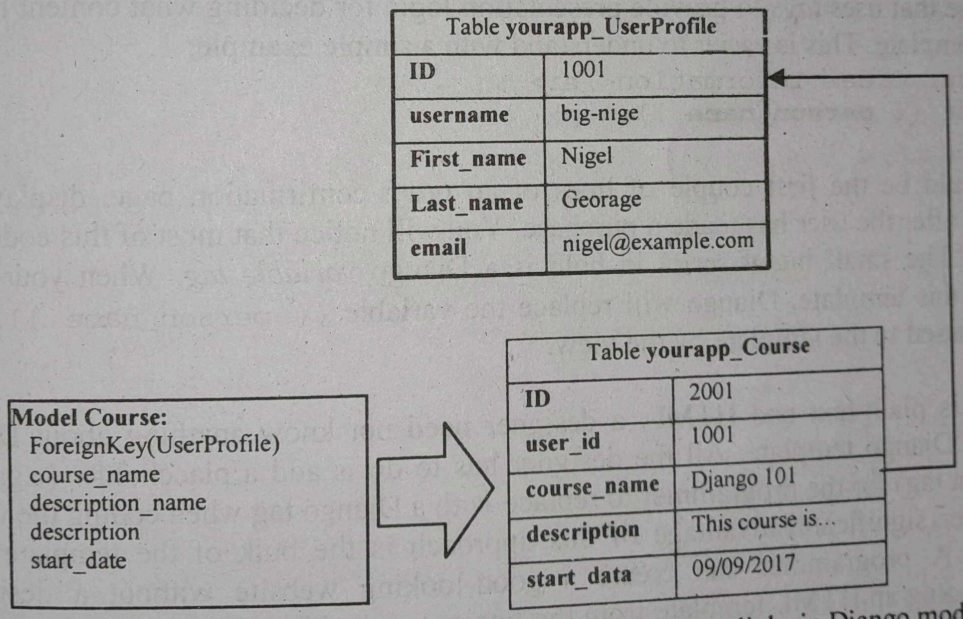


Fig: 3 Relationships between tables are created by foreign key links in Django models

This relationship is created by linking the models with a *foreign key* – i.e., the `user_id` field in the `yourapp_Course` table is a key field linked to the id field in the foreign table `yourapp_UserProfile`.

[Model Question]

➲ 3. Write about Django template?

Answer:

A Django template is a text file designed to separate an application's data from the way it is presented. Django's templates are not limited to HTML – they can be used for rendering several text formats.

The design of Django's templates is based on several core principles; however, three are key:

1. A template system should separate program logic from design;
2. Templates should discourage redundancy – Don't Repeat Yourself (DRY); and
3. The template system should be safe and secure – code execution in the template must be forbidden.

Separate Logic from Design:

Web design and web programming are two very different disciplines. For all but the smallest projects, design and programming are not done by the same people; in many cases, not even the same company.

When Django's creators first considered the design of Django's template system, it was clear programmers and website designers must be able to work independently of each other. The result is the Django Template Language (DTL) – a plain-text scripting language that uses *tags* to provide presentation logic for deciding what content to display in the template. This is easier to understand with a simple example:

```
<hi>Your Order Information</hi>
<p>Dear {{ person_name }},</p>
```

This could be the first couple of lines of an order confirmation page, displayed on a website after the user has made a purchase. You will notice that most of this code is plain HTML. The small bit of script in bold is a Django *variable tag*. When your browser renders this template, Django will replace the variable `{{ person_name }}` with the name passed to the template by the view.

As this is plain-text and HTML, a designer need not know anything about Django to create a Django template. All the designer has to do is add a placeholder (e.g., HTML comment tag) for the programmer to replace with a Django tag when coding the website. The other significant advantage of this approach is the bulk of the template is plain HTML. A programmer can create a good-looking website without a designer by downloading an HTML template from the Internet and adding Django template tags. This also works with Bootstrap templates and site templates heavy in JavaScript.

Don't Repeat Yourself (DRY):

DRY is a term that comes up often in Django discussions as it's one of Django's core principles. The DRY principle is particularly evident in how Django user *template inheritance*. To better understand how template inheritance helps minimize repetition and redundant code, let's first examine a typical webpage layout (Fig. 1).

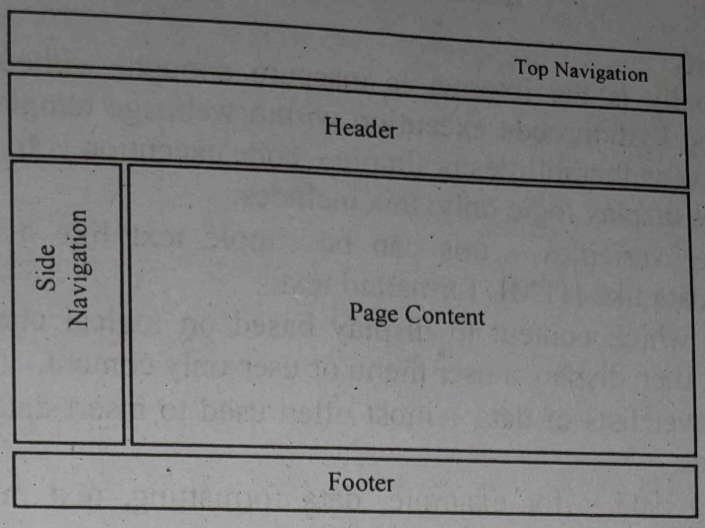


Fig: 1 A typical webpage layout with common elements such as a header, footer and navigation

This page layout has top navigation, a header image, left side navigation, the main content of the page and a footer. If you only wanted to create a few webpages, you could get away with copying the front page and simply changing the content and saving each different page as an HTML file.

The problem is, not only are we repeating a lot of code but maintaining a large site could quickly get out of hand – what if you needed to change the template? You would have to make the change on every single page in your site!

We fix this problem by creating a parent template that contains content common to the entire website and then creating child templates that inherit these common features, adding any content unique to the child template (Fig. 2).

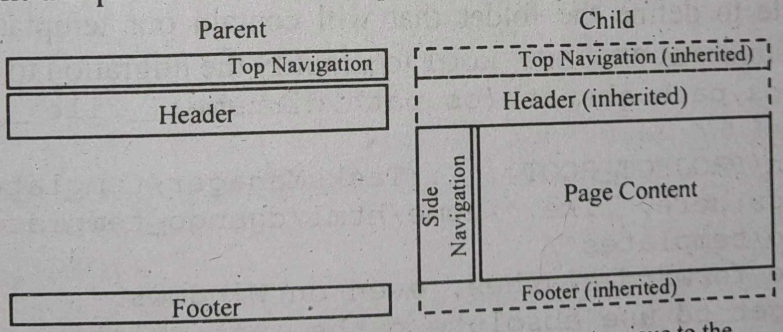


Fig: 2 A child template only adds structure and content unique to the child. All common elements are inherited from the parent template

You will notice I included the sidebar navigation in the child here. It's common for certain page on a site to have limited navigation, so not every page will need the side navigation.

Django supports multiple inheritance too. Following on from the above example, you could have a child template that adds only the side navigation to the parent and then have a third template that inherits from the child and adds the content.

The only limit to how you slice and dice Django's template inheritance is practically – if you have templates inheriting more than one or two deep, you should re-evaluate your site design.

Template Security:
Django's philosophy is the Internet is insecure enough, without introducing security issues by allowing Python code execution within webpage templates. Django's solution to template security vulnerabilities is simple – code execution is forbidden.
The DTL provides display logic only, this includes:

- Displaying variables – this can be simple text like a user's name or more complex data like HTML formatted text.
- Choosing which content to display based on logical checks. e.g., if a user is logged in, then display a user menu or user-only content.
- Iterating over lists of data – most often used to insert database information into HTML lists.
- Formatting data – for example, data formatting, text manipulation and other filters that act on the data.

Things you can't do in a Django template:

- Execute Python code
- Assign a value to a variable
- Perform advanced logic

Django's templates also add additional security features like automatically escaping all strings, *Cross-Site Scripting* protection and *Cross-Site Request Forgery* protection.

➲ 4. Give an example to display 'hello world' in a template. **[Model Question]**

Answer:
We will create the first template of our application. To do so, we must first edit the `settings.py` file to define the folder that will contain our templates. We will first define the project folder as `PROJECT_ROOT` to simplify the migration to another system:

```
PROJECT_ROOT=os.path.abspath(os.path.dirname{__file__})
TEMPLATE_DIRS={
  os.path.join(PROJECT_ROOT, '../TasksManager/templates'),
  # Put strings here, like "/home/html/dgango_templates" or
"C:/www/Django/templates"/
  # Always use forward slashes, even on Windows.
  # Don't forget to use absolute paths, not relative paths.
}
```

Now that Django knows where to look for the templates, we will create the first template of the application. To do this, use a file browser and add the `index.html` file in the `TasksManager/templates/en/public/` folder. We do not need to create the `__init__.py` file, because these files do not contain any Python files.
The following is the content of the `index.html` file:

```
<html>
  <head>
    <title>
      Hello World Title
```

```
    </title>
  </head>
  <body>
    <hi>
      Hello World Django
    </hi>
    <article>
      Hello world!
    </article>
  </body>
</html>
```

Although the template is correct, we need to change the view to indicate its uses. We will modify the `index.py` file with the following content:

```
from django.shortcuts import render
# View for index page.
def page(request):
  return render(request, 'en/public/index.html')
```

If we test this page, we will notice that the template has been taken into account by the view.

➲ 5. How to use static file in Django template? [Model Question]

Answer:

Static files such as JavaScript files, CSS, or images are essential to obtain an ergonomic website. These files are often stored in a folder, but they can be useful to modify this folder under development or in production.

According to the URLs, Django allows us to define a folder containing the static files and to easily modify its location when required.

To set the path where Django will look for static files, we have to change our `settings.py` file by adding or changing the following line:

```
STATIC_URL= '/static/'
STATICFILES_DIRS={
    os.path.join(PROJECT_ROOT, '../TasksManager/static/'),
```

We will define a proper architecture for our future static files. It is important to choose an early consistent architecture, as it makes the application support as well as include another developer easier. Our statics files' architecture is as follows:

```
static/
  images/
  javascript/
    lib/
  css/
  pdf/
```

We create a folder for each type of static file and define a `lib` folder for `JavaScript` libraries as jQuery, which we will use later in the book. For example, we change our `base.html file`. We will add a CSS file to manage the styles of our pages. To do this, we must add the following line between `</ title>` and `< / head>`:

```
<link href="{% static "css/style.css" %}" rel="stylesheet"
type="text/css" />
```

To use the tag in our static template, we must also load the system by putting the following line before using the static tag:

```
{% load staticfiles %}
```

We will create the `style.css` file in the `/static/css` folder. This way, the browser won't generate an error later in the development.

➲ 6. Give an example how to inject the data from the view to template.

[Model Question]

Answer:

Before improving our template, we must send variables to the templates. The injection of the data is based on these variables, as the template will perform certain actions. Indeed, as we have seen in the explanation of the MVC pattern, the controller must send variables to the template in order to display them.

There are several functions to send variables to the template. The two main functions are `render()` and `render_to_response()`. The `render()` function is very similar to `render_to_response()`. The main difference is that if we use render, we do not need to specify `context_instance=RequestContext(request)` in order to send the current context. This is the context that will allow us to use the CSRF middleware later in the book.

We will change our view to inject variables in our template. These variables will be useful to work with the template language. The following is our modified view:

```
from django.shortcuts import render
"""
View for index page.
"""
def page(request):
  my_variable="Hello World!"
  years_old=15
  array_city_capitale=[ "Paris", "London", "Washington" ]
  return render (request, 'en/public/index.html', {
"my_var":my_variable, "years":years_old,
"array_city":city_capitale })
```

➲ 7. How to create dynamic template ? How to integrate variables in template?

[Model Question]

Answer:

1st Part:

Django comes with a full-template language. This means that we will use template tags that will allow us to have more flexibility in our templates and display variables, perform loops and set up filters.

The HTML and template languages are mixed in the templates; however, the template language is very simplistic and there is a minority when compared to the HTML code. A web designer will easily modify the template files.

2nd Part:

In our controller, we send a variable named `my_var`. We can display it in a `<span>` tag in the following way. Add the following lines in the `<article>` tag of our template tag:

```
<span> {{my_var}} </ span>
```

In this way, because our variable contains `string="Hello World!"`, the HTML code that will be generated is as follows:

```
<span> Hello World! </span>
```

➲ 8. How to implement looping in template? [Model Question]

Answer:

Looping allows you to read through the elements of a table or data dictionary. In our controller, we sent a data table called `array_city` in which we have the names of cities. To see all these names of cities in the form of a list, we can write the following in our template:

```
<ul>
  {% for city in array_city %}
    <li>
      {{ city }}
    </li>
  {% endfor %}
</ul>
```

This looping will go through the `array_city` table and place each element in the `city` variable that we display in the `<li>` tag. With our sample data, this code will produce the following HTML code:

```
<ul>
  <li>Paris</li>
  <li>London</li>
  <li>Washington</li>
</ul>
```

➲ 9. Write a short note on Django form. [Model Question]

Answer:

Django provides a Form class which is used to create HTML forms. It describes a form and how it works and appears.

It is similar to the **ModelForm** class that creates a form by using the Model, but it does not require the Model.

Each field of the form class map to the HTML form **<input>** element and each one is a class itself, it manages form data and performs validation while submitting the form.
Lets see an example, in which we are creating some fields too.

```
from django import forms
class StudentForm(forms.Form):
  firstname=forms.CharField(label="Enter first name",max_length=50)
  lastname=forms.CharField(label="Enter last name", max_length=100)
```

A StudentForm is created that contains two fields of CharField type. Charfield is a class and used to create an HTML text input component in the form.
The label is used to set HTML label of the component and max_length sets length of an input value.

When rendered, it produces the following HTML to the browser.

```
<label for="id_firstname">Enter first name:</label>
<input type="text" name="firstname" required maxlength="50" id="i
d_firstname" />
<label for="id_lastname">Enter last name:</label> <input type="te
xt" name="lastname" required
maxlength="100" id="id_lastname" />
```

Commonly used fields and their details are given in the below table.

Name	Class	HTML Input	Empty value
BooleanField	class BooleanField(**kwargs)	CheckboxInput	False
CharField	class CharField(**kwargs)	TextInput	Whatever you've given as empty_value.
ChoiceField	class ChoiceField(**kwargs)	Select	" (an empty string)
DateField	class DateField(**kwargs)	DateInput	None
DateTimeField	class DateTimeField(**kwargs)	DateTimeInput	None
DecimalField	class DecimalField(**kwargs)	NumberInput	None
EmailField	class EmailField(**kwargs)	EmailInput	" (an empty string)
FileField	class FileField(**kwargs)	ClearableFileInput	None
ImageField	class ImageField(**kwargs)	ClearableFileInput	None

Let's see a complete example to create an HTML form with the help of Django Form class.

Building a Form in Django

Suppose we want to create a form to get Student information, use the following code.

```
from django import forms
class StudentForm(forms.Form):
   firstname=forms.CharField(label="Enter first name",max_length=50)
   lastname=forms.CharField(label="Enter last name", max_length=100)
```

Put this code into the **forms.py** file.

Instantiating Form in Django

Now, we need to instantiate the form in **views.py** file. See, the below code.

```
// views.py
from django.shortcuts import render
from myapp.form import StudentForm

def index(request):
    student = StudentForm()
    return render(request,"index.html",{'form':student})
```

Passing the context of form into index template that looks like this:

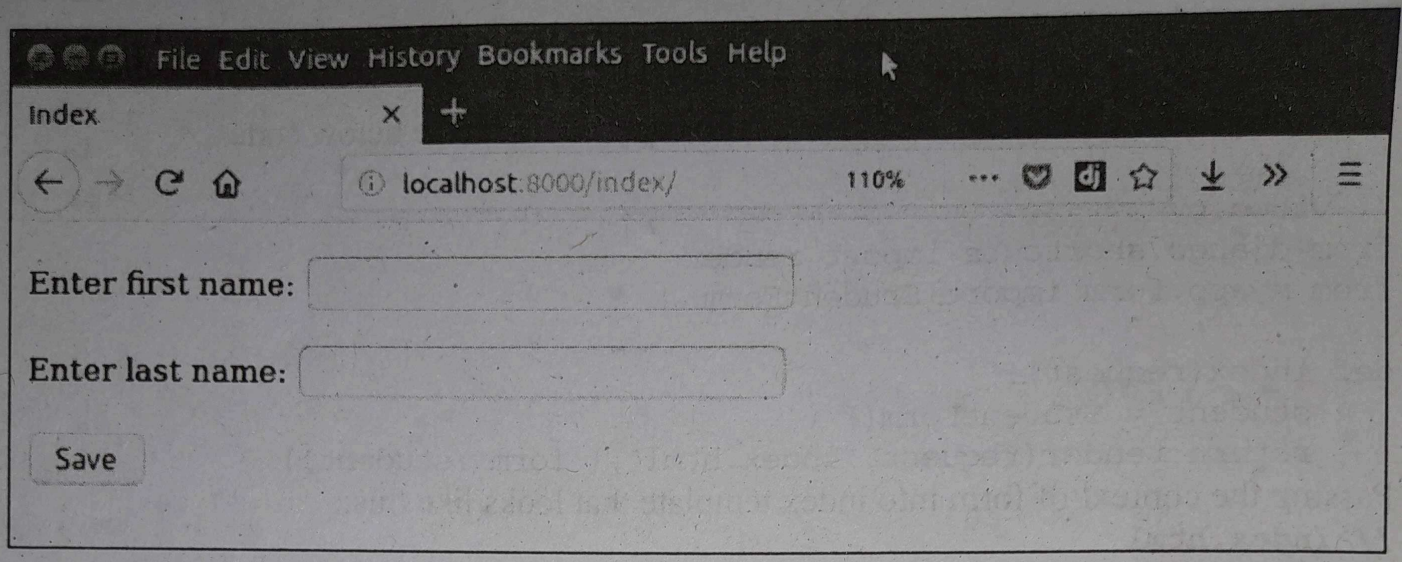
// index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Index</title>
</head>
<body>
<form method="POST" class="post-form">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-
default">Save</button>
</form>
</body>
</html>
```

Provide the URL in urls.py

```
from django.contrib import admin
from django.urls import path
from myapp import views
urlpatterns = [
   path('admin/', admin.site.urls),
   path('index/', views.index),
]
```

Run Server and access the form at browser by **localhost:8000/index,** and it will produce the following output.



There are other output options though for the `<label>/<input>` pairs:

- `{{ form.as_table }}` will render them as table cells wrapped in `<tr>` tags
- `{{ form.as_p }}` will render them wrapped in `<p>` tags
- `{{ form.as_ul }}` will render them wrapped in `<li>` tags