# TITLE: PYTHON DATA TYPES AND BASIC INPUT/OUTPUT FUNCTIONS

**OBJECTIVE**:

* Python installation and environment setup.
* Understand python data types.
* Learn basic input/output functions in python.

**THEORY**:

Python is a versatile high-level programming language that is widely used in various applications including: web development, data analysis, machine learning, artificial intelligence, automation, scientific computing, and soon. Python was first released in 1991, created by Guido Van Rossum and developed by Python Software Foundation. It is an interpreter based open-source programming language. Python 3.12 (3.X Series) is currently the latest released python version. It has large ecosystem of libraries like: NumPy, PyTorch, PyBrain, Sympy, Flask etc. and frameworks like: Django, flask, Cherrypy, FastAPI, etc.

**Python Installation:**

* Download python latest version that suits the OS at python.org in the download section.
* Run the downloaded executable.
* Verify installation: Execute “python –version” in terminal. If it shows the installed python version then python is installed in the system.

**Python data types:**

The basic python data types are: Int, float, str, list, set, tuple, dict, none, bool, bytes, bytearray, memoryview.

* **int**: Represents integers (whole numbers).
* **float**: Represents floating-point numbers (decimal numbers).
* **str**: Represents strings (sequences of characters).
* **list**: Represents ordered, mutable sequences of elements.
* **set**: Represents an unordered collection of unique elements.
* **tuple**: Represents ordered, immutable sequences of elements.
* **dict**: Represents key-value pairs in an unordered, mutable collection.
* **None**: Represents the absence of a value or a null value.
* **bool**: Represents Boolean values (True or False).
* **bytes**: Represents immutable sequences of bytes.
* **bytearray**: Represents mutable sequences of bytes.
* **memoryview**: Represents a view object that exposes an array's buffer interface.

**Rules for naming variables:**

Consider the following key notes regarding python variables.

* should contain valid characters: a-z, A-Z, 0-9 and underscore ( *\_* ). Name should not begin with numeric character. *Valid num\_1. Invalid: 1\_Num*.
* variables names are case sensitive.
* reserved keywords cannot be used as a variable name.
* can be of any length but short and meaningful is recommended.
* Data type declaration is avoided. During runtime python itself assigns the data type as considering stored value.
* Multiple declaration separated by commas. *Num\_1, Num\_2=5,7.*
* Strings can be defined inside ‘ ’ or “ ”.
* Python does not assign new memory for each new variables having same value. However, multiple memory will be assigned for same variable if value is updated.

*Example: num\_1st, num\_2nd = 5, 5. #Single memory is assigned.*

*Num\_1st= 9 #Another memory is assigned for Num\_1st this time*

* A constant is defined by the variable in all caps. Example: *PIE=3.14 #constant*

**Python Comments:**

* Comments are used for documentation or to increase the code readability and for future references like: debugging, maintainability and regular updates.
* Comments in python begins with #. Example: *# This is a single Line comment.*
* Multiple line comments are written inside ''' '''. This is actually not for comment but denotes multi-line string. However not defining any variable lead the string be ignored and thus used for defining multiple line comments.
* Shortcut for Python comment (Vs Code: Ctrl + ?)

**Python Indentations:**

While in C/C++, the indentations are used for organizing or structuring the code, indentation matters much in python.

* The indentation represents the block of code.
* The codes with same indentation belong to the same block.
* Indentation is achieved pressing tab or four spaces.

**Python Statements:**

Statements are building blocks of a Python program, and they represent specific actions or commands. There are several types of statements in Python: assignment statement, conditional statement, expression statement, loop statement, function definition, class definition, etc.

*Eg: a=5 #assignment statement*

*Eg: for i in range(5): # loop statement*

*print(i)*

*Eg: print(“Hello World !”) #expression statement*

**Input Function:**

* input(Display\_string) is a built-in python function to take input from user.

*Example: name = input(“Enter your Name”) # allows user to input the name*

Output Function:

* print( ), a built-in function is used to display the output.
* Attributes like sep, end can be defined to place desired separator and end line for formatted output.

*Example: name = print(“Hello”, “World”,sep=“!!”, end= “!!”)*

*(Outputs: Hello!!World!!) Inserted !! as the separator of two strings and the end too.*

Some formatted Outputs:

*x, y="Python", "Programming"*

*print(f"{x} is an interpreted based {y} language") # f-strings*

*print(f"{} is an interpreted based {} language". format (x. y))*

*# aligns x and y in the 20 characters left and right respectively. [ :^20 for center ]*

*print(f"{:<20} is an interpreted based {:>20} language". format (x, y))*

*#Slicing: [start : stop : step]*

*print( x [1:7: 2]) #Output: Pto*

**Python operators:**

1. **Arithmetic Operators:**
   * **+**: Sum of numbers, e.g., 5 + 2 = 7
   * **-**: Difference of numbers, e.g., 10 - 3 = 7
   * **\***: Product of numbers, e.g., 3 \* 2 = 6
   * **/**: Division, e.g., 5 / 2 = 2.5
   * **//**: Floor division (Quotient), e.g., 5 // 2 = 2
   * **%**: Remainder, e.g., 5 % 2 = 1
   * **\*\***: Exponent, e.g., 2 \*\* 3 = 8
2. **Comparison Operators:**
   * **==**: Equal to, e.g., 3 == 3 is True
   * **!=**: Not equal to, e.g., 5 != 3 is True
   * **<**: Less than, e.g., 2 < 5 is True
   * **>**: Greater than, e.g., 7 > 4 is True
   * **<=**: Less than or equal to, e.g., 4 <= 4 is True
   * **>=**: Greater than or equal to, e.g., 6 >= 7 is False
3. **Logical Operators:**
   * **and**: Logical AND, e.g., True and False is False
   * **or**: Logical OR, e.g., True or False is True
   * **not**: Logical NOT, e.g., not True is False
4. **Assignment Operators:**
   * **=**: Assigns a value to a variable, e.g., x = 10
   * **+=**: Adds the right operand to the left operand and assigns the result, e.g., x += 5 is equivalent to x = x + 5
   * **-=**: Subtracts the right operand from the left operand and assigns the result, e.g., x -= 3 is equivalent to x = x - 3
   * **\*=**: Multiplies the left operand by the right operand and assigns the result, e.g., x \*= 2 is equivalent to x = x \* 2
   * **/=**: Divides the left operand by the right operand and assigns the result, e.g., x /= 3 is equivalent to x = x / 3
5. **Bitwise Operators:**
   * **&**: Bitwise AND, e.g., 5 & 3 = 1
   * **|**: Bitwise OR, e.g., 5 | 3 = 7
   * **^**: Bitwise XOR, e.g., 5 ^ 3 = 6
   * **~**: Bitwise NOT, e.g., ~5 = -6 (in 2's complement form)
   * **<<**: Left Shift, e.g., 5 << 1 = 10
   * **>>**: Right Shift, e.g., 5 >> 1 = 2
6. **Identity Operators:**
   * **is**: True if the operands are identical, e.g., x is y
   * **is not**: True if the operands are not identical, e.g., x is not y
7. **Membership Operators:**
   * **in**: True if a value is found in the sequence, e.g., 2 in [1, 2, 3] is True
   * **not in**: True if a value is not found in the sequence, e.g., 4 not in [1, 2, 3] is True

**Debugging and Testing:**

**Debugging in Python:** Debugging in Python involves finding and fixing errors in your code. Common approaches include using print statements to output variable values, employing the built-in debugger (**pdb**) for interactive inspection, and utilizing the **logging** module for recording program execution. Assertions help check conditions during development.

**Testing in Python:** Testing is crucial for ensuring code reliability. In Python, unit testing is done with frameworks like **unittest** or **pytest**. Integration testing checks the collaboration of different program parts, and automated testing helps catch regressions. Tools like **coverage** measure test coverage, and libraries like **unittest.mock** assist in controlled testing environments. Integrating testing into continuous integration pipelines ensures ongoing code quality. Combining effective debugging and testing practices is essential for maintaining robust and error-free Python code.

**LAB ACTIVITY**

# Python Lab Code

# Python data types

num\_1 = 5 # int

num\_2 = 3.14 # float

name = "Python" # str

my\_list = [1, 2, 3] # list

my\_set = {1, 2, 3} # set

my\_tuple = (1, 2, 3) # tuple

my\_dict = {'a': 1, 'b': 2} # dict

none\_value = None # None

is\_true = True # bool

bytes\_data = b'hello' # bytes

bytearray\_data = bytearray(b'world') # bytearray

memory\_view = memoryview(b'hello') # memoryview

# Input and Output Functions

user\_input = input("Enter your name: ")

print("Hello, " + user\_input + "!")

# Python Statements

for i in range(5):

print(i)

# Python Operators

result = num\_1 + num\_2

print("Sum:", result)

# Debugging

# Uncomment the following line to intentionally create an error

# result = num\_1 / 0

# Testing

# A simple unit test

def test\_addition():

assert num\_1 + num\_2 == 8.14

test\_addition()

print("Testing passed!")

# Formatted Output

print(f"{name} is an interpreted based language")

# Python Comments

# This is a single-line comment

'''

This is a

multi-line comment

'''

# Indentation

if is\_true:

print("It is true!")

# Assignment Operator

x = 10

print("Value of x:", x)

# Bitwise Operator

bitwise\_result = num\_1 and num\_2

print("Bitwise AND:", bitwise\_result)

**OUTPUT:**

