(c) How will you define a structure **Employee** containing the attributes Name, Age, Salary, Address, Hobbies dynamically?

(d) To overload the + operator, which method should be defined in the corresponding class?

(e) To overload the % operator, which method should be defined in the corresponding class?

(f) To overload the //= operator, which method should be defined in the corresponding class?

(g) If a class contains instance methods **__ge__( )** and **__ne__( )**, what do they signify?

(h) What conclusion can be drawn if the following statements work?

```
a = (10, 20) + (30, 40)
b = 'Good' + 'Morning'
c = [10, 20, 30] + [40, 50, 60]
```

(i) What will be the output of the following code snippet?

```
a = (10, 20) - (30, 40)
b = 'Good' - 'Morning'
c = [10, 20, 30] - [40, 50, 60]
```

(j) Will the following statement work? What is your conclusion if it works?

```
print ( 'Hello' * 7)
```

(k) Which out of +, - and * have been overloaded in **str** class?

(l) When would the method **__truediv__( )** defined in the Sample class shown below would get called?

```
class Sample :
    def __truediv__(self, other) :
        pass
```

(m) If != operators has been overloaded in a class then the expression c1 <= c2 would get converted into which function call?

(n) How will you define the overloaded * operator for the following code snippet?

```
c1 = Complex(1.1, 0.2)
c2 = Complex(1.1, 0.2)
c3 = c1 * c2
```

(o) Implement a **String** class containing the following functions:

- Overloaded += operator function to perform string concatenation.
- Method **toLower( )** to convert upper case letters to lower case.
- Method **toUpper( )** to convert lower case letters to upper case.

[C] Match the following pairs:

a. Can't use as identifier name	1. class name
b. basic_salary	2. class variable
c. CellPhone	3. keyword
d. count	4. local variable in a function
e. self	5. private variable
f. _fuel_used	6. strongly private identifier
g. __draw( )	7. method that Python calls
h. __iter__( )	8. meaningful only in instance func.

20 Containership and Inheritance

“Reuse, and you will benefit...”

Contents

KanNotes

Reuse Mechanisms

- Instead of reinventing the same code that is already available, it makes sense in reusing existing code.
- Python permits two code reuse mechanisms:
  (a) Containership (also called composition)
  (b) Inheritance
- In both mechanisms we can reuse existing classes and create new enhanced classes based on them.
- We can reuse existing classes even if their source code is not available.

Which to use When?

- Containership should be used when the two classes have a 'has a' relationship. For example, a College has Professors. So **College** class's object can contain one or more **Professor** class's object(s).
- Inheritance should be used when the two classes have a 'like a' relationship. For example, a Button is like a Window. So **Button** class can inherit features of an existing class called **Window**.

Containership

- A container can contain one or more contained objects apart from other data, thereby reusing contained objects.
- In the following program a **Department** object is contained in an **Employee** object.

```
class Department :
    def set_department(self) :
        self.__id = input('Enter department id: ')
        self.__name = input('Enter department name: ')

    def display_department(self) :
        print('Department ID is: ', self.__id)
        print('Department Name is: ', self.__name)

class Employee :
    def set_employee(self) :
        self.__eid = input('Enter employee id: ')
        self.__ename = input('Enter employee name: ')
        self.__dobj = Department( )
        self.__dobj.set_department( )

    def display_employee(self) :
        print('Employee ID : ', self.__eid)
        print('Employee Name : ', self.__ename)
        self.__dobj.display_department( )

obj = Employee( )
obj.set_employee( )
obj.display_employee( )
```

Given below is the sample interaction with this program:

```
Enter employee id: 101
Enter employee name: Ramesh
Enter department id: ME
Enter department name: Mechanical Engineering
Employee ID : 101
Employee Name : Ramesh
Department ID is: ME
Department Name is: Mechanical Engineering
```

Inheritance

- In Inheritance a new class called **derived** class can be created to inherit features of an existing class called **base** class.
- Base class is also called **super** class or **parent** class.
- Derived class is also called **sub** class or **child** class.
- In the following program **Index** is the base class and **NewIndex** is the derived class. Note the definition of **NewIndex** class. The mention of **Index** within parentheses indicates that **NewIndex** is being inherited from **Index** class.

```
# base class
class Index :
    def __init__(self) :
        self._count = 0
    def display(self) :
        print('count = ' + str(self._count))

    def incr(self) :
        self._count += 1

# derived class
class NewIndex(Index) :
    def __init__(self) :
        super( ).__init__( )

    def decr(self) :
        self._count -= 1

i = NewIndex( )
i.incr( )
i.incr( )
i.incr( )
i.display( )
i.decr( )
i.display( )
i.decr( )
i.display( )
```

On execution of this program we get the following output:

```
count = 3
count = 2
count = 1
```

- Construction of an object should always proceed from base towards derived.
- So when we create the derived class object, base class **__init__( )** followed by derived class **__init__( )** should get called. The syntax used for calling base class constructor is **super( ).__init__( )**.
- Derived class object contains all base class data. So **_count** is available in derived class.
- When **incr( )** is called using derived class object, first it is searched in derived class. Since it is not found here, the search is continued in the base class.

What is Accessible where?

- Derived class members can access base class members, vice versa is not true.
- There are no keywords in Python to control access of base class members from derived class or from outside the class hierarchy.
- Instead a convention that suggests the desired access is used while creating variable names or method names. This convention is shown below:

  var - access it from anywhere in the program
  _var - access it only from within the class or its subclass
  __var - access it only within the class

- Using **_var** in the class inheritance hierarchy or using **__var** within the class is only a convention. If we violate it we won't get errors, but it would be a bad practice to follow.
- Following program shows the usage of the 3 types of variables.

```
class Base :
    def __init__(self) :
        self.i = 10
        self._a = 3.14
        self.__s = 'Hello'

    def display(self) :
        print (self.i, self._a, self.__s)

class Derived(Base) :
    def __init__(self) :
        super( ).__init__( )
        self.i = 100
        self._a = 31.44
        self.__s = 'Good Morning'
        self.j = 20
        self._b = 6.28
        self.__ss = 'Hi'

    def display(self) :
        super( ).display( )
        print (self.i, self._a, self.__s)
        print (self.j, self._b, self.__ss)
```

```
bobj = Base( )
bobj.display( )
print(bobj.i)
print(bobj._a)
print(bobj.__s)            # causes error

dobj = Derived( )
dobj.display( )
print(dobj.i)
print(dobj._a)
print(dobj.__s)            # causes error
```

If we comment out the statements that would cause error, we will get the following output:

```
10 3.14 Hello
10
3.14
100 31.44 Hello
100 31.44 Good Morning
20 6.28 Hi
100
31.44
```

- Why we get error while accessing **__ss** variable? Well, all __var type of variables get name mangled, i.e. in **Base** class **__s** becomes **_Base__s**. Likewise, in **Derived** class **__s** becomes **_Derived__s** and **__ss** becomes **_Derived__ss**.
- When in **Derived** class's **Display( )** method we attempt to use **__s**, it is not the data member of **Base** class, but a new data member of **Derived** class that is being used.

isinstance( ) and *issubclass( )*

- **isinstance( )** and **issubclass( )** are built-in functions.
- **isinstance(o, c)** is used to check whether an object **o** is an instance of a class **c**.
- **issubclass(d, b)** is used to check whether class **d** has been derived from class **b**.

The *object* class

- All classes in Python are derived from a ready-made base class called **object**. So methods of this class are available in all classes.
- You can get a list of these methods using:

```
print(dir(object))
print(dir(Index))          # Index is derived from Object
print(dir(NewIndex))       # NewIndex is derived from Index
```

Features of Inheritance

- Inheritance facilitates three things:

  (a) Inheritance of existing feature: To implement this just establish inheritance relationship.

  (b) Suppressing an existing feature: To implement this hide base class implementation by defining same method in derived class.

  (c) Extending an existing feature: To implement this call base class method from derived class by using one of the following two forms:

  ```
  super( ).base_class_method( )
  Baseclassname.base_class_method(self)
  ```

Types of Inheritance

- There are 3 types of inheritance:

  (a) Simple Inheritance - Ex. class **NewIndex** derived from class **Index**

  (b) Multi-level Inheritance - Ex. class **HOD** is derived from class **Professor** which is derived from class **Person**.

  (c) Multiple Inheritance - Ex. class **HardwareSales** derived from two base classes—**Product** and **Sales**.

- In multiple inheritance a class is derived from 2 or more than 2 base classes. This is shown in the following program:

```
class Product :
    def __init__(self) :
        self.__title = input ('Enter title: ')
```

```
        self.__price = input ('Enter price: ')

    def display_data(self) :
        print(self.__title, self.__price)

class Sales :
    def __init__(self) :
        self.__sales_figures = [int(x) for x in
            input('Enter sales fig: ').split( )]

    def display_data(self) :
        print(self.__sales_figures)

class HardwareItem(Product, Sales) :
    def __init__(self) :
        Product.__init__(self)
        Sales.__init__(self)
        self.__category = input ('Enter category: ')
        self.__oem = input ('Enter oem: ')

    def display_data(self) :
        Product.display_data(self)
        Sales.display_data(self)
        print(self.__category, self.__oem)

hw1 = HardwareItem( )
hw1.display_data( )
hw2 = HardwareItem( )
hw2.display_data( )
```

Given below is the sample interaction with this program:

```
Enter title: Bolt
Enter price: 12
Enter sales fig: 120 300 433
Enter category: C
Enter oem: Axis Mfg
Bolt 12
[120, 300, 433]
C Axis Mfg
Enter title: Nut
Enter price: 8
Enter sales fig: 1000 2000 1800
Enter category: C
Enter oem: Simplex Pvt Ltd
Nut 8
[1000, 2000, 1800]
C Simplex Pvt Ltd
```

- Note the syntax for calling **__init__( )** of base classes in the constructor of derived class:

```
Product.__init__(self)
Sales.__init__(self)
```

  Here we cannot use here the syntax **super.__init__( )**.

- Also note how the input for sales figures has been received using list comprehension.

Diamond Problem

- Suppose two classes **Derived1** and **Derived2** are derived from a base class called **Base** using simple inheritance. Also, a new class **Der** is derived from **Derived1** and **Derived2** using multiple inheritance. This is known as diamond relationship.

- If we now construct an object of **Der** it will have one copy of members from the path **Base -> Derived1** and another copy from the path **Base --> Derived2**. This will result in ambiguity.

- To eliminate the ambiguity, Python linearizes the search order in such a way that the left to right order while creating **Der** is honored. In our case it is **Derived1, Derived2**. So we would get a copy of members from the path **Base --> Derived1**. Following program shows this implementation:

```
class Base :
    def display(self) :
        print('In Base')

class Derived1(Base) :
    def display(self) :
        print('In Derived1')

class Derived2(Base) :
    def display(self) :
        print('In Derived2')

class Der(Derived1, Derived2) :
    def display(self) :
        super( ).display( )
        Derived1.display(self)
        Derived2.display(self)
        print(Der.__mro__)

d1 = Der( )
d1.display( )
```

On executing the program we get the following output:

```
In Derived2
In Derived1
In Derived2
(<class '__main__.Der'>, <class '__main__.Derived1'>, <class
'__main__.Derived2'>, <class '__main__.Base'>, <class 'object'>)
```

- **__mro__** gives the method resolution order.

Abstract Classes

- Suppose we have a **Shape** class and from it we have derived **Circle** and **Rectangle** classes. Each contains a method called **draw( )**. However, drawing a shape doesn't make too much sense, hence we do not want **draw( )** of **Shape** to ever get called. This can happen only if we can prevent creation of object of **Shape** class. This can be done as shown in the following program:

```
from abc import ABC, abstractmethod
class Shape(ABC) :
    @abstractmethod
    def draw(self) :
        pass

class Rectangle(Shape) :
    def draw(self) :
        print('In Rectangle.draw')

class Circle(Shape) :
    def draw(self) :
        print('In Circle.draw')

s = Shape( )   # will result in error, as Shape is abstract class
c = Circle( )
c.draw( )
```

- A class from which an object cannot be created is called an abstract class.
- **abc** is a module. It stands for abstract base classes. From **abc** we have imported class **ABC** and decorator **abstractmethod**.
- To create an abstract class we need to derive it from class **ABC**. We also need to mark **draw( )** as abstract method using the decorator **@abstractmethod**.
- If an abstract class contains only methods marked by the decorator **@abstractmethod**, it is often called an interface.
- Decorators are discussed in Chapter 24.

Runtime Polymorphism

- Polymorphism means one thing existing in several different forms. Runtime polymorphism involves deciding at runtime which function from base class or derived class should get called. This feature is widely used in C++.
- Parallel to Runtime Polymorphism, Java has a Dynamic Dispatch mechanism which works similarly.
- Python is dynamically typed language, where type of any variable is determined at runtime based on its usage. Hence discussion of Runtime Polymorphism or Dynamic Dispatch mechanism is not relevant in Python.

P</> *Programs*

Problem 20.1

Define a class **Shape**. Inherit two classes **Circle** and **Rectangle**. Check programmatically the inheritance relationship between the classes.

Create **Shape** and **Circle** objects. Report of which classes are these objects instances of.

Program

```
class Shape :
    pass
class Rectangle(Shape) :
    pass
class Circle(Shape) :
    pass

s = Shape( )
c = Circle( )
print(isinstance(s, Shape))
print(isinstance(s, Rectangle))
print(isinstance(s, Circle))
print(issubclass(Rectangle, Shape))
print(issubclass(Circle, Shape))
```

Output

```
True
False
False
True
True
```

Problem 20.2

Write a program that uses simple inheritance between classes **Base** and **Derived**. If there is a method in **Base** class, how do you prevent it from being overridden in the **Derived** class?

Program

```
class Base :
    def __method(self):
        print('In Base.__method')

    def func(self):
        self.__method( )

class Derived(Base):
    def __method(self):
        print('In Derived.__method')

b = Base( )
b.func( )
d = Derived( )
d.func( )
```

Output

```
In Base.__method
In Base.__method
```

Tips

- To prevent method from being overridden, prepend it with __.
- When **func( )** is called using **b**, **self** contains address of **Base** class object. When it is called using **d**, **self** contains address of **Derived** class object.
- In **Base** class **__method( )** gets mangled to **_Base__method( )** and in **Derived** class it becomes **_Derived__method( )**.
- When **func( ) calls __method( )** from **Base** class, it is the **_Base__method( )** that gets called. In effect, **__method( )** cannot be overridden. This is true, even when **self** contains address of the **Derived** class object.

Problem 20.3

Write a program that defines an abstract class called **Printer** containing an abstract method **print( )**. Derive from it two classes—**LaserPrinter** and **Inkjetprinter**. Create objects of derived classes and call the **print( )** method using these objects, passing to it the name of the file to be printed. In the **print( )** method simply print the filename and the class name to which **print( )** belongs.

Program

```
from abc import ABC, abstractmethod
class Printer(ABC) :
    def __init__(self, n) :
        self.__name = n

    @abstractmethod
    def print(self, docName) :
        pass

class LaserPrinter(Printer) :
    def __init__(self, n) :
        super( ).__init__(n)

    def print(self, docName) :
        print('>> LaserPrinter.print')
        print('Trying to print :', docName)

class InkjetPrinter(Printer) :
    def __init__(self, n) :
        super( ).__init__(n)

    def print(self, docName) :
        print('>> InkjetPrinter.print')
        print('Trying to print :', docName)

p = LaserPrinter('LaserJet 1100')
p.print('hello1.pdf')
p = InkjetPrinter('IBM 2140')
p.print('hello2.doc')
```

Output

```
>> LaserPrinter.print
Trying to print :
hello1.pdf
>> InkjetPrinter.print
Trying to print :
hello2.doc
```

Problem 20.4

Define an abstract class called **Character** containing an abstract method **patriotism( )**. Define a class **Actor** containing a method **style( )**. Define a class **Person** derived from **Character** and **Actor**. Implement the method **patriotism( )** in it, and override the method **style( )** in it. Also define a new method **do_acting( )** in it. Create an object of **Person** class and call the three methods in it.

Program

```
from abc import ABC, abstractmethod
class Character(ABC) :
    @abstractmethod
    def patriotism(self) :
        pass

class Actor :
    def style(self) :
        print('>> Actor.Style: ')

class Person(Actor, Character) :
    def do_acting(self) :
        print('>> Person.doActing')

    def style(self) :
        print('>> Person.style')

    def patriotism(self) :
        print('>> Person.patriotism')
p = Person( )
p.patriotism( )
p.style( )
p.do_acting( )
```

Output

```
>> Person.patriotism
>> Person.style
>> Person.doActing
```

____

Exercises

[A] State whether the following statements are True or False:

(a) Inheritance is the ability of a class to inherit properties and behavior from a parent class by extending it.

(b) Containership is the ability of a class to contain objects of different classes as member data.

(c) We can derive a class from a base class even if the base class's source code is not available.

(d) Multiple inheritance is different from multiple levels of inheritance.

(e) An object of a derived class cannot access members of base class if the member names begin with __.

(f) Creating a derived class from a base class requires fundamental changes to the base class.

(g) If a base class contains a member function **func( )**, and a derived class does not contain a function with this name, an object of the derived class cannot access **func( )**.

(h) If no constructors are specified for a derived class, objects of the derived class will use the constructors in the base class.

(i) If a base class and a derived class each include a member function with the same name, the member function of the derived class will be called by an object of the derived class.

(j) A class **D** can be derived from a class **C**, which is derived from a class **B**, which is derived from a class **A**.

(k) It is illegal to make objects of one class members of another class.

[B] Answer the following questions:

(a) Which module should be imported to create abstract class?

(b) For a class to be abstract from which class should we inherit it?

(c) Suppose there is a base class **B** and a derived class **D** derived from **B**. **B** has two **public** member functions **b1( )** and **b2( )**, whereas **D** has two member functions **d1( )** and **d2( )**. Write these classes for the following different situations:

- **b1( )** should be accessible from main module, **b2( )** should not be.
- Neither **b1( )**, nor **b2( )** should be accessible from main module.
- Both **b1( )** and **b2( )** should be accessible from main module.

(d) If a class **D** is derived from two base classes **B1** and **B2**, then write these classes each containing a constructor. Ensure that while building an object of type **D**, constructor of **B2** should get called. Also provide a destructor in each class. In what order would these destructors get called?

(e) Create an abstract class called **Vehicle** containing methods **speed( )**, **maintenance( )** and **value( )** in it. Derive classes **FourWheeler**, **TwoWheeler** and **Airborne** from **Vehicle** class. Check whether you are able to prevent creation of objects of **Vehicle** class. Call the methods using objects of other classes.

(f) Assume a class **D** that is derived from class **B**. Which of the following can an object of class **D** access?

- members of **D**
- members of **B**

[C] Match the following pairs:

a. __mro__( )	1. ‘has a’ relationship
b. Inheritance	2. Object creation not allowed
c. __var	3. Super class
d. Abstract class	4. Root class
e. Parent class	5. ‘is a’ relationship
f. object	6. Name mangling
g. Child class	7. Decides resolution order
h. Containership	8. Sub class

[D] Attempt the following questions:

(a) From which class is any abstract class derived?

(b) At a time a class can be derived from how many abstract classes?

(c) How do we create an abstract class in Python?

(d) What can an abstract class contain—instance method, class method, abstract method?

(e) How many objects can be created from an abstract class?

(f) What will happen on execution of this code snippet?

```
from abc import ABC, abstractmethod
class Sample(ABC) :
@abstractmethod
def display(self) :
    pass
s = Sample( )
```

(g) Suppose there is a class called **Vehicle**. What should be done to ensure that an object should not be created from **Vehicle** class?

(h) How will you mark an instance method in an abstract class as abstract?

(i) There is something wrong in the following code snippet. How will you rectify it?

```
class Shape(ABC) :
@abstractmethod
def draw(self) :
    pass

class Circle(Shape) :
@abstractmethod
def draw(self) :
    print('In draw')
```