Centre for Differently Abled Persons
Bharathidasan University

III BCA – V SEMESTER

PYTHON PROGRAMMING
(20UCA5CC6)

UNIT – V

Prepared by
Dr. M. Prabavathy
Ms. M. Hemalatha

UNIT - V

Abstract Data Types:

- The abstract data type is special kind of data type, whose behavior is defined by a set of values and set of operations.
- “Abstract” is used as we can use these data types, we can perform different operations. But how those operations are working that is totally hidden from the user.
- The ADT is made of primitive data types, but operation logics are hidden.
- Some examples of ADT are Stack, Queue, List etc.

Classes:

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.
- To create a class, use the keyword class.
  - Example:
    - class MyClass:
    - x = 5

Example:

- class Person:
- def __init__(self, name, age):
  self.name = name
- self.age = age
- p1 = Person("John", 36)print(p1.name) print(p1.age)

- **Create Object:**
  - Now we can use the class named MyClass to createobjects:
  - Example:
    - p1 = MyClass()
    - print(p1.x)
- **The _init_() Function:**
  - All classes have a function called __init__(), which is always executed when the class is being initiated.
  - Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created.

Python Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- Parent class is the class being inherited from, also called base class.
- Child class is the class that inherits from another class, also called derived class.

- **Create a Parent Class:**
    - Any class can be a parent class, so the syntax is the same as creating any other class:

Example

- class Person:
  - def __init__(self, fname, lname): self.firstname = fname
- self.lastname = lname

  def printname(self):
  - print(self.firstname, self.lastname)
- x = Person("John", "Doe") x.printname()

Create a Child Class:

- To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

- Example:

```
class Student(Person):
pass
```

- **Note:** Use the pass keyword when you do not want to add any other properties or methods to the class.

- **Create an object, and then execute the method:**
  - Example:
    - x = Student("Mike",
      "Olsen")
      x.printname()

Encapsulation

- Encapsulation is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.
- In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class.

Example:

```
class Students:
    def __init__(self, name, rank,
        points): self.name = name
self.rank = rank self.points =
    points defdemofunc(self):
    print("I am "+self.name)
print("I got Rank ",+self.rank)

# create 4 objects
st1 = Students("Steve", 1, 100)
st2 = Students("Chris", 2, 90)
```

```
st3 = Students("Mark", 3, 76)
st4 = Students("Kate", 4, 60)

# call the functions using the objects created above
st1.demofunc()
st2.demo
func()
st3.demo
func()
st4.demo
func()
```

Output

- I am Steve
- I got Rank 1 I am Chris
- I got Rank 2 I am Mark
- I got Rank 3 I am Kate
- I got Rank 4

Python Access Modifiers

- There are three access modifiers are available in Python:
  - Public
    - The public member is accessible from inside or outside the class.
  - Private
    - The private member is accessible only inside class.
    - Define a private member by prefixing the member name with two underscores.
    - Example:__age
  - Protected
    - The protected member is accessible from inside the class and its sub-class.
    - Define a protected member by prefixing the member name with an underscore.
    - Example: _points

Information Hiding

- In the official Python documentation, Data hiding isolates the client from a part of program implementation.
- Some of the essential members must be hidden from the user.
- Programs or modules only reflected how we could use them, but users cannot be familiar with how the application works.
- Thus it provides security and avoiding dependency as well.
- We can perform data hiding in Python using the __double underscore before prefix.
- This makes the class members private and inaccessible to the other classes.

Example:

```
class CounterClass:
    __privateCount = 0
    def count(self):
        self.__privateCount += 1
        print(self.__privateCount)
counter = CounterClass()
counter.count()
counter.count()
print(counter.__privateCount)
```

Advantages of Information Hiding

- The class objects are disconnected from the irrelevant data.
- It enhances the security against hackers that are unable to access important data.
- It isolates object as the basic concept of OOP.
- It helps programmer from incorrect linking to the corrupt data.
- We can isolate the object from the basic concept of OOP.
- It provides the high security which stops damage to violate data by hiding it from the public.

Disadvantages of Information Hiding

- Sometimes programmers need to write the extra lien of the code.
- The data hiding prevents linkage that act as link between visible and invisible data makes the object faster.
- It forces the programmers to write extra code to hide the important data from the common users.

Exception

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

Handling an exception

- If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a try: block.
- After the try: block, include an “except” statement, followed by a block of code which handles the problem as elegantly as possible.
    - try:
        - You do your operations here; except*ExceptionI*:
        - If there is ExceptionI, then execute this block.
    - else:
        - If there is no exception then execute this block.

- **Example 1:**
  - This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all −

```
try:
fh = open("testfile", "w")
fh.write("This is my test file for exception handling!!")
exceptIOError:
print "Error: can\'t find file or read data"
else:
print "Written content in the file successfully" fh.close()
```

  - This produces the following result: Written content

    in the file successfully

THANK YOU