Programming with Python & Java (CS 29008)

Lab 1

Basic Python Programming

KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY (KIIT)
Deemed to be University U/S 3 of UGC Act, 1956

School of Electronics Engineering
KIIT Deemed to be University
Bhubeneswar, Odisha

Contents

1 Introduction

Python, one of the most popular programming languages today with origins and evolution as :

- **Conception and Early Development:**- Python was conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands. It emerged as a successor to the ABC programming language, which was inspired by SETL. Guido van Rossum initiated Python's implementation in December 1989. Python aimed to be capable of exception handling and interfacing with the Amoeba operating system.
- **Name and Influences::**- The name "Python" was inspired by the BBC TV show Monty Python's Flying Circus. Guido van Rossum, Python's principal author, played a central role in shaping its direction and was affectionately known as the Benevolent Dictator for Life (BDFL) by the Python community.
- **Versions**- Python 1.0 was released in January 1994, introducing features like lambda, map, filter, and reduce for functional programming. Van Rossum continued Python's development at the Corporation for National Research Initiatives (CNRI) in Virginia. Python 3.0, a major, backwards-incompatible release, arrived in December 2008 after extensive testing.
- **Legacy and Impact:** Python's simplicity, readability, and versatility have made it a favorite among developers. Its adoption spans web development, data science, automation, and more. Guido van Rossum's legacy endures, even though he stepped down as Python's leader in July 2018.

2 Getting Started with Python

In Windows, there are several methods to run Python programs. Here are the most common ones:

Local Options

1. **Using Command Prompt (CMD):**
    - Open Command Prompt by pressing `Win + R`, typing `cmd`, and pressing Enter.

- Navigate to the directory where your Python script is located using the `cd` command.
- Run the Python script by typing `python script_name.py` and pressing Enter.

2. **Using Python IDLE (Integrated Development and Learning Environment):**
   - Open Python IDLE by searching for "IDLE" in the Windows search bar.
   - Once IDLE is open, go to `File > Open` and select your Python script.
   - Press `F5` or go to `Run > Run Module` to execute the script.

Cloud Options

1. **Google Colab:**
   - Access Google Colab by visiting `https://colab.research.google.com`.
   - Create new notebooks or upload existing ones, write Python code, and execute it directly in the browser.
   - Google Colab provides access to GPU and TPU hardware accelerators, making it suitable for machine learning and data analysis tasks.

2. **Jupyter Notebooks on Cloud Platforms:**
   - Many cloud platforms, such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP), offer services for running Jupyter Notebooks in the cloud.
   - These services provide virtual machines (VMs) or managed environments where you can create and run Jupyter Notebooks.

3 Using Google Colab

- Step 1: Access Google Colab

  Open a web browser and navigate to the Google Colab website " https://colab.research.google.com " as shown in Fig. 1.

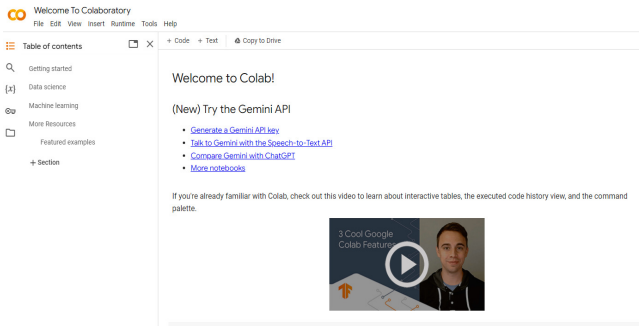

Figure 1: Google Colab Noteboook

- Step 2: Sign in with Google Account If you have a Google account, sign in. If not, create a Google account.
- Step 3: Create a New Notebook

  Click on the "New Notebook" button to create a new Python notebook (refer Fig. 2).

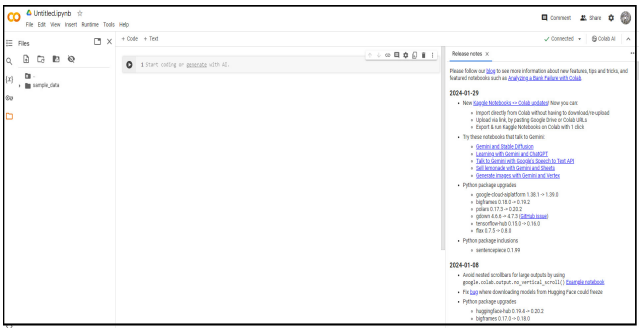

Figure 2: Creation of new notebook

- Step 4: Select the cloud resource i.e. GPU or CPU using runtime tab as shown in Fig. 3. You can optionally give your notebook a name by clicking on "Untitled" at the top of the page and entering a name.
- Step 5: Write Your Python Code

  In the first cell of the notebook, you can start writing your Python code. You can write multiple cells of code in the same notebook.
- Step 6: Run Code Cells

  To execute the code in a cell, click on the play button on the left side of the cell, or press "Shift + Enter" on your keyboard as shown in Fig. 4. The

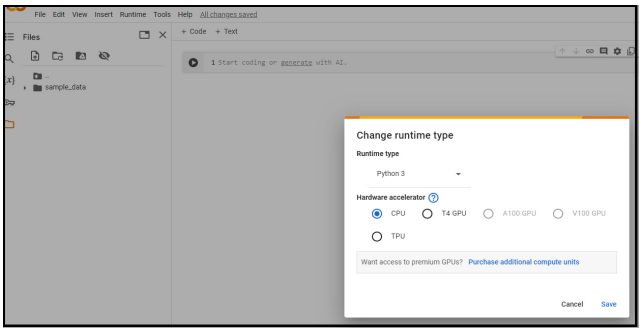

Figure 3: Selection of Virtual Machine

code will be executed, and the output (if any) will be displayed below the cell. shown in Fig. 4.

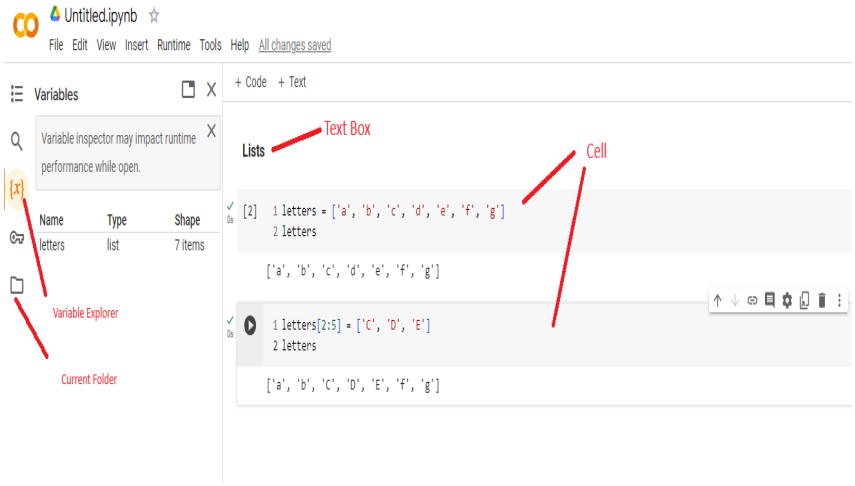

Figure 4: Execution of Python programme

- Step 7: Add Text and Documentation (Optional)

  You can add text and documentation to your notebook by creating text cells. Click on the "+" button in the toolbar to add a new cell, then change the cell type to "Text" using the dropdown menu at the top of the page. You can then type your text or documentation in the cell.

4 Python Programming basics

In Python, objects are classified into two categories based on their mutability: mutable and immutable. Understanding the distinction between mutable and

immutable objects is essential for writing efficient and bug-free code.

- **Immutable Objects:** Immutable objects are objects whose state cannot be modified after they are created. When you modify the value of an immutable object, a new object is created in memory rather than modifying the existing one. Immutable objects in Python include: integers, floats, complex numbers„booleans strings, tuples, frozensets.

```
x = 5   # x is an integer (immutable)
y = "hello"  # y is a string (immutable)
z = (1, 2, 3)  # z is a tuple (immutable)
```

  If you try to modify an immutable object, Python will create a new object with the modified value rather than modifying the original one. For instance:

```
x = 5
y = x  # y references the same object as x
x += 1  # Incrementing x creates a new object with value 6
print(x)  # Output: 6
print(y)  # Output: 5 (y remains unchanged)
```

- **Mutable Objects:** Mutable objects, on the other hand, can be modified after they are created. When you modify the value of a mutable object, the changes are reflected in the original object itself. Mutable objects in Python include: lists, dictionaries, sets, byte arrays, user-defined classes (if they are designed to be mutable)

```
lst = [1, 2, 3]  # lst is a list (mutable)
d = {'a': 1, 'b': 2}  # d is a dictionary (mutable)
s = {1, 2, 3}  # s is a set (mutable)
```

  Modifying mutable objects directly alters their content:

```
lst = [1, 2, 3]
lst.append(4)  # Modifying the list in place
print(lst)  # Output: [1, 2, 3, 4]
d = {'a': 1, 'b': 2}
d['c'] = 3  # Modifying the dictionary in place
print(d)  # Output: {'a': 1, 'b': 2, 'c': 3}
s = {1, 2, 3}
s.add(4)  # Modifying the set in place
print(s)  # Output: {1, 2, 3, 4}
```

Key Differences:

1. Memory Efficiency: Mutable objects may require less memory overhead than immutable objects because they can be modified in place.

2. Safety: Immutable objects provide safety against unintended modifications, which can help prevent bugs in complex programs.
3. Hashability: Immutable objects are hashable and can be used as keys in dictionaries or elements in sets, while mutable objects are not hashable.
4. Performance: Immutable objects tend to be faster for certain operations because they do not require copying when modified.
5. Copying: When you need to create a copy of an object, you must be cautious about whether you want a shallow copy (which shares references to nested objects) or a deep copy (which duplicates nested objects). This distinction is more critical for mutable objects.

The built in data-types of python are shown in Table 1.

Data Type	Example	Properties
Integer	`10`	Whole numbers without decimal points
Float	`3.14`	Numbers with decimal points
String	`"Hello"`	Text or sequence of characters
Boolean	`True` or `False`	sub-class of integer
List	`[1, 2, 3]`	Ordered, mutable sequence
Dictionary	`{'key': 'value'}`	Unordered key-value pairs
Tuple	`(1, 'two', 3.0)`	Ordered, immutable sequence
Set	`{1, 2, 3}`	Unordered collection of unique elements

Table 1: Python Data Types and Properties

4.1 Display output on screen

1. **Using the print() function**

```
"""Write simple Python program to display message on screen"""
# Single line comment (this has no effect on your programme)
>> print("Hello World")
#Separator and End Parameters:
#You can specify the separator between printed items and
#the string to be printed at the end using the sep and end
#parameters, respectively.
>> print("a", "b", "c", sep="-", end="***") # output a-b-c***
#Printing without a Newline:
#By default, print() ends with a newline character (\n).
#You can change this behavior by specifying the end parameter.
>> print("Hello", end="")
>> print("world!")
```

2. **Using formatted strings (f-strings) in Python 3.6 and later versions**

```
>> name = "Alice"
>> age = 30
>> print(f"My name is {name} and I am {age} years old.")
```

3. **Using string concatenation**

```
>> name = "Bob"
>> age = 25
>> print("My name is "+name+" and I am "+str(age) + "years old."
                                        )
```

4. **Using the format() method**

```
>> name = "Charlie"
>> age = 40
>> print("My name is {} and I am {} years old.".format(name, age
                                        ))
```

5. **Using the %-formatting method (old-style formatting)**

```
>> name = "Dave"
>> age = 35
>> print("My name is %s and I am %d years old." % (name, age))
```

6. **Using the sys.stdout.write() function from the sys module**

```
>> import sys
>> sys.stdout.write("Hello, world!\n")
```

7. **Raw String Method** "raw string" refers to a string that treats backslashes as literal characters, rather than as escape characters. You can create a raw string by prefixing the string literal with an r or R.

```
# Define a raw string
raw_string = r'C:\Users\Username\Desktop'
# Print the raw string
print(raw_string)
```

8. **Using file operations**

```
>> with open("output.txt", "w") as f:
     print("Hello, file!", file=f)
#Alternative Method
>> with open("output.txt", "w") as file:
    file.write("This is written to a file.")
```

4.2 Input from User

In Python, there are several ways to take input from the user:

1. **input() function**: This is the simplest way to take input in Python. It reads a line from input (usually user input), converts the line into a string, and returns it.

```
>> user_input = input("Please enter something: ")
>> print("You entered: ", user_input)
```

2. **Command Line Arguments**: You can also take input by passing arguments when you run your script from the command line. The sys module in Python provides a function called argv that can be used to capture command line arguments.

```
>> import sys
>> print("Script Name is: ", sys.argv[0])
>> print("First argument: ", sys.argv[1])
>> print("Second argument: ", sys.argv[2])
```

3. **File Input/Output**: If you have a file and you want to read data from it, you can use Python's built-in open() function.

```
>> with open('myfile.txt', 'r') as f:
       data = f.read()
>> print(data)
```

4.3 Operators

4.3.1 Arithmetic Operators

These are used to perform mathematical operations.

```
>> a = 10
>> b = 20
>> print(a + b)  # Addition: Outputs 30
>> print(a - b)  # Subtraction: Outputs -10
>> print(a * b)  # Multiplication: Outputs 200
>> print(a / b)  # Division: Outputs 0.5
>> print(a % b)  # Modulus: Outputs 10
>> print(a ** b) # Exponentiation: Outputs 100000000000000000000
>> print(a // b) # Floor division: Outputs 0
```

4.3.2 Comparison Operators

These are used to compare values

```
>> a = 10
>> b = 20
>> print(a == b)  # Equal to: Outputs False
>> print(a != b)  # Not equal to: Outputs True
>> print(a > b)   # Greater than: Outputs False
>> print(a < b)   # Less than: Outputs True
>> print(a >= b)  # Greater than or equal to: Outputs False
>> print(a <= b)  # Less than or equal to: Outputs True
```

4.3.3 Logical Operators

These are used to combine conditional statements.

```
>> a = True
>> b = False
>> print(a and b)  # Logical AND: Outputs False
>> print(a or b)   # Logical OR: Outputs True
>> print(not a)    # Logical NOT: Outputs False
```

4.3.4 Bitwise Operators

These are used to compare binary numbers

```
>> a = 10  # binary: 1010
>> b = 4   # binary: 0100
>> print(a & b)  # Bitwise AND: Outputs 0
>> print(a | b)  # Bitwise OR: Outputs 14
>> print(~a)     # Bitwise NOT: Outputs -11
>> print(a ^ b)  # Bitwise XOR: Outputs 14
>> print(a >> 2) # Bitwise right shift: Outputs 2
>> print(a << 2) # Bitwise left shift: Outputs 40
```

4.3.5 Assignment Operators

These are used to assign values to variables.

```
>> a = 10
>> a += 10  # Equivalent to a = a + 10: a becomes 20
>> a -= 10  # Equivalent to a = a - 10: a becomes 10
>> a *= 10  # Equivalent to a = a * 10: a becomes 100
>> a /= 10  # Equivalent to a = a / 10: a becomes 10.0
>> a %= 10  # Equivalent to a = a % 10: a becomes 0.0
>> a **= 10 # Equivalent to a = a ** 10: a becomes 0.0
```

```
>> a //= 10 # Equivalent to a = a // 10: a becomes 0.0
```

4.3.6 Membership Operators

These are used to test whether a value or variable is in a sequence.

```
>> a = [1, 2, 3, 4, 5]
>> print(1 in a)    # Outputs True
>> print(6 not in a)# Outputs True
```

4.3.7 Identity Operators

These are used to compare the memory locations of two objects.

```
>> a = 10
>> b = 10
>> print(a is b)    # Outputs True
>> print(a is not b)# Outputs False
```

4.4 Conditional statements

4.4.1 'if' statement

The if statement is used for decision making. It runs the block of code if the condition is true.

```
a = 10
if a > 0:
    print("a is positive")
```

4.4.2 'if . . . else' statement

If-Else Statement: The if-else statement provides an alternative choice when the if condition is false.

```
a = -10
if a > 0:
    print("a is positive")
else:
    print("a is not positive")
```

4.4.3 Elif Statement

The elif statement allows you to check multiple expressions for truth and execute a block of code as soon as one of the conditions is true.

```
a = 0
if a > 0:
    print("a is positive")
elif a < 0:
    print("a is negative")
else:
    print("a is zero")
```

4.4.4 Nested 'if' statement

You can use one if or else if statement inside another if or else if statement(s). This is called nesting in computer programming.

```
 a = 10
if a >= 0:
    if a == 0:
        print("a is zero")
    else:
        print("a is positive")
else:
    print("a is negative")
```

4.4.5 Miscellaneous

Ternary Operator: This is a shorthand version of the if-else statement. It allows you to test a condition in a single line replacing the multi-line if-else making the code compact.

```
a = 10
print("a is positive") if a > 0 else print("a is not positive")
```

The pass Statement: if statements cannot be empty, but if for some reason you have an if statement with no content, put in the pass statement to avoid getting an error.

```
a = 10
if a > 0:
    pass
```

4.5 Looping statements

4.5.1 For Loop

The for loop in Python is used to iterate over a sequence (like a list, tuple, string, or range) or other iterable objects. Iterating over a sequence is called traversal.

```
# Iterating over a list
for i in [1, 2, 3, 4, 5]:
    print(i)
```

4.5.2 While Loop

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

```
# Iterating until a condition is met
i = 1
while i <= 5:
    print(i)
    i += 1
```

4.5.3 Nested Loops

Python programming language allows the usage of one loop inside another loop. A loop inside another loop is called a nested loop.

```
# Nested loop
for i in range(1, 3):
    for j in range(1, 3):
        print(i, j)
```

4.5.4 Loop Control Statements

Loop control statements change the execution from its normal sequence. Python supports the following control statements.
Break Statement: It terminates the current loop and resumes execution at the next statement. Python

```
for i in [1, 2, 3, 4, 5]:
    if i == 3:
        break
    print(i)
```

Continue Statement: The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

```
 for i in [1, 2, 3, 4, 5]:
    if i == 3:
        continue
    print(i)
```

Pass Statement: The pass statement is a null operation; nothing happens when it executes. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed.

```
for i in [1, 2, 3, 4, 5]:
    pass
```

4.6 Built in Data Type

The various built-in data type in python are as follows

4.6.1 String

Creation of String
Strings in Python can be created using single (' '), double (" "), or triple (''' ''' or """ """) quotes.

```
>>string1 = 'Hello, World!'
>> string2 = "Python Programming"
>> string3 = '''This is a
multi-line
string.'''
```

Accessing
Individual characters in a string can be accessed using indexing.

```
s = "Python"
print(s[0])     # Positive Indexing \& Output: 'P'
print(s[2])     # Positive Indexing \ &Output: 't'
print(s[-1])    #  Negative Indexing \& Output: 'n' (Negative
                                    indexing starts from the end)
```

Slicing
You can extract a portion of a string using slicing.

```
s = "Python Programming"
print(s[0:6])   # Output: 'Python'
print(s[7:])    # Output: 'Programming'
print(s[:6])    # Output: 'Python'
```

```
#Stride (Step) in Slicing
my_string = "Python"
every_second_char = my_string[::2]   # 'Pto'
reverse_string = my_string[::-1]   # 'nohtyP' (reverse the string)
```

String Methods

Python provides a variety of string methods for various operations.

```
text = "python programming"
#Capitalizes the first character of the string
print(text.capitalize())  # Output:  Python programming
# Capitalizes the first letter of each word in the string.
print(text.title())  # Output: Python Programming
# Convert to lowercase and uppercase
print(text.lower())    # Output: 'python programming'
print(text.upper())    # Output: 'PYTHON PROGRAMMING'
# Replace substring
new_text = text.replace("Programming", "Code")
print(new_text)    # Output: 'Python Code'
# Check if a substring exists
print('Python' in text)    # Output: True
#Using find() or index()
my_string = "Python"
index_t = my_string.find('t')   # 2
index_o = my_string.index('o')   # 4
# Using split()
string = "hello world"
print(string.split())  # Output: ['hello', 'world']
# Using Join()
words = ['hello', 'world']
print(' '.join(words))  # Output: "hello world"
# Using startswith() and endswith()
string = "hello world"
print(string.startswith("hello"))  # Output: True
print(string.endswith("world"))    # Output: True
#Remove leading and trailing whitespace from the string.
string = "  hello world  "
print(string.strip())   # Output: "hello world"
print(string.lstrip())  # Output: "hello world  "
print(string.rstrip())  # Output: "  hello world"
```

4.6.2 List

In Python, a list is a versatile and commonly used data structure that stores a collection of items that are ordered and changeable. Lists are **mutable**, meaning they can be modified after creation, and they can contain elements of different data types, including integers, floats, strings, and even other lists.

- **Creation of List** - Lists are defined using square brackets [ ] and individual elements are separated by commas.

```
my_list = [1, 2, 3, "apple", "banana"]
list =[] # Empty List
```

Operation on List: The different operations on the list are as follows

- **Access Items**: You can access the list items by referring to the index number. Python uses zero-based indexing.

```
print(my_list[0])  # Outputs: 1 Positive indexing
print(my_list[-1]) # Outputs: banana (negative indexing starts
                                    from the end)
# Slicing
sliced_list = my_list[1:4]  # Elements from index 1 to 3
print(sliced_list)  # Output: [ 2, 3, "apple"]
#Omitting Start or Stop:
print(my_list[:3])   # Elements from beginning to index 2
print(my_list[2:])   # Elements from index 2 to the end
#Negative Indices:
print(my_list[-3:])  # Last three elements of the list
#Stepping:You can specify a step value, which determines
#the increment between elements in the slice.
print(my_list[::2])  # Every second element of the list
#Reversing a List
#Slicing with a negative step value can reverse a list.
print(my_list[::-1])  # Reversed list
#Slicing can be used to make a shallow copy of a list.
new_list = my_list[:]  # Copy the entire list
```

- **Change Item Value**: Lists are mutable, which means you can change their content.

```
my_list[1] = "mango"
print(my_list)  # Outputs: [1, 'mango', 3, 'apple', 'banana']
my_list[1:3] = [8, 9]  # Replace elements at index 1 and 2
```

- **Add Items:** You can use the append() method to add an item at the end of the list. Use the insert() method to add an item at a specific position.

```
my_list.append("cherry")
print(my_list)  # Outputs: [1, 'mango', 3, 'apple', 'banana', '
                                    cherry']
my_list.insert(1, "orange")
print(my_list)  # Outputs: [1, 'orange', 'mango', 3, 'apple', '
                                    banana', 'cherry']
```

- **Remove Items**: The remove() method removes the specified item. The pop() method removes the specified index. If you do not specify the index, the pop() method removes the last item.

```
my_list.remove("mango")# Removes the first occurrence of "mango"
print(my_list)  # Outputs: [1, 'orange', 3, 'apple', 'banana', '
                                    cherry']
my_list.pop(1) ## Removes element at index 1
print(my_list)  # Outputs: [1, 3, 'apple', 'banana', 'cherry']
```

- **Sort List:** The sort() method sorts the list ascending by default. You can also make it descending by making reverse=True.

```
my_list.sort()
print(my_list)  # Outputs: [1, 3, 'apple', 'banana', 'cherry']
# It will give error as both are of different data type.
my_list.sort(reverse=True)
print(my_list)  # Outputs: ['cherry', 'banana', 'apple', 3, 1]
```

- **Extends List:** You can extend a list by appending elements from another list using the extend() method.

```
another_list = [7, 8, 9]
my_list.extend(another_list)
# Alternatively you can use concatenation operator
x = [1,2,3]
y = x + [1,2,3]
```

- **Unpack List**: Only when we know how many elements are in list.

```
x,y,z = [1,2,3] # Now x is 1, y is 2 \& z is 3
_,y = [1,2] # Underscore for a value which is thrown away
```

- Conversion of numeric datatype to list

```
A =102 # A is integer data type
B= List(A) # B is List data type
A="Rohan" # A 's data type is string
B = List(A) # B's data type is list
```

- **Advanced List**

```
 a = [1,2,3,4,5]
b = [5,4,3,6,1]
print(zip(a,b))
print(list(zip(a,b)))
```

4.6.3 Tuples

A tuple is a collection of objects which are **ordered** and **immutable.** Tuples are sequences, just like lists. The main difference between the tuples and the lists is that the tuples cannot be changed unlike lists. Tuples use parentheses (optional), whereas lists use square brackets.
Applications - Tuples are commonly used in scenarios where immutable sequences of data are required, such as

- Storing coordinates (x, y) or RGB color values.
- Returning multiple values from a function.
- Representing fixed collections of related data, like days of the week or months of the year.
- **Create Tuple**:

```
# Creating a tuple
t = () # empty tuple
one_element_tuple = (42, ) # you need the comma!
my_tuple = ('apple', 'banana', 'cherry')
print(my_tuple)  # Output: ('apple', 'banana', 'cherry')
```

- **Access Tuple**:

```
# Accessing elements from the tuple
print(my_tuple[0])  # Output: 'apple'
print(my_tuple[1])  # Output: 'banana'
print(my_tuple[-1])  # Output: 'cherry'
```

- **Counting and finding the elements in Tuple**:

```
my_tuple = (1, 2, 3, 2, 4, 2)
count_of_twos = my_tuple.count(2)
print(f"Number of twos in the tuple: {count_of_twos}")
# Output: Number of twos in the tuple: 3
my_tuple = ('apple', 'banana', 'cherry', 'banana')
banana_index = my_tuple.index('banana')
print(f"Index of 'banana': {banana_index}")
# Output: Index of 'banana': 1
```

- **Tuple Packing and Unpacking:** Tuples support packing and unpacking, which allows multiple values to be assigned or returned together.

```
# Tuple packing
my_tuple = 1, 2, 3, 'a', 'b', 'c'
# Tuple unpacking
a, b, c, d, e, f = my_tuple
print(a, b, c)  # Output: 1 2 3
```

- **Iterate over tuple**

```
my_tuple = (1, 2, 3, 'a', 'b', 'c')
for item in my_tuple:
    print(item)
```

- **Update Tuple**: Tuples are immutable, which means you can't add or remove items after the tuple is defined. But, you can concatenate or merge two tuples.

```
# Concatenating two tuples
my_tuple = my_tuple + ('dragonfruit',)
print(my_tuple)  # Output: ('apple', 'banana', 'cherry', '
                                    dragonfruit')
```

- **List to tuple**:

```
A = [1,2,3]
tuple(A) # output is tuple (1,2,3)
```

4.6.4 Set

A set is an **unordered** collection of of unique elements. Sets are mutable, meaning they can be modified after creation, but they are not indexed, and their elements are not stored in any particular order. Sets are denoted by enclosing the elements within curly braces .

- **Create Set**:

```
# Creating a set
my_set = {'apple', 'banana', 'cherry'}
print(my_set)  # Output: {'apple', 'banana', 'cherry'}
# Empty set
empty_set = set()
```

- **Access Set**: Sets are unordered, so you cannot access items using an index like lists. However, you can loop through the set items using a for loop, or ask if a specified value is present in a set by using the in keyword.

```
my_set = {'apple', 'banana', 'cherry'}
# Check if 'apple' is present in the set
print('apple' in my_set)  # Output: True
```

- **Update Set**:

```
# Adding an item to the set
my_set.add('dragonfruit')
print(my_set)  # Output: {'apple', 'banana', 'cherry', '
                                    dragonfruit'}
# Removing an item from the set
# It raises a key error if element is not present
my_set.remove('banana')
print(my_set)  # Output: {'apple', 'cherry', 'dragonfruit'}
# Use of discard which does not raises a error
my_set.discard('banana')
print(my_set)   # Output: {'apple', 'cherry', 'dragonfruit'}
# pop(): Removes and returns an arbitrary element from the set.
#Raises a Key Error if the set is empty.
element = my_set.pop()
print(element)
#clear(): Removes all elements from the set, making it empty.
my_set.clear()
print(my_set)  # Output: set()
# update(iterable): Adds multiple elements to the set from an
                                    iterable.
my_set = {1, 2, 3}
my_set.update([4, 5, 6])
print(my_set)  # Output: {1, 2, 3, 4, 5, 6}
```

- **Set Operations**
  - **union(set1, set2, ...)**: Returns a new set containing all unique elements from the sets passed as arguments.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_set = set1.union(set2)
print(union_set)  # Output: {1, 2, 3, 4, 5}
```

  - **intersection(set1, set2, ...)**: Returns a new set containing elements that are common to all sets passed as arguments..

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection_set = set1.intersection(set2)
print(intersection_set)  # Output: {3}
```

  - **difference(set1):** Returns a new set containing elements that are present in the original set but not in the specified set.

```
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4}
difference_set = set1.difference(set2)
print(difference_set)  # Output: {1, 2, 5}
```

- **symmetric difference(set1)**: Returns a new set containing elements that are present in either the original set or the specified set, but not in both.

```
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4, 5, 6, 7}
symmetric_difference_set = set1.symmetric_difference(set2)
print(symmetric_difference_set)  # Output: {1, 2, 6, 7}
```

- **superset**:

```
my_set = {1, 2, 3}
superset = {1, 2, 3, 4, 5}
print(my_set.issubset(superset))  # Output: True
```

- **Conversion**:

```
# list to a set
list_set = set([1, 2, 3, 4, 5])
# string to set
string_set = set("hello")
```

4.6.5 Dictionaries

A dictionary is a collection of items that are stored as “key-value” pairs. They are mutable, meaning they can be modified after creation, and they provide an efficient way to store and retrieve data based on keys rather than indices. Dictionaries are denoted by enclosing the key-value pairs within curly braces . Key and value can be of any type.

- **Create Dictionary**:

```
# Empty dictionary
empty_dict = {}
# Dictionary with key-value pairs
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
print(my_dict)  # Output: {'name': 'John', 'age': 30, 'city': '
                                    New York'}
# Dictionary from key-value pairs using dict() constructor
my_dict = dict(name='John', age=30, city='New York')
```

- **Access a dictionary**:

```
# Accessing elements from the dictionary
print(my_dict['name'])  # Output: 'John'
#If a key is not present in the dictionary, it raises a Key
                                    Error.
```

```
#You can also use the get() method to avoid this:
print(my_dict.get('name'))  # Output: John
print(my_dict.get('gender', 'N/A'))  # Output: N/A (default
                                     value)
# keys(): Returns a view object that displays a list of all the
                                     keys in the dictionary.
print(my_dict.keys())  # Output: dict_keys(['name', 'age', 'city
                                     '])
#values(): Returns a view object that displays a list of all the
                                      values
#in the dictionary.
print(my_dict.values())  # Output: dict_values(['John', 30, 'New
                                      York'])
#items(): Returns a view object that displays a list of key-
                                     value tuples in
#the dictionary.
print(my_dict.items())  # Output: dict_items([('name', 'John'),
                                     ('age', 30), ('city', 'New
                                     York')])
#pop(key, default): Removes the key-value pair with the
                                     specified key and returns
#its value. If the key is not found, it returns the default
                                     value.
print(my_dict.pop('age'))  # Output: 30
print(my_dict)  # Output: {'name': 'John', 'city': 'New York'}
#popitem(): Removes and returns the last key-value pair inserted
                                      into
#the dictionary as a tuple.
print(my_dict.popitem())  # Output: ('city', 'New York')
print(my_dict)  # Output: {'name': 'John', 'age': 30}
```

- **Update a dictionary**:

```
# Adding an item to the dictionary
my_dict['job'] = 'Engineer'
print(my_dict)  # Output: {'name': 'John', 'age': 30, 'city': '
                                     New York', 'job': 'Engineer'}
# Updating an item in the dictionary
my_dict['age'] = 31
print(my_dict)  # Output: {'name': 'John', 'age': 31, 'city': '
                                     New York', 'job': 'Engineer'}
#update(dict): Updates the dictionary with the key-value pairs
                                     from the specified dictionary
                                     .
my_dict = {'name': 'John', 'age': 30}
my_dict.update({'city': 'New York', 'country': 'USA'})
print(my_dict)  # Output: {'name': 'John', 'age': 30, 'city': '
                                     New York', 'country': 'USA'}
# Removing an item from the dictionary
del my_dict['job']
```

```
print(my_dict)  # Output: {'name': 'John', 'age': 31, 'city': '
                                    New York'}
#clear(): Removes all key-value pairs from the dictionary,
                                    making it empty.
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
my_dict.clear()
print(my_dict)  # Output: {}
```

- **Looping through Dictionary**:

```
# Loop through keys
my_dict = {'name': 'John', 'age': 30, 'city': 'New York'}
for key in my_dict:
    print(key)
# Loop through values
for value in my_dict.values():
    print(value)
#Loop through key-value pairs
for key, value in my_dict.items():
    print(key, value)
```

4.7 User defined Python function

1. **Function with minimum 2 arguments**

```
def add_numbers(num1, num2):
    result = num1 + num2
    print(result)

# Call the function
add_numbers(3, 5)  # Output: 8
```

2. **Function returning values:** A function in Python can return a value to the caller, using the return statement.

```
def multiply_numbers(num1, num2):
    result = num1 * num2
    return result
# Call the function
product = multiply_numbers(3, 5)
print(product)  # Output: 15
```

Docstring In Python, a docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. It is used to provide documentation about the purpose, usage, and behavior of the module, function, class, or method. Docstrings are enclosed in triple quotes (either

single or double) and can span multiple lines. They serve as a form of inline documentation that can be accessed using various tools, such as Python's built-in help() function, documentation generators like Sphinx, and integrated development environments (IDEs) that offer code introspection features.

```
def greet(name):
    """This function greets the user with the given name.
    Parameters:
    name (str): The name of the user.
    Returns:
    str: A greeting message.    """
    return f"Hello, {name}! Welcome."
print(help(greet))
```

In this example:

- The greet() function has a docstring immediately following its definition. The docstring provides information about what the function does, what parameters it accepts, and what it returns.
- The help() function is used to access the docstring of the greet() function. When executed, it will display the documentation string defined for the function.

4.8 Built in module and User defined module

1. **Built-in modules**: These comes with a library of standard modules. Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. Here's an example of using the math module, which provides mathematical functions:

```
import math
# Use the sqrt function from the math module
print(math.sqrt(16))  # Output: 4.0
#In this example, we first import the math module using the
#import keyword. Then we can use any functions or variables
#defined in that module by #prefixing them with math.
```

2. **User-defined modules:** You can create your own modules in Python. A module is simply a file containing Python definitions and statements. The file name is the module name with the suffix .py added. Let's say you have a file greetings.py with the following content:

```
# greetings.py
def say_hello(name):
    print(f"Hello, {name}!")
```

You can import and use the say hello function from the greetings module like this:

```
import greetings
greetings.say_hello("John")  # Output: Hello, John!
# In this example, greetings.py is a user-defined module.
#We can use the import keyword to import this module into
#another Python script, and then we can use the say hello
#function defined in the greetings module
```

4.9 Advanced Concepts

4.9.1 List comprehension

List comprehension is a concise and elegant way of creating new lists from existing iterables in Python. It consists of square brackets containing an expression followed by one or more for or if clauses. The expression is evaluated for each element of the iterable and the result is added to the new list. For example, suppose we have a list of numbers and we want to create a new list of their squares. We can use list comprehension as follows:

```
# Syntax
new_list = [expression for element in iterable if condition]
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers]
print(squares)  # Output: [1, 4, 9, 16, 25]
# More examples
even_numbers=[x for x in range(5) if x % 2==0]
square_dict = {x : x *x for x in range(5)}
square_set = { x * x for x in [1,-1] }
#  Multiple for
pairs  = [(x,y) for x in range(10)
                for y in range(10)]
increasing_pairs  = [(x,y) for x in range(10)
                for y in range(x+1,10)]
```

Note: A dictionary can also be a comprehension. The general format for this is:

```
dict_variable= {key:value for (key,value) in dictionary.items()}
#A common use for this is to build up an index to symbolic column
                                names.
text = ['col-zero' , 'col-one' , 'col-two' , 'col-three']
```

```
lookup = { key : value for (value,key) in enumerate(text)}
print(lookup)
```

4.9.2 Anonymous function Lambda

We must occasionally "define a function as we go" and apply it immediately. Defining a function without naming it, using it, and then continuing is made possible by Python. It is called anonymous function. Here is an anonymous function equivalent to $f(x) = x^2 + 4$

```
>> lambda x : x**2 + 4
>> (lambda x : x**2 + 4)(10) # Output is 104
# Giving a name to an anonymous function
>> f = lambda x: x + 3
>> f(1) # Output is 4
#  Define anonymous functions with several variables
>> (lambda x, y, z: x + y + z)(1, 2, 3)
>> (lambda x, y, z: x + y + z)('one', ' two', ' three')
# Define default values for parameters
>> (lambda x, y, z = 3: x * y * z)(1, 2)
# Checking if a number is even:
>> is_even = lambda x: x % 2 == 0
>> print(is_even(6))  # Output: True
>> print(is_even(7))  # Output: False
```

4.9.3 Map function

The map() function in Python applies a given function to each item of an iterable (like a list) and returns an iterator that yields the results. It takes two arguments: the function to apply and the iterable to apply it to.

Here's an example of using map() to apply a function to a list of numbers:

```
# Function to double a number
def double(x):
    return x * 2
# List of numbers
numbers = [1, 2, 3, 4, 5]
# Use map to double each number in the list
doubled_numbers = map(double, numbers)
# Convert the iterator to a list
doubled_numbers_list = list(doubled_numbers)
print(doubled_numbers_list)  # Output: [2, 4, 6, 8, 10]
```

```
#Suppose f is a function and [a,b,c] is a list.We want to be able
#to push the function f inside the list and get [f(a),f(b),f(c)].
from math import sin, cos, pi
```

```
x = map(sin, range(1, 10))
xlist = list(x)
print(xlist)
#Converting a list of strings to uppercase:
names = ['alice', 'bob', 'charlie']
uppercase_names = map(str.upper, names)
print(list(uppercase_names))  # Output: ['ALICE', 'BOB', 'CHARLIE']
#Calculating the lengths of strings in a list:
words = ['apple', 'banana', 'orange']
lengths = map(len, words)
print(list(lengths))  # Output: [5, 6, 6]
# Rounding off a list of floating-point numbers:
float_numbers = [3.14, 2.718, 1.618]
rounded_numbers = map(round, float_numbers)
print(list(rounded_numbers))  # Output: [3, 3, 2]
#Applying a custom function to a list of tuples:
def product(pair):
    x, y = pair
    return x * y
pairs = [(1, 2), (3, 4), (5, 6)]
products = map(product, pairs)
print(list(products))  # Output: [2, 12, 30]
#Using map() with multiple iterables:
numbers = [1, 2, 3]
squares = [4, 9, 16]
result = map(lambda x, y: x + y, numbers, squares)
print(list(result))  # Output: [5, 11, 19]
```

4.9.4 Generator

Generators in Python are special functions that can be paused and resumed. They allow you to generate a sequence of values lazily, one at a time, instead of computing them all at once and storing them in memory. Generators are defined using the yield keyword instead of return. Here's an example of a generator function that generates a sequence of even numbers:

```
def even_numbers(n):
    for i in range(n):
        if i % 2 == 0:
            yield i
# Using the generator to print even numbers up to 10
for num in even_numbers(10):
    print(num, end=' ')  # Output: 0 2 4 6 8
```

Using Map and Generators Together:
You can combine map() and generators to apply a function lazily to each item of an iterable, avoiding the need to store the entire result in memory. This can be particularly useful when dealing with large datasets.

```
# Generator function to double numbers
def double_generator(numbers):
    for num in numbers:
        yield num * 2
# List of numbers
numbers = [1, 2, 3, 4, 5]
# Using map with the double generator
doubled_numbers = map(double_generator, [numbers])
# Accessing the values using a loop
for nums in doubled_numbers:
    for num in nums:
        print(num, end=' ')  # Output: 2 4 6 8 10
```

4.9.5 Random Numbers

In Python, the random number generator is a built-in module called random, which provides functions to generate random numbers. These random numbers can be integers, floating-point numbers, or even selections from sequences like lists. The random module uses a pseudorandom number generator (PRNG) to produce random numbers.

```
#Random(): This function returns a random floating-point number
#in the range [0.0, 1.0).
import random
print(random.random())  # Output: a random float between 0.0 and 1.0
#randint(a, b): This function returns a random integer N such
#that a <= N <= b.
import random
print(random.randint(1, 10))  # Output: a random integer between 1
                                  and 10
#uniform(a, b): This function returns a random floating-point
#number N such that a <= N <= b.
import random
print(random.uniform(1.0, 10.0))  # Output: a random float between 1.
                                    0 and 10.0
#randrange([start], stop[, step]): This function returns a randomly
#selected element from the specified range.
import random
print(random.randrange(0, 100, 2))  # Output: a random even number
                                    between 0 and 100
#choice(seq): This function returns a random element from the
#non-empty sequence seq.
import random
my_list = ['apple', 'banana', 'cherry']
print(random.choice(my_list))  # Output: a random element from the
                                  list
#shuffle(seq): This function shuffles the elements of the
```

```
#sequence seq in place.
import random
my_list = [1, 2, 3, 4, 5]
random.shuffle(my_list)
print(my_list)  # Output: a shuffled version of the list
#sample(population, k): This function returns a random sample
#of k elements from the population sequence without replacement.
import random
my_list = ['apple', 'banana', 'cherry', 'date', 'elderberry']
print(random.sample(my_list, k=3))  # Output: a random sample of 3
                                    elements from the list
```

```
#It's important to note that Python's random number generator is
#deterministic, meaning that it produces the same sequence of
#random numbers every time the program runs with the same seed
#value. You can set the seed value using random.seed() to produce
#reproducible results.
import random
random.seed(42)  # Set seed value for reproducibility
print(random.random())  # Output: 0.6394267984578837
```

4.9.6 Use of *args & **kwargs

In Python, *args and **kwargs are used to pass a variable number of arguments to a function.

- Using *args:

  *args is used to pass a variable number of positional arguments to a function. The * symbol unpacks the arguments into a tuple within the function.

```
>> def sum_values(*args):
     total = 0
     for num in args:
        total += num
     return total
>> print(sum_values(1, 2, 3, 4, 5))  # Output: 15
```

- **Using kwargs: **kwargs is used to pass a variable number of keyword arguments to a function. The ** symbol unpacks the keyword arguments into a dictionary within the function.

```
>> def print_info(**kwargs):
     for key, value in kwargs.items():
        print(f"{key}: {value}")
>> print_info(name="Alice", age=30, city="New York")
# Output:
```

```
# name: Alice
# age: 30
# city: New York
```

- **Using both *args and kwargs

```
>> def example_func(*args, **kwargs):
    print("Positional arguments:")
    for arg in args:
        print(arg)
    print("\nKeyword arguments:")
    for key, value in kwargs.items():
        print(f"{key}: {value}")
>> example_func(1, 2, 3, name="Alice", age=30)
# Output:
# Positional arguments:
# 1
# 2
# 3
# Keyword arguments:
# name: Alice
# age: 30
```

4.9.7 default dictionary

In Python, a default dictionary is a subclass of the built-in dictionary (dict) that provides a default value for each key that does not exist in the dictionary. This means that you can specify a default value to be returned when you try to access a key that does not exist, instead of raising a KeyError as a regular dictionary would. The default value can be of any data type, such as int, list, tuple, set, dictionary, etc. The default dictionary is particularly useful when you're working with dictionaries and you want to ensure that accessing non-existent keys doesn't result in errors or requires additional handling. It simplifies code and makes it more concise.

```
from collections import defaultdict
# Example 1: Creating a defaultdict with int as default value
d = defaultdict(int)
d['a'] = 1
d['b'] = 2

print(d['a'])  # Output: 1
print(d['b'])  # Output: 2
print(d['c'])  # Output: 0 (default value for int)

# Example 2: Creating a defaultdict with list as default value
d = defaultdict(list)
```

```
d['a'].append(1)
d['b'].extend([2, 3])

print(d['a'])  # Output: [1]
print(d['b'])  # Output: [2, 3]
print(d['c'])  # Output: [] (default value for list)

# Example 3: Creating a defaultdict with custom default value
def default_value():
    return 'default'

d = defaultdict(default_value)
d['a'] = 'apple'
d['b'] = 'banana'

print(d['a'])  # Output: apple
print(d['b'])  # Output: banana
print(d['c'])  # Output: default (custom default value)
```

In the examples above:

- In Example 1, we create a defaultdict with int as the default value. When accessing the key 'c', which does not exist, it returns 0 (the default value for int).
- In Example 2, we create a defaultdict with list as the default value. When accessing the key 'c', it returns an empty list, which is the default value for list.
- In Example 3, we create a defaultdict with a custom default value function. When accessing the key 'c', it invokes the default value function and returns 'default'.

———————————

4.9.8 Exception handling

In Python, exceptions are used to handle errors that might occur during program execution. The try block is used for risky code that can raise an exception, and the except block handles the error raised in the try block.

```
try:
    numerator = 10
    denominator = 0
    result = numerator / denominator
    print(result)
except:
    print("Error: Denominator cannot be 0.")
```

```
#Output:
#Error: Denominator cannot be 0.
```

Catching Specific Exceptions:
For each try block, there can be zero or more except blocks. Multiple except blocks allow us to handle different exceptions differently.

```
try:
    even_numbers = [2, 4, 6, 8]
    print(even_numbers[5])  # Accessing index 5 (out of bounds)
except ZeroDivisionError:
    print("Denominator cannot be 0.")
except IndexError:
    print("Index Out of Bound.")
#Output:
#Index Out of Bound.
```

Try with Else Clause: Sometimes, we want to run a certain block of code if the try block runs without errors. We can use the optional else keyword with the try statement.

```
try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1 / num
    print(reciprocal)
#Output:
#If we pass an odd number:
#Enter a number: 1
#Not an even number!
#If we pass an even number:
#Enter a number: 4
#0.25
#If the input is even, the reciprocal is computed and displayed.
```

4.9.9 Miscellaneous

- **Operator overloading**: In short, it means that operators such as +, -. , %, and so on, may represent different operations according to the context they are used in. Therefore, the + sign is used to concatenate them. Hence, the sign is used to concatenate the list to itself according to the right operand.

```
a = [1, 2, 3, 4, 5]
b = [4, 5, 6, 7, 8]
```

```
a + b
a *2
c= "Aaru"
c * 2
```

- **Find the information a data type**

```
import sys
sys.float_info
# Dictionary with zip
d = dict(zip('hello', range(5)))
```

Lab 1 Exercises

Objective:
The objective of the Python Programming Lab is to provide students with hands-on experience and practical exposure to the fundamentals of Python programming language. Students will gain understanding of the following fundamentals through the lab:

- Basic data types (Numeric,string,Lists,Dictionary, Tuple,set)
- Decision making (IF/ELSE ,While)
- User Defined Functions
- Using Built in Modules
- Advanced - Lambda ,List comprehension,Filter, Map, generators

Outcome

- To understand the fundamentals of Python programming concepts and its applications

Lab Assignments

1. Given two lists—one containing keys and the other containing values. Create a dictionary by pairing corresponding elements from both lists.
2. Write a program that compares two numbers and prints whether they are equal, not equal, greater than, or less than each other.
3. Create a list of length 10 of your choice containing multiple types of data. Using for loop print the element and its data type.
4. Using a while loop, verify if the number A is purely divisible by number B and if so then how many times it can be divisible.
5. Create a list containing 25 integer type data. Using for loop and if-else condition print if the element is divisible by 3 or not.
6. Given a list my list = [1, 2, 3, 4, 5], write the code to slice the list and obtain the sub-list [2, 3].
7. Given the string "Hello, World!", extract the sub-string "World".
8. Calculate the multiplication and sum of two numbers: Given two integer numbers, return their product only if the product is equal to or lower than 1000. Otherwise, return their sum.

9. Write a program to iterate through the first 10 numbers, and in each Print characters from a string that are present at an even index number. Accept a string from the user and display characters that are present at an even index number.