Day 3 Materials:

Variables, Keywords, Indentation, Comments:

1. Variables

- **Definition**: Containers for storing data values. In Python, variables are created when you assign a value to them.
- **Syntax**:
- name = "John" # String variable
- age = 25 # Integer variable
- height = 5.9 # Float variable
- **Key Points**:
    - Python is dynamically typed, so no need to declare the type explicitly.
    - Variable names should start with a letter or underscore (_) but not a number.

---

2. Keywords

- **Definition**: Reserved words in Python that have a specific meaning and cannot be used as variable names.
- **Examples**:
- False, True, None, and, as, assert, break, class, continue, def, del, elif, else, except, finally, for, from, global, if, import, in, is, lambda, nonlocal, not, or, pass, raise, return, try, while, with, yield
- Use the following to view all keywords:
- import keyword
- print(keyword.kwlist)

---

3. Indentation

- **Definition**: Used to define blocks of code. Python does not use braces {} like other languages; instead, it uses indentation.
- **Example**:
- if True:
- print("Indented correctly")
- **Error**: Improper indentation will throw an error.

---

4. Comments

- **Definition**: Used to write notes or disable code during development.
- **Single-line Comment**:
- # This is a single-line comment
- **Multi-line Comment**:
- """
- This is a
- multi-line comment
- """

String, Numbers, Boolean, List, Tuples

1. Strings

- **Definition**: A string is a sequence of characters enclosed in quotes (' ', " "). Strings are immutable, meaning they cannot be changed after creation. Python provides several methods like .lower(), .upper(), .split(), and slicing to manipulate strings. Strings support indexing and concatenation using +.name = "Python"
- print(name.upper()) # Converts to uppercase
- print(name[0]) # Access character by index
- print(len(name)) # Get the length of the string

- **String Concatenation**:
- first_name = "John"
- last_name = "Doe"
- full_name = first_name + " " + last_name
- print(full_name)

---

2. Numbers

Definition: Python supports integers, floats, and complex numbers. Integers are whole numbers, floats represent decimal values, and complex numbers have real and imaginary parts. Arithmetic operations like addition (+), subtraction (-), and modulus (%) can be performed on numbers. Python dynamically assigns the data type.

- Types: Integer, Float, Complex
- num1 = 10 # Integer
- num2 = 3.14 # Float
- num3 = 1 + 2j # Complex
- print(type(num3)) # Outputs <class 'complex'>

---

3. Boolean

Definition: Booleans represent one of two values: True or False. They are used in logical operations like comparisons (>, <, ==) and control flow statements such as if and while. Booleans can be derived from expressions or explicitly defined.

- Represents True or False.
- is_active = True
- is_logged_in = False
- print(is_active and is_logged_in) # Logical AND operation

---

4. List

Definition: A list is an ordered, mutable collection of items, defined using square brackets ([]). Lists can store heterogeneous data types and support operations like appending, removing, and sorting elements. Examples: my_list = [1, 'Python', 3.14].

- Ordered, mutable, and allows duplicate values.
- **Example**:
- fruits = ["apple", "banana", "cherry"]
- fruits.append("orange") # Add an item
- print(fruits[1]) # Access by index
- fruits[1] = "blueberry" # Modify value

---

5. Tuples

Definition: Tuples are immutable, ordered collections of items, defined using parentheses (()). Once created, their elements cannot be changed. Tuples are often used for fixed data, like coordinates. Example: my_tuple = (1, 'Python', 3.14).

- Ordered, immutable, and allows duplicate values.
- **Example**:
- numbers = (1, 2, 3)
- print(numbers[0]) # Access by index
- # numbers[0] = 10 # Throws an error because tuples are immutable

---

Sets, Dictionary, Arrays, Type Casting

1. **Sets**

Definition: Sets are unordered collections of unique items, defined using curly braces ({}). They do not allow duplicate elements and support operations like union, intersection, and difference. Example: my_set = {1, 2, 3}.

- Unordered, mutable, does not allow duplicate values.

- **Example**:
- my_set = {1, 2, 3, 3} # Duplicate 3 will be removed
- my_set.add(4) # Add an element
- print(my_set)

---

2. **Dictionary:**

Dictionaries are unordered collections of key-value pairs, defined using curly braces ({}). Keys are unique, and values can be any data type. Example: my_dict = {'name': 'Alice', 'age': 25}. Useful for fast lookups and data mapping.

- Collection of key-value pairs, unordered and mutable.
- **Example**:
- student = {"name": "John", "age": 21}
- print(student["name"]) # Access value by key
- student["age"] = 22 # Update value
- student["grade"] = "A" # Add a new key-value pair

---

3. **Arrays**

Definition: Arrays are collections of elements of the same data type, commonly used for numerical computations. Unlike lists, arrays require explicit import using libraries like array or numpy for advanced operations. Example: from array import array; my_array = array('i', [1, 2, 3]).

- Homogeneous data structures for storing a collection of items (requires array module).
- **Example**:
- from array import array
- numbers = array('i', [1, 2, 3]) # 'i' for integer array
- numbers.append(4)
- print(numbers)

4. Type Casting

Definition: Type casting converts one data type to another, such as int to float or string to int. It is achieved using functions like int(), float(), str(), and list(). Example: str(123) converts an integer to a string. Useful for ensuring compatibility between data types.

- Converting one data type to another.
- **Example**:
- # String to Integer
- num_str = "123"
- num_int = int(num_str)
- print(type(num_int))
-
- # List to Set
- my_list = [1, 2, 3]
- my_set = set(my_list)
- print(my_set)

Here's a detailed explanation of the additional topics:

---

1. Output Formatting

Output formatting in Python refers to controlling the structure and appearance of output. The print() function allows combining strings, variables, and expressions. Advanced formatting can be achieved using f-strings, .format() method, or % formatting.
Example:

name = "Alice"

age = 25

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

---

2. Taking User Input and Type Casting

Python allows taking user input using the input() function, which always returns data as a string. Type casting is used to convert this string into the required type, such as int or float.

Example:

```
age = input("Enter your age: ")  # Always returns a string
age = int(age)  # Cast string to integer
print(f"You are {age} years old.")
```

---

3. Multiple Inputs from Users

To take multiple inputs in a single line, you can use the input() function with .split(). By default, .split() separates inputs by spaces. These inputs can be type-cast as needed.

Example:

```
x, y, z = input("Enter three numbers separated by space: ").split()
x, y, z = int(x), int(y), int(z)  # Type casting
print(f"The sum of numbers is: {x + y + z}")
```

More Elaborated examples:

Here's an extended explanation with more examples for each topic:

---

1. Output Formatting

Output formatting helps display information clearly and concisely. Common methods include:

- **f-strings**: Introduced in Python 3.6, allows embedding expressions inside string literals using {}.
- **.format() Method**: Useful for positional and keyword arguments.
- **Old-Style Formatting (%)**: An older approach using %s, %d, etc.

Examples:

```
# Using f-strings

name = "Alice"

age = 25

score = 93.4567

print(f"My name is {name}, I am {age} years old, and my score is {score:.2f}.")  # Format float to 2 decimal places


# Using .format()

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


# Using Old-Style Formatting

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

Advanced Formatting:

```
# Aligning text

print(f"|{'Left':<10}|{'Center':^10}|{'Right':>10}|")


# Padding numbers

print(f"Number with padding: {42:05}")  # Output: 00042
```

---

2. Taking User Input and Type Casting

Taking input and converting it to the desired type is crucial for numeric or boolean operations.

Examples:

```
# Taking a number as input

num = input("Enter a number: ")  # Input is always a string

num = float(num)  # Convert to a float
```

```
print(f"The square of the number is {num ** 2}")


# Type casting a boolean
response = input("Do you want to continue? (yes/no): ")
is_continue = response.lower() == "yes"  # Convert to boolean
print(f"Continue: {is_continue}")
```

Edge Cases:
Always handle invalid input gracefully using exception handling.

```
try:
    num = int(input("Enter an integer: "))
    print(f"You entered: {num}")
except ValueError:
    print("Invalid input! Please enter a valid integer.")
```

---

3. Multiple Inputs from Users

Taking multiple inputs in one line is common in competitive programming or quick scripts.

Examples:

```
# Taking space-separated inputs
x, y, z = input("Enter three space-separated numbers: ").split()
x, y, z = int(x), int(y), int(z)  # Convert to integers
print(f"Sum: {x + y + z}")


# Taking comma-separated inputs
values = input("Enter comma-separated values: ").split(",")
print(f"Values: {values}")
```

Using Loops to Handle Multiple Inputs:

```
# Accepting a list of numbers
numbers = list(map(int, input("Enter space-separated numbers: ").split()))
print(f"Numbers entered: {numbers}")
print(f"Sum: {sum(numbers)}")
```

Error Handling with Multiple Inputs:

```
try:
    x, y = map(int, input("Enter two integers separated by space: ").split())
    print(f"Product: {x * y}")
except ValueError:
    print("Error: Please enter valid integers.")
```