Own Module Creation Using Python

1. Introduction to Python Modules

A **module** in Python is simply a file containing Python definitions and statements. The filename is the module name with the suffix .py. Modules allow you to logically organize your Python code, making it reusable, easier to manage, and understandable.

- **Concept:** Modules are analogous to libraries or header files in other programming languages.
- **Purpose:**
    - **Reusability:** Functions, classes, and variables defined in a module can be reused across different programs.
    - **Namespacing:** Prevents confusion between functions/variables with the same name (e.g., math.sqrt() vs. my_module.sqrt()).
    - **Organization:** Keeps related code together.

2. Steps to Create a Python Module

Creating your own module is a straightforward process:

Step 1: Write the Python Code

Create a new Python file (.py) and define the functions, classes, or variables you want to include in your module.

- **Example File:** bca.py

```
# greetings.py module file
def add(a, b):
    c = a+b
    return c

def sub(a, b):
    c = a-b
    return c

pi = 3.14159
```

Step 2: Save the File

Save the file with a descriptive name (e.g., bca.py). The module name will be **bca**.

Step 3: Import the Module

To use the module, save a separate Python script (e.g., main_app.py) in the **same directory** as the module file, and use the import statement.

- **Example Usage File:** main_app.py

```
import bca  # Imports the entire module

c= bca.add(10,5)
print(“Addition: “,c)
c= bca.sub(10,5)
print(“Subtraction: “,c)

print("PI is approximately: ", pi)
```

Step 4: Run the Application

When main_app.py is executed, the greetings.py module is loaded, and its defined members become available.

3. Benefits for BCA Students (Application Focus)

Understanding and using modules is crucial for professional software development, which is a key BCA objective:

- **Team Collaboration:** Different team members can work on separate module files simultaneously.
- **Modular Programming:** Promotes breaking down large problems into smaller, manageable components (modules), adhering to the principles of good software design.
- **Maintenance & Debugging:** Bugs can be isolated and fixed within a specific module without affecting the rest of the application.
- **Code Sharing:** Modules can be packaged and shared with other developers (e.g., via the Python Package Index, **PyPI**).

Python Lists

In Python, a list is a built-in data structure that can hold an ordered collection of items. Unlike arrays in some languages, Python lists are very flexible:

- Can contain duplicate items
- **Mutable:** items can be modified, replaced, or removed
- **Ordered:** maintains the order in which items are added
- **Index-based:** items are accessed using their position (starting from 0)
- Can store mixed data types (integers, strings, booleans, even other lists)

1. Creating a List

Lists can be created in several ways, such as using square brackets, the list() constructor or by repeating elements. Let's look at each method one by one with example:

Program

```
a = [1, 2, 3, 4, 5] # List of integers

b = ['apple', 'banana', 'cherry'] # List of strings

c = [1, 'hello', 3.14, True] # Mixed data types

print(a)

print(b)

print(c)
```

Output

```
[1, 2, 3, 4, 5]

['apple', 'banana', 'cherry']

[1, 'hello', 3.14, True]
```

2. Accessing List Elements

Elements in a list are accessed using indexing. Python indexes start at 0, so a[0] gives the first element. Negative indexes allow access from the end (e.g., -1 gives the last element).

Program

```
fruit = ['apple', 'banana', 'cherry']
print(fruit [0])
print(fruit [1])
print(fruit[1:2])
```

Output

```
['apple']

['banana']

['banana', 'cherry']
```

3. Adding Elements into List

We can add elements to a list using the following methods:

- **append():** Adds an element at the end of the list.
- **insert():** Adds an element at a specific position.

Program

```
fruit = ['apple', 'banana', 'cherry']
fruit.append(“orange”)

print("After append:", fruit)

fruit.insert(1, “grape”)

print("After insert:", fruit)
```

Output

```
After append:  ['apple', 'banana', 'cherry', ‘orange’]

After insert: ['apple', ’grape’, 'banana', 'cherry', ‘orange’]
```

4. Removing Elements from List

We can remove elements from a list using:

- **remove():** Removes the first occurrence of an element.
- **pop():** Removes the element at a specific index or the last element if no index is specified.

Program

```
fruit = ['apple', 'banana', 'cherry']
print(fruit.pop())

print(fruit.remove(‘apple’))
```

Output

```
['apple', 'banana']

['banana']
```

Python-Functions

Introduction

A **function** in Python is a block of organized, reusable code that performs a specific task. Functions make programs easier to read, test, and maintain.

Types of Functions

1. **Built-in Functions**
   - Already provided by Python.
   - Examples: print(), len(), type(), sum(), max(), etc.
2. **User-defined Functions**
   - Created by the user using the def keyword.
   - Example:
   - def greet():
   -     print("Hello, Welcome to Python!")
   - greet()

Syntax of a Function

```
def function_name(parameters):
   """Optional docstring"""
   # Function body
   statement(s)
   return value
```

Explanation:

- def – keyword to define a function
- function_name – name of the function
- parameters – data passed to the function (optional)
- return – used to send back a result (optional)

Example 1: Function with Parameters

```
def add(a, b):
   c=a+b
   return c
def sub(a, b):
   c=a-b
   return c


result = add(10, 5)
print("Add:", result)
result = sub(10, 5)
print("Sub:", result)
```

Output:

```
Add: 15
Sub: 5
```

Example 2: Function with Default Argument

```
def greet(name="User"):
    print("Hello", name)

greet()
greet("Bharathidasan")
```

Output:

```
Hello User
Hello Bharathidasan
```

<u>Pattern Program</u>

1. Square Pattern

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Program

```
n = int(input("Enter the number of rows: "))
for i in range(n):
    for j in range(n):
        print("*", end=" ")
    print()
```

2. Triangle Pattern

```
*
* *
* * *
* * * *
* * * * *
```

Program

```
n = int(input("Enter the number of rows: "))
for i in range(1, n + 1):
    for j in range(i):
        print("*", end=" ")
    print()
```

3. Diamond pattern

```
    *
   * *
  * * *
 * * * *
* * * * *
 * * * *
  * * *
   * *
    *
```

Program

```
n = int(input("Enter the number of rows: "))
# Upper half of diamond
for i in range(1, n + 1):
    print(" " * (n - i) + "* " * i)
# Lower half of diamond
for i in range(n - 1, 0, -1):
    print(" " * (n - i) + "* " * i)
```

4. Left Arrow Pattern

```
* * * * *
 * * * * *
  * * * * *
   * * * * *
    * * * * *
   * * * * *
  * * * * *
 * * * * *
* * * * *
```

Program

```
n = int(input("Enter the number of rows: "))
# Upper half of the arrow
for i in range(n, 0, -1):
    print(" " * (n - i) + "* " * n)
# Lower half of the arrow
for i in range(2, n + 1):
    print(" " * (n - i) + "* " * n)
```

5. Right Arrow Pattern

```
   * * * * *
  * * * * *
 * * * * *
* * * * *
* * * * *
 * * * * *
  * * * * *
   * * * * *
    * * * * *
```

Program

```
n = int(input("Enter the number of rows: "))
# Upper half of the arrow
for i in range(2, n + 1):
    print(" " * (n - i) + "* " * n)
# Lower half of the arrow
for i in range(n, 0, -1):
    print(" " * (n - i) + "* " * n)
```

OOPS in Python

1. Class

- A **class** is a **blueprint** for creating objects.
- It defines attributes and methods that describe an object’s behavior.

Example:

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print("Name:", self.name)
        print("Age:", self.age)
```

---

2. Object

- An **object** is an **instance of a class**.
- It represents a real-world entity.

```
s1 = Student("Arthi", 20)
s1.display()
```

---

3. Inheritance

- Inheritance allows one class to **acquire properties and methods** of another class.
- Promotes **code reusability**.

Example:

```
class Person:
    def show(self):
        print("This is a person")

class Student(Person):
    def study(self):
        print("This is a student")

s = Student()
s.show()
s.study()
```

---

4. Polymorphism

- **Polymorphism** means **one name with many forms**.

- Same method behaves differently depending on the object.

Example:

```
class Dog:
    def sound(self):
        print("Barks")

class Cat:
    def sound(self):
        print("Meows")

for animal in (Dog(), Cat()):
    animal.sound()
```

---

5. Encapsulation

- Binding **data and methods** into a single unit (class).
- Protects data using **private members** (using _ or __).

Example:

```
class Account:
    def __init__(self, balance):
        self.__balance = balance

    def show_balance(self):
        print("Balance:", self.__balance)
```

---

6. Abstraction

- Hiding complex details and showing only the necessary features.
- Implemented using **abstract classes or interfaces** (using `abc` module).

Example:

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def area(self):
        print("Area of circle")
c = Circle()
c.area()
```

Python Classes and Objects

1. Introduction

- Python is an **object-oriented programming (OOP)** language.
- **Class** and **Object** are the main concepts of OOP.
- A **class** defines a blueprint for creating objects.
- An **object** is an instance of a class containing both **data (attributes)** and **functions (methods)**.

---

2. Defining a Class

A class is defined using the `class` keyword.

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print("Name:", self.name)
        print("Age:", self.age)
```

- `__init__()` → Constructor method, called automatically when an object is created.
- `self` → Refers to the current instance of the class.

---

3. Creating Objects

Objects are instances of a class.

```
s1 = Student("Arthi", 20)
s2 = Student("Bharath", 21)

s1.display()
s2.display()
```

Output:

```
Name: Arthi
Age: 20
Name: Bharath
Age: 21
```

---

4. Class and Instance Variables

- **Instance variables:** Belong to each object (e.g., `self.name`).
- **Class variables:** Shared by all objects in the class.

```
class Student:
    college = "Takshashila University"  # Class variable
    def __init__(self, name):
        self.name = name  # Instance variable

s1 = Student("Arthi")
print(s1.name, Student.college)
```

---

5. Importance of Classes and Objects

- Promotes **code reusability** through inheritance.
- Provides **data encapsulation** and security.
- Makes programs **modular and easy to maintain**.
- Represents **real-world entities** in programming.

---

6. Example Program

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def start(self):
        print(self.brand, self.model, "is starting...")

car1 = Car("Toyota", "Innova")
car1.start()
```

Output:

```
Toyota Innova is starting...
```

GUI Components in Python

Introduction

GUI (Graphical User Interface) in Python allows users to interact with applications through graphical elements such as buttons, labels, text boxes, and menus instead of text commands. Python provides several libraries for GUI programming, the most commonly used being **Tkinter**.

---

1. Tkinter Overview

- **Tkinter** is the standard GUI library that comes bundled with Python.
- It provides a fast and easy way to create GUI applications.
- GUI components in Tkinter are called **widgets**.

To use Tkinter:

```
from tkinter import *
root = Tk()
root.mainloop()
```

---

2. Common GUI Components (Widgets)

Widget	Description	Example Code
Label	Displays text or images.	Label(root, text="Hello").pack()
Button	Performs an action when clicked.	Button(root, text="Click Me").pack()
Entry	Single-line text input field.	Entry(root).pack()
Text	Multi-line text input area.	Text(root).pack()
Checkbutton	Used for on/off options.	Checkbutton(root, text="Accept").pack()
Radiobutton	Select one option from a set.	Radiobutton(root, text="Male").pack()
Listbox	Displays a list of selectable items.	Listbox(root).pack()
Frame	Container widget for grouping elements.	Frame(root).pack()
Canvas	Used for drawing shapes and images.	Canvas(root, width=200, height=100).pack()
Menu	Adds menu bar with commands.	Menu(root)

---

3. Geometry Management

Tkinter provides three geometry managers:

1. **pack()** – Organizes widgets in blocks.
2. **grid()** – Organizes widgets in a tabular (row/column) structure.

3. **place()** – Positions widgets at specific coordinates.

---

4. Event Handling

- Events are user actions like mouse clicks or key presses.
- Handled using **command parameter** or **bind()** method.

Example:

```
def showMsg():
    print("Button clicked!")

btn = Button(root, text="Click", command=showMsg)
btn.pack()
```

---

5. Example GUI Program

```
from tkinter import *

def greet():
    lbl.config(text="Hello, " + name.get())

root = Tk()
root.title("Greeting App")

Label(root, text="Enter your name:").pack()
name = Entry(root)
name.pack()
Button(root, text="Greet", command=greet).pack()
lbl = Label(root, text="")
lbl.pack()

root.mainloop()
```

---

6. Advantages of Using Tkinter

- Simple and lightweight.
- Cross-platform (works on Windows, Linux, macOS).
- Integrated with Python, no extra installation required.
- Easy to learn for beginners.

Python MVC (Model – View – Controller)

1. Introduction

- **MVC** stands for **Model–View–Controller**.
- It is a **software design pattern** used to separate an application's **logic**, **data**, and **user interface**.
- MVC improves **code organization**, **reusability**, and **maintainability**.
- Python frameworks like **Django**, **Flask**, and **Tkinter (in GUI)** use the MVC or similar pattern.

---

2. MVC Architecture Overview

Component	Purpose	Example Functionality
Model	Manages data, database, and business logic.	Store, retrieve, and update data.
View	Handles user interface (UI).	Display data to the user.
Controller	Acts as a link between Model and View; controls the flow.	Takes input, processes it, updates Model and View.

---

3. Structure of MVC Pattern

Flow of Control:

1. The **user interacts** with the **View** (UI).
2. The **Controller** receives the input and processes it.
3. The **Model** is updated with the new data.
4. The **View** refreshes and displays the updated data.

---

4. Example of MVC in Python

```
# Model
class StudentModel:
    def __init__(self):
        self.data = {"Arthi": 85, "Bharath": 90}

    def get_marks(self, name):
        return self.data.get(name, "Student not found")

# View
class StudentView:
    def display(self, name, marks):
        print(f"Student: {name}, Marks: {marks}")

# Controller
class StudentController:
    def __init__(self, model, view):
        self.model = model
        self.view = view

    def get_student_info(self, name):
        marks = self.model.get_marks(name)
        self.view.display(name, marks)

# Main
model = StudentModel()
view = StudentView()
controller = StudentController(model, view)
controller.get_student_info("Arthi")
```

Output:

```
Student: Arthi, Marks: 85
```

---

5. Advantages of MVC

- **Separation of Concerns:** Logic, UI, and data are handled separately.
- **Code Reusability:** Components can be reused independently.
- **Ease of Maintenance:** Changing UI doesn't affect business logic.
- **Team Collaboration:** Different developers can work on Model, View, and Controller simultaneously.

---

6. MVC in Python Frameworks

- **Django:** Uses an MVT (Model–View–Template) pattern, similar to MVC.
- **Flask:** Follows a lightweight MVC structure.
- **Tkinter GUI:** Can also be organized using MVC for cleaner code.

Python List

1. Introduction

- A **List** in Python is a **collection of ordered, changeable (mutable), and heterogeneous elements**.
- Lists allow storing multiple values in a single variable.
- Defined using **square brackets `[ ]`**.
- Lists can contain items of **different data types** (integers, strings, floats, etc.).

Example:

```
my_list = [10, "apple", 3.5, True]
print(my_list)
```

Output:

```
[10, 'apple', 3.5, True]
```

---

2. Characteristics of List

- **Ordered:** Maintains insertion order.
- **Mutable:** Elements can be changed or updated.
- **Heterogeneous:** Can store mixed data types.
- **Indexed:** Elements are accessed using index numbers (starting from 0).
- **Dynamic:** Can grow or shrink in size.

---

3. Creating a List

```
# Empty list
a = []

# List with elements
b = [1, 2, 3, 4, 5]
```

---

4. Accessing List Elements

You can access elements using **indexing** or **slicing**.

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])      # apple
print(fruits[-1])     # cherry
print(fruits[0:2])    # ['apple', 'banana']
```

---

5. Updating and Deleting Elements

```
fruits[1] = "orange"   # Update
print(fruits)          # ['apple', 'orange', 'cherry']

del fruits[0]          # Delete
print(fruits)          # ['orange', 'cherry']
```

---

6. List Operations

Operation	Example	Output
Concatenation	[1,2] + [3,4]	[1,2,3,4]
Repetition	[1,2]*2	[1,2,1,2]
Membership	3 in [1,2,3]	True
Length	len([1,2,3])	3

---

7. Common List Methods

Method	Description	Example
append(x)	Adds an item at the end	fruits.append("grape")
insert(i,x)	Inserts item at index *i*	fruits.insert(1,"kiwi")
remove(x)	Removes the first occurrence of *x*	fruits.remove("apple")
pop(i)	Removes element at index *i*	fruits.pop(0)
sort()	Sorts list in ascending order	nums.sort()
reverse()	Reverses the list	nums.reverse()
clear()	Removes all elements	nums.clear()

---

8. Iterating Through a List

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

Output:

```
apple
banana
```

```
cherry
```

---

9. Nested Lists

Lists can contain other lists (2D or multi-dimensional).

```
matrix = [[1,2,3],[4,5,6],[7,8,9]]
print(matrix[1][2])  # Output: 6
```

Python Tuple

1. Introduction

- A **Tuple** in Python is an **ordered and immutable** collection of elements.
- It is similar to a list, but **cannot be modified** (no addition, deletion, or update of elements).
- Tuples are defined using **parentheses `( )`**.
- They can store **heterogeneous data** (different data types).

Example:

```
tup = (10, "apple", 3.5, True)
print(tup)
```

Output:

```
(10, 'apple', 3.5, True)
```

---

2. Characteristics of Tuples

- **Ordered:** Elements have a fixed order.
- **Immutable:** Once created, elements cannot be changed.
- **Heterogeneous:** Can contain different data types.
- **Indexed:** Elements are accessed using index numbers.
- **Faster:** More memory-efficient and faster than lists.

---

3. Creating a Tuple

```
# Empty tuple
a = ()

# Tuple with elements
b = (1, 2, 3, 4)
```

```
# Tuple without parentheses (optional)
c = 1, 2, 3

# Single element tuple (add a comma)
d = (5,)
```

---

4. Accessing Tuple Elements

Tuples are accessed by index or slicing.

```
tup = ("apple", "banana", "cherry")
print(tup[0])      # apple
print(tup[-1])     # cherry
print(tup[0:2])    # ('apple', 'banana')
```

---

5. Tuple Operations

Operation	Example	Output
Concatenation	(1,2)+(3,4)	(1,2,3,4)
Repetition	('a',)*3	('a','a','a')
Membership	'a' in ('a','b')	True
Length	len((1,2,3))	3

---

6. Tuple Functions

Function	Description	Example
len(tup)	Returns number of elements	len((1,2,3)) → 3
max(tup)	Returns maximum value	max((1,5,3)) → 5
min(tup)	Returns minimum value	min((1,5,3)) → 1
sum(tup)	Returns sum of numeric elements	sum((1,2,3)) → 6
tuple(seq)	Converts sequence to tuple	tuple([1,2,3]) → (1,2,3)

---

7. Tuple Packing and Unpacking

```
# Packing
```

```
student = ("Arthi", 20, "BCA")

# Unpacking
name, age, course = student
print(name)
print(age)
print(course)
```

Output:

```
Arthi
20
BCA
```

---

8. Nested Tuples

Tuples can contain other tuples.

```
nested = ((1,2), (3,4), (5,6))
print(nested[1][0])  # Output: 3
```

---

9. Advantages of Tuples

- **Faster** than lists (better performance).
- **Data safety** (cannot be modified accidentally).
- **Can be used as keys** in dictionaries (since tuples are immutable).
- **Useful for fixed data collections**.

Python Set

1. Introduction

- A **Set** in Python is an **unordered**, **unindexed**, and **mutable** collection of unique elements.
- Sets are used to store multiple items in a single variable **without duplicates**.
- Defined using **curly braces** **`{ }`** or the **`set()`** constructor.

Example:

```
my_set = {1, 2, 3, 3, 4}
print(my_set)
```

Output:

```
{1, 2, 3, 4}
```

(Duplicate element `3` is automatically removed.)

2. Characteristics of Sets

- **Unordered:** Elements have no fixed position or index.
- **Mutable:** You can add or remove items.
- **Unique Elements:** Duplicates are automatically removed.
- **Heterogeneous:** Can store different data types (int, str, float, etc.).

3. Creating a Set

```
# Empty set
s = set()

# Set with values
fruits = {"apple", "banana", "cherry"}
```

4. Accessing Set Elements

- Sets do not support **indexing** or **slicing** because they are unordered.
- You can access elements using **loops**.

```
for fruit in fruits:
    print(fruit)
```

5. Adding and Removing Elements

```
fruits.add("orange")       # Add single element
fruits.update(["grape"])   # Add multiple elements
fruits.remove("banana")    # Remove specific item (error if not found)
fruits.discard("apple")    # Remove item (no error if not found)
fruits.pop()               # Remove a random element
```

6. Set Operations

Operation	Operator/Method	Example	Result
Union	`	orunion()`	`{1,2}
Intersection	& or intersection()	{1,2,3} & {2,3,4}	{2,3}
Difference	- or difference()	{1,2,3} - {2}	{1,3}
Symmetric Difference	^ or symmetric_difference()	{1,2} ^ {2,3}	{1,3}

---

7. Example Program

```
a = {1, 2, 3, 4}
b = {3, 4, 5, 6}

print("Union:", a | b)
print("Intersection:", a & b)
print("Difference:", a - b)
```

Output:

```
Union: {1, 2, 3, 4, 5, 6}
Intersection: {3, 4}
Difference: {1, 2}
```

Dictionary in Python

1. Introduction

- A **Dictionary** in Python is an **unordered**, **mutable** collection that stores data as **key–value pairs**.
- Each **key** is unique and used to access its corresponding **value**.
- Dictionaries are defined using **curly braces `{ }`**.

Example:

```
student = {"name": "Arthi", "age": 20, "course": "BCA"}
print(student)
```

Output:

```
{'name': 'Arthi', 'age': 20, 'course': 'BCA'}
```

---

2. Characteristics of Dictionary

- **Unordered:** Items are not stored in a specific order (in older Python versions).
- **Mutable:** Can be changed (add, remove, or modify items).
- **Indexed by Keys:** Keys must be unique and immutable (string, number, tuple).
- **Heterogeneous:** Can store different types of data.

---

3. Creating a Dictionary

```
# Empty dictionary
```

```
d = {}

# Dictionary with data
student = {"name": "Arthi", "age": 20}

# Using dict() constructor
emp = dict(id=101, name="Bharath")
```

---

4. Accessing Dictionary Elements

Access values using **keys**.

```
print(student["name"])     # Output: Arthi
print(student.get("age"))  # Output: 20
```

---

5. Adding, Updating, and Deleting Elements

```
student["course"] = "BCA"          # Add new key-value pair
student["age"] = 21                # Update value
del student["course"]              # Delete key-value pair
student.clear()                    # Remove all item
```

6. Example Program

```
student = {"name": "Arthi", "age": 20, "course": "BCA"}
student["grade"] = "A"

for key, value in student.items():
    print(key, ":", value)
```

Output:

```
name : Arthi
age : 20
course : BCA
grade : A
```