Lecture Notes

**Unit 1: **

1.1.1 Introduction Data, Expression, Statements

Introduction to Python and installation, data types: Int, float, Boolean, string, and list; variables, expressions, statements, precedence of operators, comments; modules, function and its use, flow of execution, parameters and arguments.

Introduction to Python and installation:

Python is a widely used general-purpose, high level programming language. It was initially designed by Guido van Rossum in 1991 and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Python is a programming language that lets you work quickly and integrate systems more efficiently.

There are two major Python versions- Python 2 and Python 3.

• On 16 October 2000, Python 2.0 was released with many new features.

• On 3rd December 2008, Python 3.0 was released with more testing and includes new features.

1.1.2 Beginning with Python programming:

1) Finding an Interpreter:

Before we start Python programming, we need to have an interpreter to interpret and run our programs. There are certain online interpreters like https://ide.geeksforgeeks.org/, http://ideone.com/ or http://codepad.org/ that can be used to start Python without installing

an interpreter.

Windows: There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from http://python.org/downloads/

Writing first program:

Script Begins

Statement1

PYTHON PROGRAMMING

2

Statement2

Statement3

Script Ends

1.1.3 Why to use Python:

The following are the primary factors to use python in day-to-day life:

1. Python is object-oriented

- Structure supports such concepts as polymorphism, operation overloading and multiple inheritance.

2. Indentation

- Indentation is one of the greatest feature in python

3. It's free (open source)

- Downloading python and installing python is free and easy

4. It's Powerful

- Dynamic typing
- Built-in types and tools
- Library utilities
- Third party utilities (e.g. Numeric, NumPy, sciPy)
- Automatic memory management

5. It's Portable

- Python runs virtually every major platform used today
- As long as you have a compaitable python interpreter installed, python programs will run in exactly the same manner, irrespective of platform.

6. It's easy to use and learn

- No intermediate compile
- Python Programs are compiled automatically to an intermediate form called
- byte code, which the interpreter then reads.
- This gives python the development speed of an interpreter without the
- performance loss inherent in purely interpreted languages.
- Structure and syntax are pretty intuitive and easy to grasp.

7. Interpreted Language

- Python is processed at runtime by python Interpreter

8. Interactive Programming Language

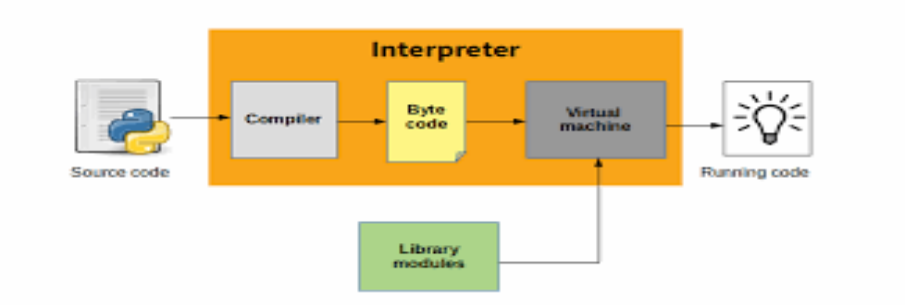
- Users can interact with the python interpreter directly for writing the programs

9. Straight forward syntax

- The formation of python syntax is simple and straight forward which also makes it popular.

1.1.4 Python Code Execution:

Python's traditional runtime execution model: Source code you type is translated to byte code, which is then run by the Python Virtual Machine (PVM). Your code is automatically compiled, but then it is interpreted.



1.1.5 String:

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes.

'hello' is the same as "hello".

Strings can be output to screen using the print function. For example: print("hello").


```
>>> print("Silver Oak college")
Silver Oak college
>>> type("Silver Oak college")
<class 'str'>
```

PYTHON PROGRAMMING III YEAR/II SEM MRCET

9

```
>>> print(Silver Oak college')
Silver Oak college
>>> " "
' '
```

If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type. If a string is to contain a single quote, delimit it with double quotes and vice versa:

```
>>> print("Silver Oak is an autonomous (') college")
```

Silver Oak is an autonomous (') college

>>> print(' Silver Oak is an autonomous (") college')

Silver Oak is an autonomous (") college

Suppressing Special Character:

Specifying a backslash (\) in front of the quote character in a string “escapes” it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single quote character:

>>> print("Silver Oak is an autonomous (\') college")

Silver Oak is an autonomous (') college

>>> print(' Silver Oak is an autonomous (\") college')

Silver Oak is an autonomous (") college

The following is a table of escape sequences which cause Python to suppress the usual special interpretation of a character in a string:

>>> print('a\

....b')

a....b

>>> print('a\

b\

c')

PYTHON DEVELOPMENT

10

abc

>>> print('a \n b')

a

b

>>> print("Silver Oak \n college")

Silver Oak

College

1.2.1 Branching Programs

Branching programs allow you to make decisions within a program. By evaluating certain conditions, the program can decide which blocks of code to execute. This concept is fundamental in programming, enabling the creation of dynamic and responsive software.

1. Conditional Statements

Conditional statements in Python are used to execute code based on whether a certain condition is True or False. The basic building blocks of branching in Python are if, elif, and else.

- if Statement

Purpose: The if statement is used to test a condition. If the condition evaluates to True, the code block inside the if statement is executed. If it evaluates to False, the code block is skipped.

Syntax:

```
if condition:
    # Code block that runs if the condition is True
```

Example:

```
age = 20
if age >= 18:
    print("You are eligible to vote.")
```

Explanation: In this example, the condition age >= 18 is checked. If the age is 18 or more, the message "You are eligible to vote." is printed.

- elif Statement

Purpose: The elif (short for "else if") statement allows you to check multiple conditions sequentially. If the first if condition is False, the program moves on to the elif condition. You can have as many elif statements as you need.

Syntax:

```
if condition1:
    # Code block for condition1
elif condition2:
    # Code block for condition2
```

Example:

```
score = 85
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
else:
    print("Grade: D")
```


Explanation: The program checks the conditions in order:

If score >= 90, it prints "Grade: A".

If score >= 80 but less than 90, it prints "Grade: B".

If score >= 70 but less than 80, it prints "Grade: C".

If none of the above conditions are met, it defaults to printing "Grade: D".

- else Statement

Purpose: The else statement is optional and executes a block of code when all previous conditions are False. It acts as a "catch-all" if none of the preceding conditions are met.

Syntax:

```
if condition:
    # Code block for condition
else:
    # Code block if condition is False
```

Example:

```
temperature = 25
if temperature > 30:
    print("It's hot outside.")
else:
    print("It's cool outside.")
```

Explanation: If the temperature is greater than 30, the message "It's hot outside." is printed. If it's 30 or less, the message "It's cool outside." is printed.

Branching programs are essential for making decisions within a Python program. By using conditional statements (if, elif, else), nested conditions, and logical operators, you can control the flow of your program based on various conditions. These tools allow you to create dynamic, interactive applications that respond intelligently to different inputs and situations.

1.3.1 Control Structures

Loops

Loops allow you to execute a block of code multiple times, either a fixed number of times or until a certain condition is met. Python supports two main types of loops: for and while.

- for Loop

Purpose: Iterates over a sequence (such as a list, tuple, string, or range) and executes a block of code for each item in the sequence.

Syntax:

```
for item in sequence:
```

```
    # Code block to be executed for each item
```

Example:

```
for i in range(5):
    print(i)
```

Explanation: This loop prints numbers 0 through 4.

Looping Through Lists:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Explanation: This loop prints each item in the fruits list.

- While Loop

Purpose: Repeats a block of code as long as a specified condition is True.

Syntax:

```
while condition:
    # Code block to be executed while condition is True
```

Example:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Explanation: This loop prints numbers 0 through 4. The count variable is incremented by 1 in each iteration.

1.3.2 Loop control Statements

Python provides several statements to control the flow of loops: break, continue, and else with loops.

- break Statement

Purpose: Terminates the loop prematurely, skipping any remaining iterations.

Syntax:

```
for item in sequence:
    if condition:
        break
    # Code block to be executed if break is not triggered
```

Example:

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Explanation: The loop stops when i equals 5, so only numbers 0 through 4 are printed.

- continue Statement

Purpose: Skips the current iteration of the loop and moves to the next iteration.

Syntax:

```
for item in sequence:
    if condition:
        continue
    # Code block to be executed if continue is not triggered
```

Example:

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Explanation: The number 2 is skipped, so the loop prints 0, 1, 3, and 4.

- else with Loops

Purpose: The else block is executed after the loop completes normally (i.e., without encountering a break).

Syntax:

```
for item in sequence:
    # Loop code block
else:
    # Code block executed if the loop did not encounter a break
```

Example:

```
for i in range(5):
    print(i)
else:
    print("Loop completed without a break.")
```

Explanation: After the loop prints numbers 0 through 4, the else block is executed.

1.3.3 Nested Loops

Purpose: A loop inside another loop. Each iteration of the outer loop triggers a full iteration of the inner loop.

Syntax:

```
for item1 in sequence1:
    for item2 in sequence2:
        # Code block to be executed for each combination of item1 and item2
```

Example:

```
for i in range(3):
    for j in range(2):
        print(f"i={i}, j={j}")
```

Explanation: The outer loop runs 3 times, and for each iteration, the inner loop runs 2 times.

Control structures in Python, including conditional statements (if, elif, else), loops (for, while), and loop control statements (break, continue, else with loops), are essential for directing the flow of a program. These structures allow you to create dynamic, responsive

programs that can handle a wide range of tasks, from making decisions to iterating over data. Understanding and mastering these control structures are key to becoming proficient in Python programming.

1.4.1 Strings and Inputs

A string is a sequence of characters enclosed within single quotes (' ') or double quotes (" "). Python also supports multi-line strings enclosed within triple quotes (''' ''' or """ """).

- Creating Strings

a. Single-line strings:

```
name = "Alice"
greeting = 'Hello, world!'
```

b. Multi-line strings:

```
message = """This is a multi-line string.
It spans multiple lines.
"""
```


- Accessing Characters in a String

Strings are indexed, meaning each character in a string has a position, starting from 0 for the first character.

Example:

```
word = "Python"
print(word[0])  # Output: P
print(word[-1]) # Output: n
```

Explanation: You can access characters using positive indices (left to right) or negative indices (right to left).

- String Slicing

String slicing allows you to access a range of characters within a string.

Syntax:

string[start:end]

start: The starting index (inclusive).

end: The ending index (exclusive).

Example:

```
word = "Python"
print(word[0:2])  # Output: Py
print(word[2:])   # Output: thon
print(word[:4])   # Output: Pyth
```

- String Concatenation

You can combine (concatenate) strings using the + operator.

Example:

```
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
print(full_name)  # Output: John Doe
```

- String Repetition

You can repeat strings using the * operator.

Example:

```
echo = "Hello! " * 3
print(echo)  # Output: Hello! Hello! Hello!
```

- String Methods

Python provides various built-in methods to manipulate strings.

Common String Methods:

upper(): Converts all characters to uppercase.

```
print("python".upper())  # Output: PYTHON
```

lower(): Converts all characters to lowercase.

```
print("PYTHON".lower())  # Output: python
```

strip(): Removes leading and trailing whitespace.

```
print("  hello  ".strip())  # Output: hello
```

replace(old, new): Replaces occurrences of a substring with another.

```
print("Hello world".replace("world", "Python"))  # Output: Hello Python
```


split(separator): Splits the string into a list of substrings based on a separator.

```
sentence = "Python is fun"
words = sentence.split(" ")
print(words)  # Output: ['Python', 'is', 'fun']
```

join(iterable): Joins elements of an iterable into a single string, separated by a string.

```
words = ['Python', 'is', 'fun']
sentence = " ".join(words)
print(sentence)  # Output: Python is fun
```

- String Formatting

String formatting allows you to create strings with dynamic content, such as inserting variables into a string.

f-strings (Python 3.6+):

```
name = "Alice"
age = 25
```

print(f"My name is {name} and I am {age} years old.")

format() method:

name = "Alice"

age = 25

print("My name is {} and I am {} years old.".format(name, age))

Percentage formatting (older style):

name = "Alice"

age = 25

print("My name is %s and I am %d years old." % (name, age))

1.4.2 Input Handling in Python

The input() function allows you to capture user input as a string.

- Basic Input

Example:

name = input("Enter your name: ")

print(f"Hello, {name}!")

Explanation: The program prompts the user to enter their name and then greets them using the input.

- Converting Input Types

By default, the input() function returns user input as a string. To work with other data types (e.g., integers, floats), you need to convert the input.

Example:

age = int(input("Enter your age: "))

print(f"You are {age} years old.")

Explanation: The input() function captures the user's age as a string, which is then converted to an integer using int().

1.4.3 Other Conversions

- Float:

```
height = float(input("Enter your height in meters: "))
print(f"Your height is {height} meters.")
```

Boolean:

```
is_student = input("Are you a student (yes/no)? ").lower() == "yes"
print(f"Student status: {is_student}")
```

1.4.4 Handling Multiple Inputs

To capture multiple inputs in a single line, you can use the split() method.

Example:

```
x, y = input("Enter two numbers separated by a space: ").split()
x = int(x)
y = int(y)
print(f"The sum of {x} and {y} is {x + y}.")
```


1.5.1 Iterators in Python

An iterator in Python is an object that is used to iterate over iterable objects like lists, tuples, dicts, and sets. The Python iterators object is initialized using the iter() method. It uses the next() method for iteration.

- __iter__(): The iter() method is called for the initialization of an iterator. This returns an iterator object.
- __next__(): The next method returns the next value for the iterable. When we use a for loop to traverse any iterable object, internally it uses the iter() method to get an iterator object, which further uses the next() method to iterate over. This method raises a Stop Iteration to signal the end of the iteration.

1.6.1 Scoping

In Python, variables are the containers for storing data values. Unlike other languages like C/C++/JAVA, Python is not “statically typed”. We do not need to declare variables before

using them or declare their type. A variable is created the moment we first assign a value to it.

Python Scope variable

The location where we can find a variable and also access it if required is called the scope of a variable.

1.6.2 Python Local variable

Local variables are those that are initialized within a function and are unique to that function. It cannot be accessed outside of the function. Let's look at how to make a local variable.

```
def f():
  # local variable
   s = "Python"
   print(s)
f()
```

Output

Python

If we will try to use this local variable outside the function then let's see what will happen.

```
def f():
    # local variable
   s = "Python"
   print("Inside Function:", s)
f()
print(s)
```

Output:

NameError: name 's' is not defined

1.6.3 Python Global variables

Global variables are the ones that are defined and declared outside any function and are not specified to any function. They can be used by any part of the program.

Example:

```
# This function uses global variable s
def f():
    print(s)


# Global scope
s = "Python"
f()
```

Output:

1.7.1 Specifications in Python

Specifications in Python refer to the process of clearly defining what a function or a piece of code is supposed to do. Writing specifications is essential for creating reliable, maintainable, and understandable code. Specifications typically involve writing detailed comments or docstrings that describe the input parameters, the output, and any assumptions or conditions that must be met for the function to work correctly.

1.7.2 What Are Specifications?

Specifications act as a contract between the function and its users (which might be other functions or human developers). They tell you:

What the function does (its purpose).

What inputs the function expects (parameters).

What output the function produces (return value).

What assumptions or constraints exist on the input or the environment.

1.7.3 Importance of Specifications

Clarity: They make it clear to others (and to your future self) what the function is supposed to do.

Error Prevention: By defining clear expectations, you reduce the chances of misusing the function or passing incorrect arguments.

Testing and Debugging: Specifications make it easier to write tests because they describe the expected behavior of the function.

1.7.4 Writing Specifications Using Docstrings

In Python, specifications are commonly written using docstrings. A docstring is a string literal that occurs as the first statement in a function, class, or module. It describes the purpose of the function and details about its parameters and return values.

Docstring Example

Example:

```
def add(a, b):
    """
    Add two numbers.

    Parameters:
    a (int or float): The first number.
    b (int or float): The second number.

    Returns:
    int or float: The sum of the two numbers.
    """
    return a + b
```


Explanation: This docstring explains that the add function takes two numbers (a and b) and returns their sum. It also specifies the expected types of the parameters and the return value.

1.8.1 Recursion

Recursion occurs when a function calls itself within its definition. Each recursive call works on a smaller instance of the problem until it reaches a base case, which stops the recursion.

1.8.1 Components of a Recursive Function

Base Case: The condition under which the function stops calling itself. This is crucial to prevent infinite recursion.

Recursive Case: The part of the function where it calls itself with a modified argument that brings it closer to the base case.

1.8.2 Example of a Simple Recursive Function

Example:

```
def countdown(n):
    if n <= 0:  # Base case
        print("Blast off!")
    else: # Recursive case
        print(n)
        countdown(n-1)
countdown(5)
```

Explanation: The countdown function prints a number and then calls itself with n-1 until n reaches 0, at which point it prints "Blast off!" and stops.

1.8.3 How Recursion Works: Call Stack

When a recursive function is called, each invocation is placed on the call stack. The call stack keeps track of the active function calls, and when a base case is reached, the function calls begin to return in reverse order.

1.8.4 Visualizing the Call Stack

Consider the factorial function with n=3:

factorial(3) calls factorial(2).

factorial(2) calls factorial(1).

factorial(1) calls factorial(0).

factorial(0) returns 1 (base case).

factorial(1) returns 1 * 1 = 1.

factorial(2) returns 2 * 1 = 2.

factorial(3) returns 3 * 2 = 6.

The call stack grows with each recursive call and shrinks as the base case is reached and the functions return.

1.8.5 Types of Recursion

- Direct Recursion

Direct recursion occurs when a function calls itself directly.

Example:

```
def direct_recursion():
    direct_recursion()
```


- Indirect Recursion

Indirect recursion occurs when a function calls another function, which eventually calls the original function.

Example:

```
def function_a():
    function_b()


def function_b():
    function_a()
```

Explanation: function_a calls function_b, and function_b calls function_a, creating an indirect recursion loop.

1.9.1 Global Variables & Modules

When working with multiple modules in Python, you may want to share global variables across different modules. To do this, you typically define the global variable in one module and then import it into other modules.

1.9.2 Defining a Global Variable in a Module

Module config.py:

```
# config.py
global_var = 42
```

Explanation: The global_var is defined in the config.py module.

1.9.2 Accessing a Global Variable from Another Module

Module main.py:

```
# main.py
import config
def print_global_var():
    print(config.global_var)  # Accessing the global variable from config module
print_global_var()  # Output: 42
```

Explanation: The global_var defined in config.py is accessed in the main.py module.

1.9.3 Modifying Global Variables Across Modules

To modify a global variable in another module, you need to use the global keyword. This tells Python that you want to use the global variable from the imported module, not create a new local variable.

```
# config.py
global_var = 42
```

Module main.py:

```
# main.py
import config
```

```
def modify_global_var():
    global config.global_var
    config.global_var = 100  # Modifying the global variable

modify_global_var()
print(config.global_var)  # Output: 100
```

Explanation: The global keyword is not used here because config.global_var directly refers to the variable in the config module. When modifying a global variable from another module, you can directly assign a new value to it.

1.10.1 System Functions and Parameters

In Python, system functions enable interaction with the operating system and the environment in which your code is running. These functions are primarily provided by the `os`, `sys`, and `subprocess` modules, each serving different purposes.

The `os` module offers functions for interacting with the file system, managing environment variables, and executing system commands. For example, `os.system(command)` allows you to execute a command as if you were typing it in the terminal. `os.getenv(key, default=None)` is used to retrieve environment variables, while `os.listdir(path='.')` lists files in a directory. Additionally, `os.path.join(path, *paths)` helps in creating platform-independent file paths by intelligently joining path components.

The `sys` module provides access to variables and functions related to the Python interpreter. `sys.argv` is a list containing command-line arguments passed to the script. `sys.exit([status])` allows you to terminate the program with a specified exit status. `sys.path` lists directories where Python looks for modules, and `sys.platform` reveals the operating system on which Python is running.

For more advanced system interactions, the `subprocess` module is used. `subprocess.run(args, ...)` is a powerful function for running commands and capturing their output. It waits for the command to complete and returns a `Completed Process` instance. `subprocess.Popen(args, ...)` offers even more control by allowing you to spawn new processes, connect to their input/output pipes, and handle their execution in a non-blocking way.

These system functions are essential for tasks such as file management, environment manipulation, running external commands, and process management. Understanding and

utilizing them effectively is crucial for writing Python scripts that can interact seamlessly with the operating system.