Syllabus ...

Unit - I Introduction and Control Flow Statements in Python

1.1 Features of Python - Interactive, object oriented, interpreted, platform independent.

1.2 Python building blocks - Identifiers, keywords, indentation, variables, comments.

1.3 Python data types: Numbers, string, tuples, lists, dictionary.

1.4 Basic operators: Arithmetic, comparison/relational, assignment, logical, bitwise, membership, identity operators, Python operator precedence.

1.5 Control flow: Conditional statements (if, if else, nested if), loopingin python (while loop, for loop, nested loops) loop manipulation using continue, pass, break, else.

Unit - II Python Specific Data Structures and Functions

2.1 Lists: Defining lists, accessing values in list, deleting values inlist, updating lists, basic list operations, built-in list functions.

2.2 Tuples: Accessing values in tuples, deleting values in tuples, and updating tuples, basic tuple operations, built-in tuple functions.

2.3 Sets: Accessing values in set, deleting values in set and updating sets, basic set operations, built-in set functions.

2.4 Dictionaries: Accessing values in dictionary, deleting values in dictionary and updating dictionary, basic dictionary operations, built-in dictionaries functions.

2.5 Use of Python built-in functions (e.g. type/data conversion functions, math functions etc.)

2.6 User defined functions: Function definition, function calling, function arguments and parameter passing, return statement, scope of variables: global variable and local variable.

Unit - III Python Modules and Packages

3.1 Modules: Writing modules, importing modules, importing objects from modules, Python built-in modules (e.g. numeric and mathematical module, functional programming module).

3.2 Python packages: Introduction, writing Python packages.

3.3 Using standard NumPy: Methods in NumPy, creating arrays and initializing, reading arrays from files, special initializing functions, slicing and indexing, reshaping arrays, combining arrays, NumPy maths.

Unit - IV Object Oriented Programming in Python

4.1 Introduction to object oriented programming, creating classes and objects, constructor and destructor in Python.

4.2 Data abstraction and data encapsulation.

4.3 Concept of polymorphism - method overloading and overriding.

4.4 Inheritance and types of inheritance.

Unit - V Linear Data Structure Arrays, Link List, Stack and Queues using Python

5.1 Data Structures - Definition, linear data structures, non-liner data structures arrays - overview, types of arrays, operations on arrays, arrays vs list.

5.2 Searching - Linear search and binary search, sorting - bubble sort, insertion sort.

5.3 Linked Lists - Singly linked list, doubly linked list, circular linked lists, implementation using Python packages for link list.

5.4 Stacks: Introduction to stacks, stack applications - expression evaluation, backtracking, traversal - infix, prefix and postfix concepts.

5.5 Queues: Introduction to queues, queue applications - breadth first search, depth first search.

Unit - VI Non-Linear Data Structure

6.1 Trees - Tree Terminology, binary trees: Implementation, tree traversals, binary search trees.

6.2 Applications of trees - Spanning, tree, BST, tree traversal - in order, preorder and postorder concepts.

Contents ...

Chapter 1...

Introduction and Control Flow Statements in Python

Weightage of Marks = 12, Teaching Hours = 06

Syllabus

1.1 Features of Python - Interactive, object oriented, interpreted, platform independent.

1.2 Python building blocks - Identifiers, keywords, indentation, variables, comments.

1.3 Python data types: Numbers, string, tuples, lists, dictionary.

1.4 Basic operators: Arithmetic, comparison/relational, assignment, logical, bitwise, membership, identity operators, Python operator precedence.

1.5 Control flow: Conditional statements (if, if else, nested if), looping in python (while loop, for loop, nested loops) loop manipulation using continue, pass, break, else.

Objectives

At the end of this chapter students will be able to:

- Describe the given variables, keywords and constants in Python.
- Use indentation, comments in the given program.
- Use different types of operators for writing arithmetic expressions.
- Write Python programs using control flow.

1.1 FEATURES OF PYTHON - INTERACTIVE, OBJECT - ORIENTED, INTERPRETED, PLATFORM INDEPENDENT

1.1.1 Python

Python is a high-level, general-purpose programming language that was created by Guido van Rossum and first released in 1991. It is known for its simplicity, readability, and versatility, making it one of the most popular programming languages in the world.

Python emphasizes code readability and follows a clean and elegant syntax, which makes it easier to write and understand. It uses indentation to define code blocks rather than using braces or keywords, promoting a consistent and readable style.

Python is widely used in various domains such as web development, scientific computing, data analysis, machine learning, artificial intelligence, and automation. Its versatility, coupled with a large and active community, has made it a preferred choice for both beginners and experienced developers.

1.1.2 Features of Python

- **Easy-to-learn:** Python has few keywords, simple structure, and a clearly defined syntax.
- **Easy-to-read:** Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain:** Python's source code is fairly easy-to-maintain.
- **Object-Oriented Approach:** One of the key aspects of Python is its object-oriented approach. This basically means that Python recognizes the concept of class and object encapsulation thus allowing programs to be efficient in the long run.
- **A broad standard library:** Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
- **Interactive Mode:** Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.

- **Portable:** Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
- **Extendable:** You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
- **Databases:** Python provides interfaces to all major commercial databases.
- **GUI Programming:** Python supports GUI applications that can be created and ported to many system calls, libraries, and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
- **Scalable:** Python provides a better structure and support for large programs than shell scripting.

1.1.3 Python Interpreter

- The interpreter is a program code that reads the code and executes it one-by-one. It converts the code into a language that is compatible with computer hardware. It directly executes the instructions whether it is written in a programming language or scripting language.
- The Python interpreter is a virtual machine, meaning that it is software that emulates a physical computer.
- This virtual machine is a stack machine, it manipulates several stacks to perform its operations (as contrasted with a register machine, which writes to and reads from memory locations).

Working of Interpreter:

- It breaks the line of code in tokens.
- The structure is generated. Moreover, it depicts the relationship between tokens.
- Now, it turns into object code(s).
- Finally, the object code will execute.

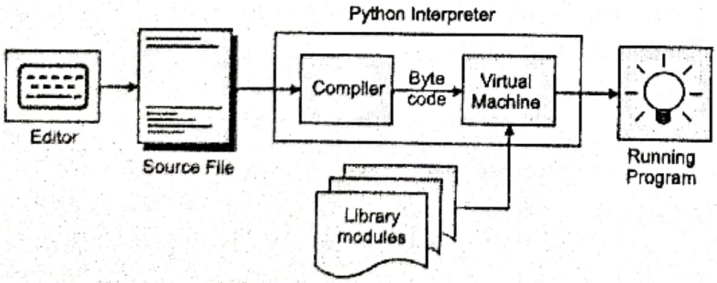


Fig. 1.1: How Python Interpreter Work

1.2 PYTHON BUILDING BLOCKS - IDENTIFIERS, KEYWORDS, INDENTION, VARIABLES COMMENTS

1.2.1 Identifiers

- Identifier is a name used to identify a variable, function, class, module etc.
- The identifier is a combination of character digits and underscore.
- The identifier should start with a character or Underscore then use a digit.
- The characters are A-Z or a-z, an Underscore (_) , and digit (0-9). We should not use special (#, @, $, %, !) in identifiers.

Examples of valid identifiers:

- var1
- _var1
- _1_var
- var_1

Examples of invalid identifiers:

- !var1
- 1var
- 1_var
- var#1

1.2.2 Python Keywords

Python keywords are reserved words that have special meanings and purposes in the Python programming language. These keywords cannot be used as variable names or identifiers because they are part of the language syntax. There are 33 keywords in Python 3.7, and there are 35 keywords in Python 3.11.

Here is a list of Python keywords along with their descriptions.

1. **False:** Represents the Boolean value "false" (0 in numeric context).
2. **None:** Represents the absence of a value or a null value.
3. **True:** Represents the Boolean value "true" (1 in numeric context).
4. **and:** A logical operator used to perform a logical "and" operation between two conditions.
5. **as:** Used to create an alias while importing a module or to give a different name to an imported module or symbol.
6. **assert:** Used for debugging purposes to check if a condition is true, and if not, raises an exception.
7. **break:** Used to exit from a loop prematurely.
8. **class:** Used to define a class, which is a blueprint for creating objects.
9. **continue:** Used to skip the rest of the current iteration in a loop and move to the next iteration.
10. **def:** Used to define a function or method.

11. **del:** Used to delete a reference to an object or element from a collection.
12. **elif:** Short for "else if," it is used in conditional statements to specify an alternative condition to be tested.
13. **else:** Used in conditional statements to define a block of code that should be executed if the preceding conditions are not met.
14. **except:** Used in exception handling to define a block of code that should be executed when a specific exception occurs.
15. **finally:** Used in exception handling to define a block of code that should be executed regardless of whether an exception occurred or not.
16. **for:** Used to create a loop that iterates over a sequence of elements.
17. **from:** Used to import specific attributes or functions from a module.
18. **global:** Used to indicate that a variable is a global variable, meaning it can be accessed and modified from anywhere in the program.
19. **if:** Used to perform conditional execution of code based on a condition.
20. **import:** Used to import a module into the current program.
21. **in:** Used to check if a value is present in a sequence.
22. **is:** Used to test if two objects are identical.
23. **lambda:** Used to create anonymous functions (functions without a name).
24. **non-local:** Used to indicate that a variable refers to the nearest enclosing scope, rather than the global or local scope.
25. **not:** A logical operator used to perform a logical negation operation on a condition.
26. **or:** A logical operator used to perform a logical "or" operation between two conditions.
27. **pass:** Used as a placeholder when no action is required in a block of code.
28. **raise:** Used to raise an exception.
29. **return:** Used to exit a function and return a value.
30. **try:** Used to specify a block of code to be tested for exceptions.
31. **while:** Used to create a loop that continues as long as a certain condition is true.
32. **with:** Used to define a block of code with a context manager, providing a clean way of working with resources.
33. **yield:** Used in generator functions to return a value and temporarily suspend the function's execution.
34. **async:** The async keyword is used with def to define an asynchronous function, or coroutine. The syntax is just like defining a function, with the addition of async at the beginning.
35. **Await:** Python's await keyword is used in asynchronous functions to specify a point in the function where control is given back to the event loop for other functions to run.

1.2.2.1 Difference between Keywords and Identifier

Table 1.1

Kewwords	Identifiers
Keywords are reserved words with special meaning.	Identifier is a unique name given to the class, function, array and so on.
Keywords do not have symbols.	Identifiers can have symbols.
Specify the type / kind of entity.	Identify the name of a particular entity
Specify the type / kind of entity.	Identifiers are classified into 'external name' and 'internal name'.
Keywords are not further classified.	Except underscore (_) no symbols or punctuation is used.
Example: class, while, in and so on.	**Example:** var. a, _newstr, new_var, and so on

1.2.3 Python Indention

Indentation refers to the spaces at the beginning of a code line. Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important. Python uses indentation to indicate a block of code, whereas, in C programming uses '{}'for indentation to indicate a block of code.

Example 1.1:

Output:

Example 1.2:

```
if 5 > 2:
print("Five is greater than two!")
```

Output:

```
    print("Five is greater than two!")
IndentationError: expected an indented block after 'if' statement on line 1
```

// Python will give you an error if you skip the indentation:

A comparison of C & Python program:

In C programming	In Python programming

In C
#include<stdio.h>
int main ()
{ //Beginning of Block 1
int a;
printf("the easiest programming language: ");
scanf("%d", &a);
if (a = "python")
{ // Beginning Of Code Block 2
printf("Yes! You are right");
} //End of Code Block 2
else
{ //Beginning of Code Block 3
printf("Nope! You are wrong");
//End of Code Block 3
return 0;
} //End of Block 1
	a = input('Enter the easiest programming language:")
	if a == 'python':
	print('Yes! You are right')
	else:
	print('Nope! You are wrong')
	""""
	Additional space before print is used to indicate a new block of code. Here, both the print are indented 2 spaces
	""""

1.2.4 Variables

- Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.

Example 1.3:

```
x = 5
y = "John"
print(x)
print(y)
```

//Output:

```
5
John
```

Variables do not need to be declared with any particular type and can even change type after they have been set.

Example 1.4:

```
x = 4      # x is of type int
x = "Sally" # x is now of type str
print(x)
```

//Output:

```
Sally
```

Table 1.3

Python Variables	Data Types generated internally
x = 45	Type Integer
name = "Python"	Type = String
nums = [1, 2, 3, 4]	Type = Lists

1.2.5 Comment

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.

1.2.3 Single Line Comments

Comments starts with a #, and Python will ignore them.

Example 1.4:

```
#This is a comment
print("Hello, World!")
```

Output

```
Hello, World!
```

1.2.4 Multi Line Comments

To add a multiline comment you could insert triple quotes """ in your code, and place your comment inside it:

Example 1.5:

```
"""
This is a comment written in more than just one line
"""
print("Hello, World!")
```

1.3 PYTHON DATA TYPES: NUMBERS, STRING, LISTS, TUPLES, DICTIONARY

1.3.1 Built-In Numeric Datatypes in Python

In Python, the 4 built-in numeric data types are:

- **int**: These are whole numbers of unlimited range. ...
- **long**: These are long integers in Python 2. ...
- **float**: These are floating point numbers represented as 64-bits double precision numbers. ...
- **complex**: Are unsigned numbers with real and imaginary components.

1.3.2 Python Data Types

Built-in Data Types:

- In programming, data type is an important concept.
- Variables can store data of different types, and different types can do different things.
- Python has the following **datatypes built-in** by default, in these categories:
  - Text Type: str
  - Numeric Types: int, long, float, complex
  - Sequence Types: list, tuple, range
  - Mapping Type: dict
  - Set Types: set, frozenset
  - Boolean Type: bool
  - Binary Types: bytes, byte array, memory view
  - None Type: NoneType

1.3.3 Python Numbers

There are three numeric types in Python:

- int
- float
- complex

Variables of numeric types are created when you assign a value to them:

Example 1.6:

```
x = 1          # int
y = 2.8        # float
z = 1j         # complex
```

1.3.4 Strings in Python

A string is a sequence of characters. It can be declared in python by using double quotes. Strings are immutable, i.e., they cannot be changed.

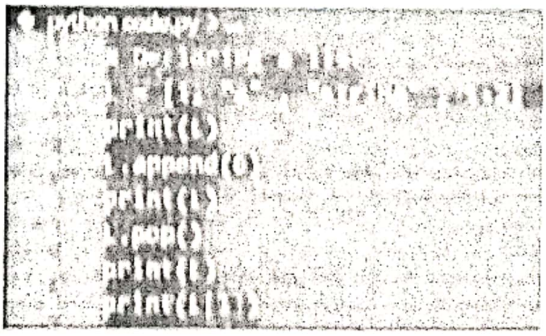
```
# Assigning string to a variable
a = "This is a string"
print (a)
```

Output:

```
This is a string
```

1.3.5 Lists in Python

- Lists are one of the most powerful tools in python.
- They are just like the arrays declared in other languages.
- But the most powerful thing is that list need not be always homogeneous.
- A single list can contain strings, integers, as well as objects.
- Lists can also be used for implementing stacks and queues.
- Lists are mutable, i.e., they can be altered once declared.

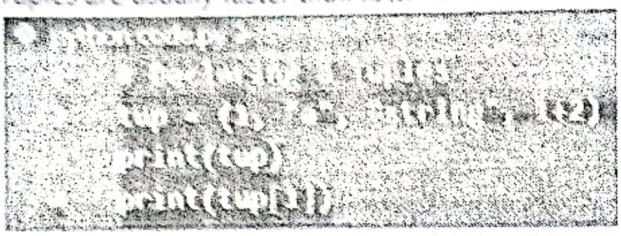


Output:

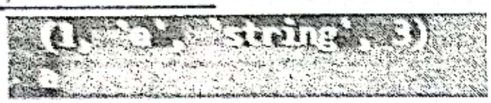
1.3.6 Tuples in Python

- A tuple is a sequence of immutable Python objects.

- Tuples are just like lists with the exception that tuples cannot be changed once declared.
- Tuples are usually faster than lists.

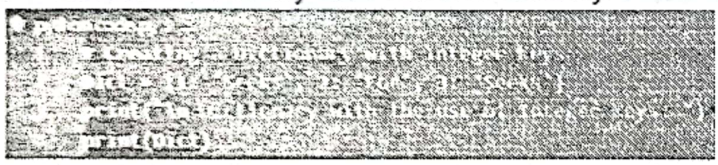


Output:

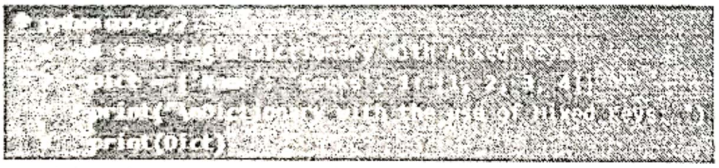


1.3.7 Python Dictionary

- **Dictionary** in Python is an unordered collection of data values, used to store data values like a map; which, unlike other data types that hold only a single value as an element, Dictionary holds **key:value** pair.
- Key-Value is provided in the dictionary to make it more optimized.
- Dictionary keys are case sensitive, the same name but different cases of Key will be treated distinctly.



Output:



Output:

1.4 BASIC OPERATORS: ARITHMETIC, COMPARISON/ASSIGNMENT, IDENTITY OPERATORS, MEMBERSHIP, PYTHON OPERATOR PRECEDENCE, TERNARY OPERATOR

1.4.1 Arithmetic Operators

Arithmetic operators are used to performing mathematical operations like addition, subtraction, multiplication, and division.

Table 1.4

Operator	Description	Syntax
+	Addition: adds two operands	x + y
-	Subtraction: subtracts two operands	x - y
*	Multiplication: multiplies two operands	x * y
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	x // y
%	Modulus: returns the remainder when the first operand is divided by the second	x % y
**	Power: Returns first raised to power second	x ** y

Example 1.6: Arithmetic operators in Python

```
# Examples of Arithmetic Operator
a = 9
b = 4
# Addition of numbers
add = a + b# Output 13
# Subtraction of numbers
sub = a - b# Output5
# Multiplication of number
mul = a * b# Output36
# Division(float) of number
div1 = a / b# Output2.25
# Division(floor) of number
div2 = a // b# Output2
# Modulo of both number
mod = a % b# Output1
# Power
p = a ** b  # Output6561
# print results
print(add)
print(sub)
print(mul)
print(div1)
print(div2)
print(mod)
print(p)
```

1.4.2 Comparison of Relational Operators

- Comparison of Relational operators compares the values.
- It either returns **True** or **False** according to the condition.

Table 1.5

Operator	Description	Syntax
>	Greater than: True if the left operand is greater than the right	x > y
<	Less than: True if the left operand is less than the right	x < y
==	Equal to: True if both operands are equal	x == y
!=	Not equal to – True if operands are not equal	x != y
>=	Greater than or equal to True if the left operand is greater than or equal to the right	x >= y
<=	Less than or equal to True if the left operand is less than or equal to the right	x <= y

Example 1.7: Comparison Operators in Python

```
# Examples of Relational Operators in python
a =13
b =33
# a > b is False
print(a > b)
# a < b is True
print(a < b)
# a == b is False
print(a ==b)
# a != b is True
print(a !=b)
# a >= b is False
print(a >=b)
# a <= b is True
print(a <=b)
```

Output:

```
False
True
False
True
False
True
```

1.4.3 Logical Operators

Logical operators perform **Logical AND**, **Logical OR** and **Logical NOT** operations. It is used to combine conditional statements.

Operator	Description	Syntax
and	**Logical AND:** True if both the operands are true	x and y
or	**Logical OR:** True if either of the operands is true	x or y
not	**Logical NOT:** True if the operand is false	not x

Example 1.9: Logical Operators in Python

```
# Examples of Logical Operator
a =True
b =False

# Print a and b is False
print(a and b)

# Print a or b is True
print(a or b)

# Print not a is False
print(nota)
```

Output:

```
False
True
False
```

1.4.4 Bitwise Operators

Bitwise operators act on bits and perform the bit-by-bit operations. These are used to operate on binary numbers.

Table 1.7

Operator	Description	Syntax
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

Example 1.10: Bitwise Operators in Python

```
a =10
b =4
# Print bitwise AND operation
```

```
print(a & b) # 0
# Print bitwise OR operation
print(a | b) # 14
# Print bitwise NOT operation
print(-a) # -11
# print bitwise XOR operation
print(a ^ b) # 14
# print bitwise right shift operation
print(a >> 2) # 2
# print bitwise left shift operation
print(a << 2) # 40
```

Output:

```
0
14
-11
14
2
40
```

1.4.5 Assignment Operators

Assignment operators are used to assigning values to the variables.

Table 1.8

Operator	Description	Syntax
=	Assign value of right side of expression to left side operand	x = y + z
+=	**Add AND:** Add right-side operand with left side operand and then assign to left operand	a+=b a=a+b
-=	**Subtract AND:** Subtract right operand from left operand and then assign to left operand	a-=b a=a-b
*=	**Multiply AND:** Multiply right operand with left operand and then assign to left operand	a*=b a=a*b
/=	**Divide AND:** Divide left operand with right operand and then assign to left operand	a/=b a=a/b
%=	**Modulus AND:** Takes modulus using left and right operands and assign the result to left operand	a%=b a=a%b
//=	**Divide(floor) AND:** Divide left operand with right operand and then assign the value(floor) to left operand	a//=b a=a//b
**=	**Exponent AND:** Calculate exponent (raise power) value using operands and assign value to left operand	a=a**b
&=	Performs Bitwise AND on operands and assign value to left operand	a&=b a=a&b
|=	Performs Bitwise OR on operands and assign value to left operand	a|=b a=a|b
^=	Performs Bitwise xOR on operands and assign value to left operand	a^=b a=a^b
>>=	Performs Bitwise right shift on operands and assign value to left operand	a>>=b a=a>>b
<<=	Performs Bitwise left shift on operands and assign value to left operand	a <<= b a= a << b

Example 1.11: Assignment Operators in Python

```
# Examples of Assignment Operators
a =10

# Assign value
b =a
print(b) #10

# Add and assign value
b +=a
print(b) #20

# Subtract and assign value
b -=a
print(b) #10

# Multiply and assign
b *=a
print(b) #100

# bitwise left shift operator
b <<=a
print(b) #102400
```

Output:

```
10
20
10
100
102400
```

1.4.6 Identity Operators

is and **is not** are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

- **is** True if the operands are identical.
- **is not** True if the operands are not identical.

Example 1.12: Identity Operator

```
a =100
b =200
c =a
print(a is not b)
print(a is c)
```

Output:

```
True
True
```

1.4.7 Membership Operators

in and **not in** are the membership operators; used to test whether a value or variable is in a sequence.

- **in** True if value is found in the sequence
- **not in** True if value is not found in the sequence.

Example 1.13: Membership Operator

```
# Python program to illustrate
# not 'in' operator
x =24
y =20
list=[10, 20, 30, 40, 50]

if(x not in list):
    print("x is NOT present in given list")
else:
    print("x is present in given list")

if(y in list):
  print("y is present in given list")
else:
  print("y is NOT present in given list")
```

Output:

```
x is NOT present in given list
y is present in given list
```

1.4.8 Precedence and Associativity Operators

- Operator precedence and associativity determine the priorities of the operator.
- This is used in an expression with more than one operator with different precedence to determine which operation to perform first.

Example 1.14: Operator Precedence

```
# Examples of Operator Precedence
python code.py > name
# Precedence of '+' & '*'
expr = 10 + 20 * 30
print(expr)

# Precedence of 'or' & 'and'
name = "Ali"
age = 0

if name == "Ali" or name == "John" and age >= 2:
    print("Hello! Python.")
else:
    print("Bye Bye!!")
```

Output

```
610
Hello! Python.
```

1.4.9 Ternary Operators

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

Syntax :

```
value_if_true if condition elsevalue_if_false
```

Example 1.15: python code.py >...

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

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

print(min)
```

Output:

```
10
```

Example 1.16: python code.py >...

```
#Ternary operator to print even or odd
x = 5
result = "Even" if x % 2 == 0 else "Odd"
print(result)
```

Output:

```
Odd
```

1.5 CONTROL FLOW: CONDITIONAL STATEMENTS (if, if ... else, nested if), LOOPING IN PYTHON (while loop, for loop, nested loops) LOOP MANIPULATION USING CONTINUE, PASS, BREAK, ELSE

1.5.1 Control Flow: Conditional Statements (if, ... else, nested if)

1.5.1.1 if statement

if statement is the simplest decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not.

Syntax

```
if condition:
    # Statements to execute if
    # condition is true
```

```
# python program to illustrate If statement
python code.py > ...
i = 12

if (i > 15):
    print("12 is les than 15")
print("I am Not in if")
i = 12

if (i < 15):
    print("12 is less than 15")
print("I am Not in if")
```

Output:

```
12 is less than 15
I am Not in if
```

As the condition present in the if statement is false. So, the block below the if statement is not executed.

1.5.1.2 if-else

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

Syntax

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```

Example 1.17: Python if else statement

```
python code.py >...
if else statement
i = 20
if (i < 15):
    print("i is smaller than 15")
    print("i'm in if Block")
else:
    print("i is greater than 15")
    print("i'm in else Block")
print("i'm not in if and not in else Block")
```

Output:

```
i is greater than 15
i'm in else Block
i'm not in if and not in else Block
```

The block of code following the else statement is executed as the condition present in the if statement is false after calling the statement which is not in block (without spaces).

1.5.1.3 nested-if

A nested if is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Python allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1):
  # Executes when condition1 is true
  if (condition2):
    # Executes when condition2 is true
  # if Block is end here
# if Block is end here
```

Example 1.18: Python Nested if

```
python code.py >...
# python program on nested If statement
i = 10
if (i == 10):
    # First if statement
    if (i < 15):
        print("i is smaller than 15")
    #Nested - if statement
    #Will only be executed if statement above
```

```
#it is true
if (i < 12):
    print("i is smaller than 12 too")
else:
    print("i is greater than 15")
```

Output:

```
i is smaller than 15
i is smaller than 12 too
```

1.5.2 Looping in Python (while loop, for loop, nested loops)

1.5.2.1 Python for Loop

Python For loop is used for sequential traversal i.e. it is used for iterating over an iterable like string, tuple, list etc. It falls under the category of **definite** iteration. Definite iterations mean the number of repetitions is specified explicitly in advance. In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is "for in" loop which is like for each loop in other languages. Let us learn how to use for in loop for sequential traversals.

Note: In Python, for loops only implements the collection-based iteration.

Syntax:

```
for var in iterable:
    # statements
```

Example 1.19: Python for Loop using List, String

```
python code.py > i
    # Python program Iterating over a list
    print("list Iteration")
    l = ["Seeks", "for", "Seeks"]
    for i in l:
        print(i)
```

Output:

```
List Iteration
Hello
I
am
list
```

```
# Iterating over a tuple (immutable)
python code.py >...
    #Iterating over a tuple (immutable)
    print("\nTuple Iteration")
    t = ("I", "am", "Tuple")
    for i in t:
        print(i)
```

Output:

```
Tuple Iteration
I
am
Tuple
```

```
# Iterating over a String
    python code.py >...
        print("\nString Iteration")
        s = "String"
        for i in s:
            print(i)
```

Output:

```
String Iteration
S
t
r
i
n
g
```

```
# Iterating over dictionary
    python code.py >
        #Iterating over dictionary
        print("\nDictionary Iteration")
        d = dict( )
        d['DSP'] = 22395
        d['DST'] = 22396
        for i in d:
            print("% s # d" % (i, d[i])
```

Output:

```
Dictionary Iteration
DSP    22395
DST    22396
```

range() function:

The Python range() function returns the sequence of the given number between the given range. The most common use of it is to iterate sequence type (Python range() List, string etc.) with for and while loop using Python.

Syntax of range()

Syntax: range(start, stop, step)

Parameter:

start: integer starting from which the sequence of integers is to be returned.

stop: integer before which the sequence of integers is to be returned. The range of integers ends at stop – 1.

step: integer value which determines the increment between each integer in the sequence.

Example 1.20: Example of Python range()

for in range():

```
python code.py >...
    #Python program Iterating over range 0 to n - 1
    n = 4
    for i in range (0, n):
        print(i)
```

Output :

```
0
1
2
3
```

```
#Even Number
python code.py > ...
    #printing a number
    for i in range (0, 10, 2):
        print(i, end=" ")
    print ( )
```

Output:

```
0 2 4 6 8
```

Iterating by the index of sequences:

```
python code.py > ...
    #Python program Iterating by index
    list = ["I", "am", "List"]
    for index in range(len(list)):
        print(list[index])
```

Output:

```
I
am
List
```

Using else statement with for loops:

```
python code.py >...
    #Python program combining else with for
    list = ["I", "am", "List"]
    for index in range(len(list)):
        print(list[index])
    else:
        print("Inside Else Block")
```

Output:

```
I
am
List
Inside Else Block
```

1.5.2.2 Python While Loop

Python While Loop is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in the program is executed. While loop falls under the category of **indefinite iteration**. Indefinite iteration means that the number of times the loop is executed is not specified explicitly in advance.

Syntax:

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

Example 1.18: Python While Loop

```
python code.py >...
    #Python program while loop
    count = 0
    while (count < 3)"
        count = count + 1
        print("Hello, Python!")
```

Output:

```
Hello, Python!
Hello, Python!
Hello, Phthon!
```

In the above example, the condition for while will be True as long as the counter variable (count) is less than 3.

Example 1.22: Python while loop with list

```
python code.py >...
    #checks if list still
    #contains any element
    a = [1, 2, 3, 4]

    while a:
        print(a.pop( ))
```

Output:

```
4
3
2
1
```

In the above example, we have run a while loop over a list that will run until

1.5.2.3 Loop Manipulation Using Continue, Pass, Break

Loops in Python automates and repeats the tasks in an efficient manner. But sometimes, there may arise a condition where you want to exit the loop completely, skip an iteration or ignore that condition.

These can be done by **loop control statements.** Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements.

- continue statement.
- break statement.
- pass statement.

1.5.2.3.1 Break Statement

"break" statement in Python is used to bring the control out of the loop when some external condition is triggered.

In Python, the "break" statement is a control flow statement that allows you to abruptly exit a loop before its normal completion. When the "break" statement is encountered within a loop, the program immediately exits the loop, and the control flow resumes with the next statement following the loop.

Syntax:

```
break
```

Example 1.23:

python code.py >...

```
#Python program demonstrate break statement
s = 'Hello, I am Python'
#Using for loop
for letter in s:
    print(letter)
    #break the loop as soon it sees ',' or 'a'
    if letter == ',' or letter == 'a':
        break
print("Out of for loop")
```

Output:

```
H
e
l
l
o
,
Out of for loop
```

Example 1.24:

python code.py >...

```
#Python programs demonstrate break statement
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
#Using for loop
for num in list:
    print(num)
    #break the loop as soon it sees '5'
    if num == 5:
        break
print("Out of for loop")
```

Output:

```
1
2
3
4
5
6
Out of for loop
```

1.5.2.3.2 Continue Statement

"continue" statement allows you to immediately skip the remaining code.

In Python, the "continue" statement is a control flow statement that is used within loops, such as "for" or "while" loops. When encountered, the "continue" statement allows you to immediately skip the remaining code within the current iteration of the loop and move on to the next iteration.

Syntax:

```
continue
```

Example: Continue statement in Python

Let us say you have a task to write a program that prints numbers from 1 to 10, but there is a catch: you cannot print the number 6. The only rule is that you can use just one loop. This is where the "continue" statement comes in handy. Here is how it works: you run a loop from 1 to 10, and each time you check if the current number is equal to 6. If it is, you use "continue" to skip that number and move on to the next one without printing anything. But if the number is not 6, you go ahead and print it. By following this approach, you can fulfill the given requirement successfully.

Below is the implementation of the above idea:

Example 1.25:

python code.py ? i

```
#Python program to demonstrate continue statement
#loop from 1 to 10
#If i is equals to 6, continue to next iteration without printing
if i ==6:
    continue
else
    #otherwise print the value of i
    print(i, end=" ")
```

Output:

1 2 3 4 5 6 7 8 8 10

Note: The continue statement can be used with any other loop also like while loop in a similar way as it is used with for loop above.

1.5.2.3.3 pass statement

In Python, the "pass" statement is a placeholder or a no-op (no operation) statement. It is used when a statement is syntactically required but you don't want to put any code or action in that block.

The "pass" statement does nothing and acts as a placeholder to satisfy the Python syntax requirements. It is often used in situations where you want to define a function, class, or conditional block but don't have the implementation ready yet.

Syntax:

```
pass
```

Example 1.26: pass statement can be used in empty functions

```
def myFunction:
    pass
```

Example 1.27: pass statement can also be used in empty class

```
class myClass:
    pass
```

Example 1.28: pass statement can be used in for loop when user doesn't know what to code inside the loop.

```
n =10
for in range(n):
    # pass can be used as placeholder
    # code is to added later
    pass
```

Practice Questions

1. Describe Python Interpreter.
2. List features of Python.
3. Determine various data types available in Python with example.
4. List different data types in Python.
5. Explain any 4 built-in numeric data types in Python.
6. State how to perform comments in Python.
7. Explain variable in Python with its rules and conventions for declaration.
8. Explain two Membership and two logical operators in python with appropriate examples.
9. Explain different loops available in python with suitable examples.
10. List identity operators.
11. Mention the use of //, **, % operator in Python.
12. Describe any two identity operators and two relational operators in Python.

Winter 2023

13. List the four features of python. **(2 Marks)**

Summer 2023

14. State any two control statements in python with suitable example. **(4 Marks)**
15. Enlist data types in python. Describe any two with suitable examples. **(4 Marks)**
16. Explain membership operator in python with example.

Winter 2022

17. List any two identity operators. **(2 Marks)**
18. State the use of //, **, % operator in Python. **(2 Marks)**
19. List data types used in Python. Explain any two with examples. **(4 Marks)**

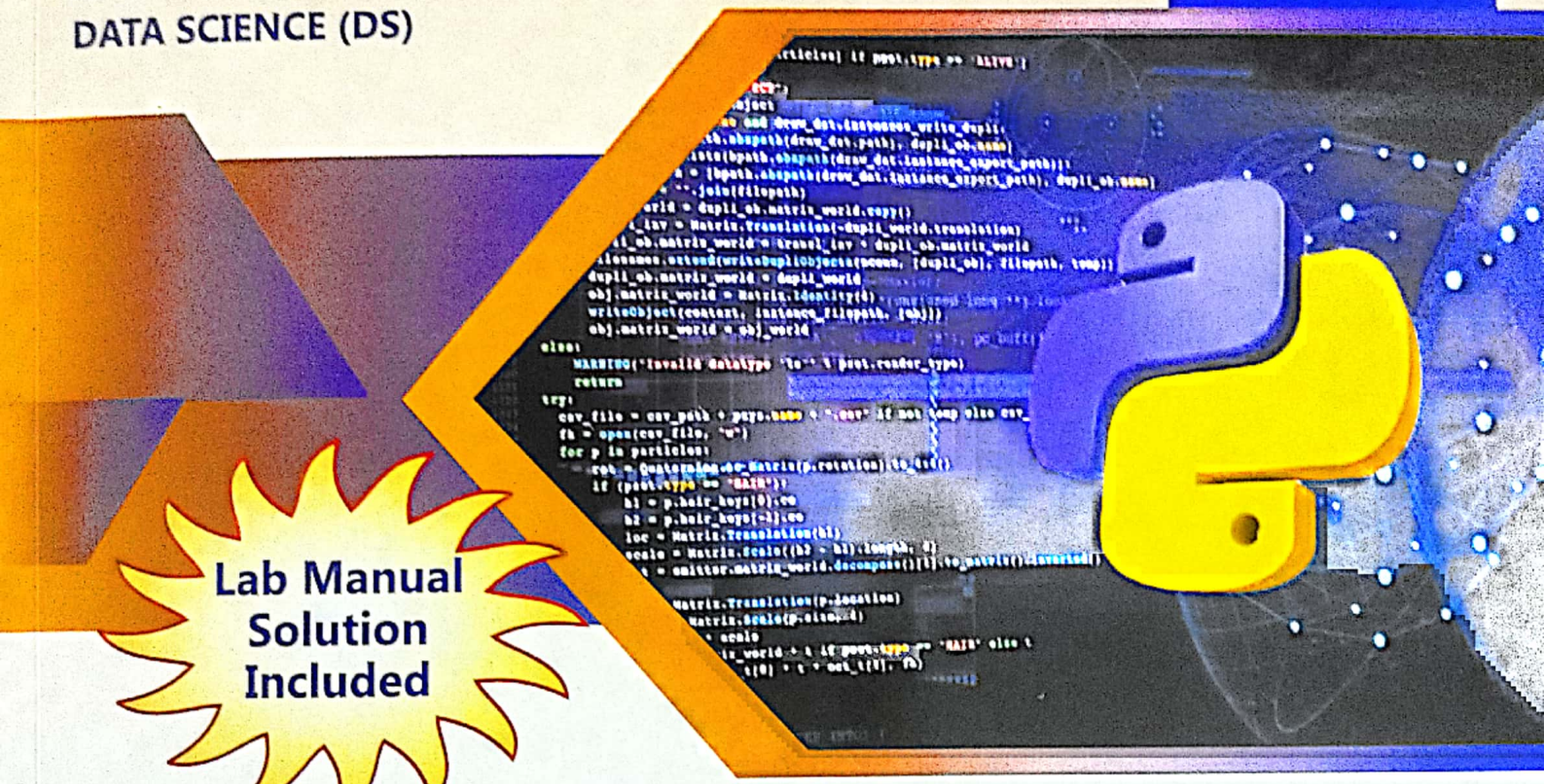
For Python Playlist of Video Lecture Scan QR Code.

SECOND YEAR DIPLOMA
Engineering and Technology

ARTIFICIAL INTELLIGENCE (AI) /
ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING (AN) /
DATA SCIENCE (DS)

MSBTE's K SCHEME

SEMESTER-III



Lab Manual
Solution
Included

DATA STRUCTURE USING PYTHON

Prof. ALI KARIM SAYED

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE