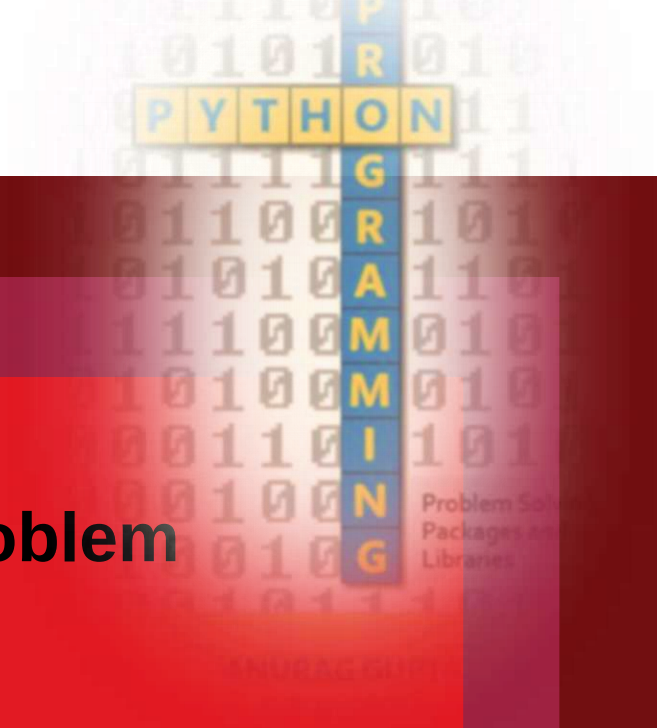
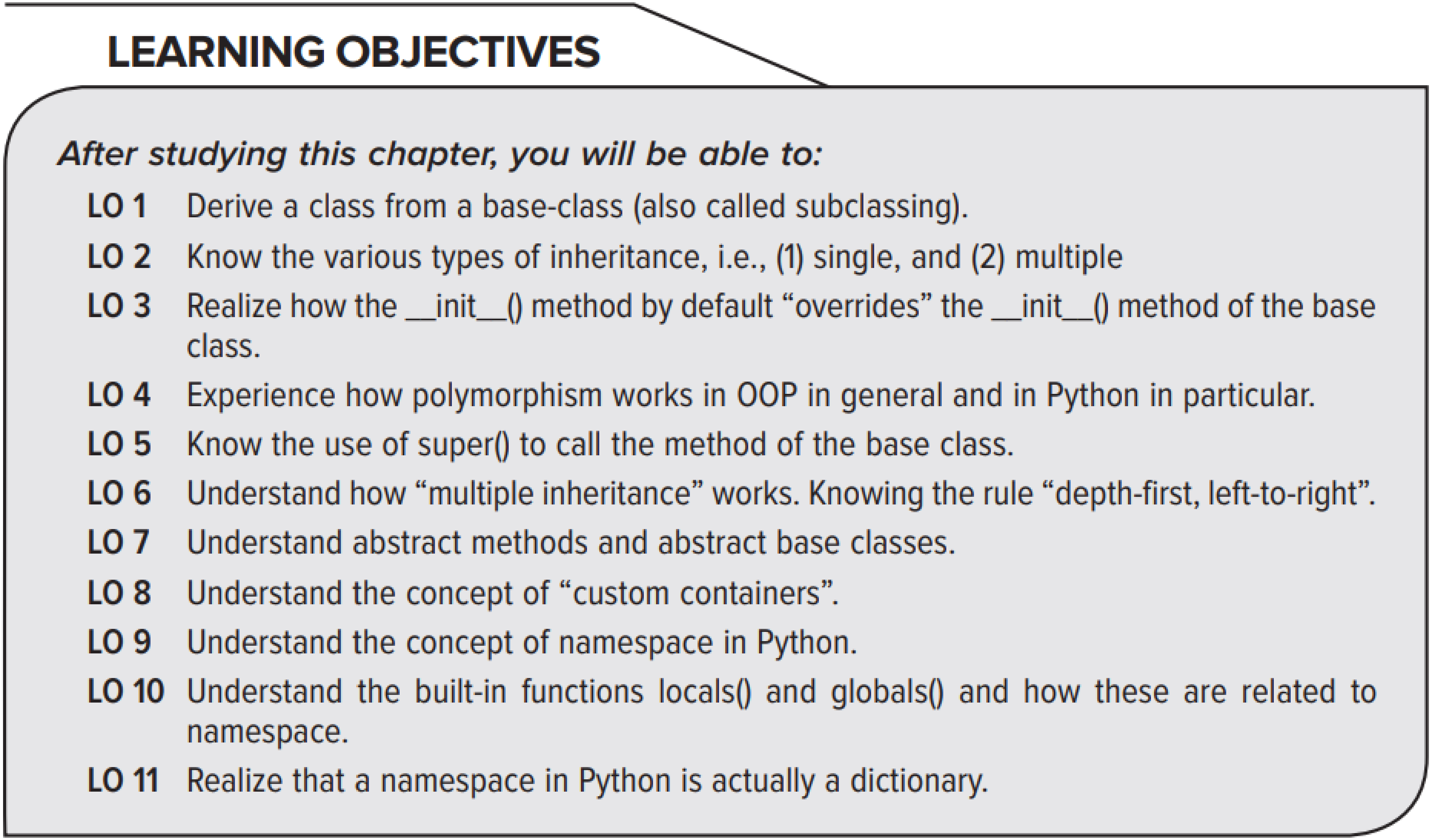
Because learning changes everything.®



TITLE: Python Programming: Problem Solving, Packages and Libraries

Edition

Lecture PPT



LEARNING OBJECTIVES

After studying this chapter, you will be able to:

LO 1 Derive a class from a base-class (also called subclassing).

LO 2 Know the various types of inheritance, i.e., (1) single, and (2) multiple

LO 3 Realize how the __init__() method by default “overrides” the __init__() method of the base class.

LO 4 Experience how polymorphism works in OOP in general and in Python in particular.

LO 5 Know the use of super() to call the method of the base class.

LO 6 Understand how “multiple inheritance” works. Knowing the rule “depth-first, left-to-right”.

LO 7 Understand abstract methods and abstract base classes.

LO 8 Understand the concept of “custom containers”.

LO 9 Understand the concept of namespace in Python.

LO 10 Understand the built-in functions locals() and globals() and how these are related to namespace.

LO 11 Realize that a namespace in Python is actually a dictionary.

Introduction to sub-classing (Inheritance)

The concept of sub-classing is best understood with an example. The example code below shows a “base or parent” class named BaseClass and a “derived” class named DerivedClass.

```
class BaseClass:
    pass
# Derived class
class DerivedClass(BaseClass):# DerivedClass is derived from BaseClass
    pass
# Create objects b (Of type BaseClass) and d (of type DerivedClass)
b = BaseClass()
d = DerivedClass()
print('Type of b ', type(b))
print('Type of d ', type(d))
print('Parent of d',DerivedClass.__bases__)#.__bases__ -> parent of derived class
print('Parent of b ',BaseClass.__bases__)# All classes inherit from class object
```

```
# Output
Type of b  <class '__main__.BaseClass'>
Type of d  <class '__main__.DerivedClass'>
Parent of d (<class '__main__.BaseClass'>,)
Parent of b  (<class 'object'>,)
```

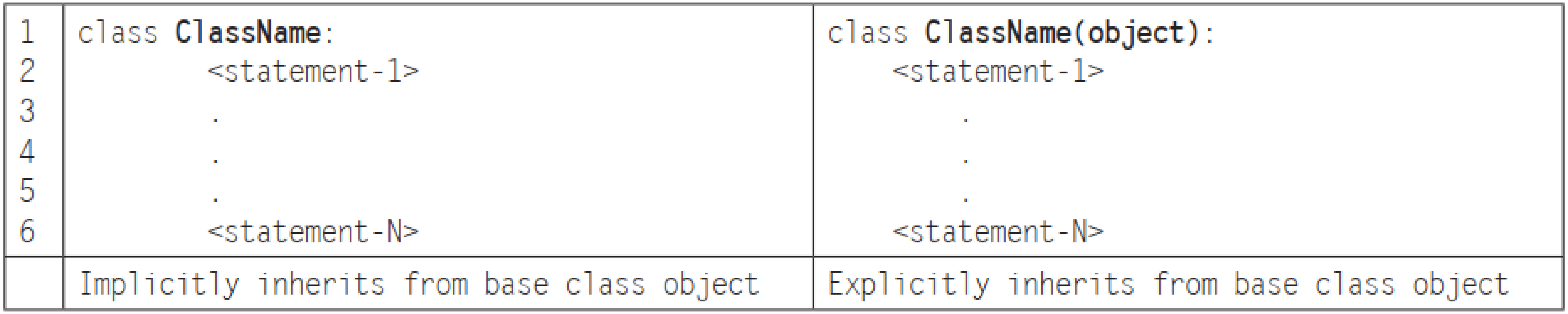
Explicit and implicit inheritance from the “base class object”

In Python the class at top of class hierarchy is called “object” and this strange terminology is cause for some confusion.

When giving a class definition, you may or may not specify this super-most class called object.

If you don’t mention the keyword object (As is in the LHS of figure below). Then you “implicitly” inherit from “super most class ie object”

However if you mention the word “object” within brackets in the class definition, as is shown on the RHS of figure below, then you “explicitly inherit from the super-most class object”



```
class ClassName:
        <statement-1>
        .
        .
        .
        <statement-N>
```

Implicitly inherits from base class object

```
class ClassName(object):
    <statement-1>
        .
        .
        .
    <statement-N>
```

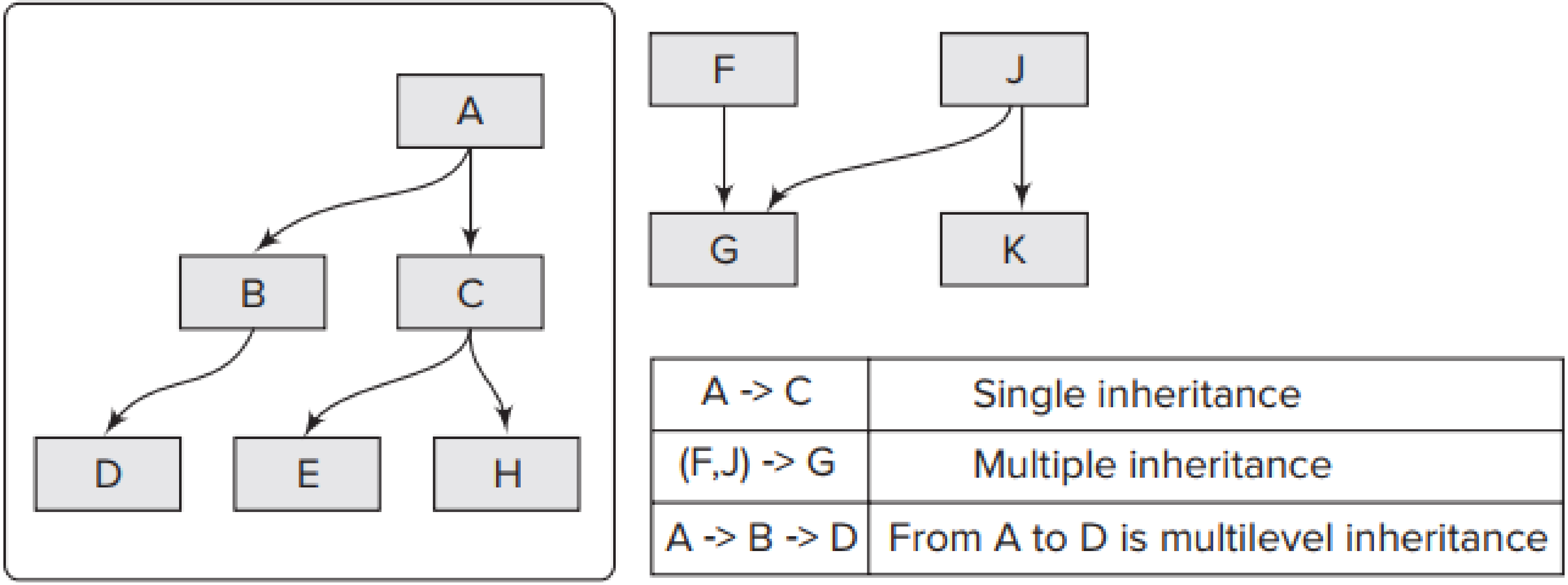
Explicitly inherits from base class object

Types of Inheritance

1. Single inheritance: If a subclass inherits from only one superclass as its immediate superclass then it is single inheritance. A class acquires the properties of another class.
2. Multiple inheritance: If a subclass inherits from more than 1 superclass as its immediate superclass, then it is multiple inheritance.
3. Multilevel inheritance: If a subclass inherits from another class which in turn has inherited from another superclass, then it is called multiple level inheritance. In Python there can be any "number of levels".
4. Hierarchical inheritance: A class may have multiple subclasses. Then the relationship of the superclass to its subclasses is called hierarchical inheritance. Basically the superclass is serving as the base class for a number of its subclasses.
5. Hybrid inheritance: a mix of two or more of the above types of inheritance.

Types of Inheritance -- Figure

The following figure shows the various types of inheritence mentioned in the previous slide.



A -> C	Single inheritance
(F,J) -> G	Multiple inheritance
A -> B -> D	From A to D is multilevel inheritance

FIGURE 14.1 "Single", "Multiple" and "Multi-level" inheritance

Single Inheritance

In the topic of single inheritance the following topics are discussed in the book:

- **A simple inheritance of all functionalities of the base class or parent class.**
- **Inheritance, where the derived class has an __init__() method of its own**
- **Both the derived class and the parent class have their own __init__() methods and there is a need to call the __init__() of base class from the derived class.**
- **Use of super() to call methods other than __init__() of base class also**
- **Calling the __init__() methods of the parent class by using the name of the parent class**
- **Abstract methods**

A simple inheritance of all functionalities of the base class or parent class.

The following script shows the simplest case of inheritence. Here the clas Dog subclasses the class Pet.

Detailed explanation is given in the book

```
class Pet:      # Parent class
    def __init__(self, pName= 'No name'):
        self.pName = pName

class Dog(Pet): # Derived class. Has no __init__() of its own
    pass
p = Pet('my Pet')   #Create an instance of Pet
d = Dog('Tommy')    #Create an instance of Dog
print(p.pName,type(p))  # Output is my Pet <class '__main__.Pet'>
print(d.pName, type(d))  # Output is Tommy <class '__main__.Dog'>
# Output
my Pet <class '__main__.Pet'>
Tommy <class '__main__.Dog'>
```

Inheritance, where the derived class has an __init__() method of its own

The following script gives a slightly more advanced version of the derived class. Here the derived class has an __init__() method of its own also.

See book for detailed explanation

```python
class Pet:
    def __init__(self, pName= 'No name'):
        self.pName = pName
        print('__init__() of Pet class called')

class Dog(Pet):     # Derived class
    def __init__(self, pName= 'Dog'):   # __init__() of derived class
        self.pName = pName
        print('__init__() of Dog Class called')
# Create objects p (Of class Pet) and d (Of class Dog)
p = Pet('my Pet')   #Call __init__() of Pet Class
d = Dog('Tommy')    #Call __init__() of Dog Class
print(p.pName)
print(d.pName)
```

```
#OUTPUT
__init__() of Pet class called
__init__() of Dog Class called
my Pet
Tommy
```

When there is a need to call the __init__() of base class from the __init__() of derived class -- 1

- In Python there may be a situation where you want to use the init() methods of both the derived Class and the Parent class, because you may want some of the initialization to be done in the __init__() method of the derived class and rest of the initialization to be done in the __init__() method of the Base Class.

- If you want to call the __init__() method of the base class then you have to
  - (a) Make this call from inside the __init__() method of the derived class
  - (b) Make a call to the __init__() method of base class using the keyword super().

- Here again there is a slight difference in Python 2.x and Python 3.x. The syntax for the two version is shown in the following code:

```
super(DerivedClass, self).__init__()     # In Python 2.x
super().__init__()                  # In Python 3.x
```

There are two things to note in the call to __init__() of the base class:

- The super() does not have a self-parameter in Python 3.x though it does have a self parameter in Python 2.x.
- If you want to pass any parameter (other than self), you must pass it here.

When there is a need to call the __init__() of base class from the __init__() of derived class -- 2

The example code below shows how one can do the initialization in the __init__() method of the derived class and then also call the __init__() of super class. The usage of super() for both Python 2.x and 3.x is shown, but the line of code for Python 2.x is commented out.

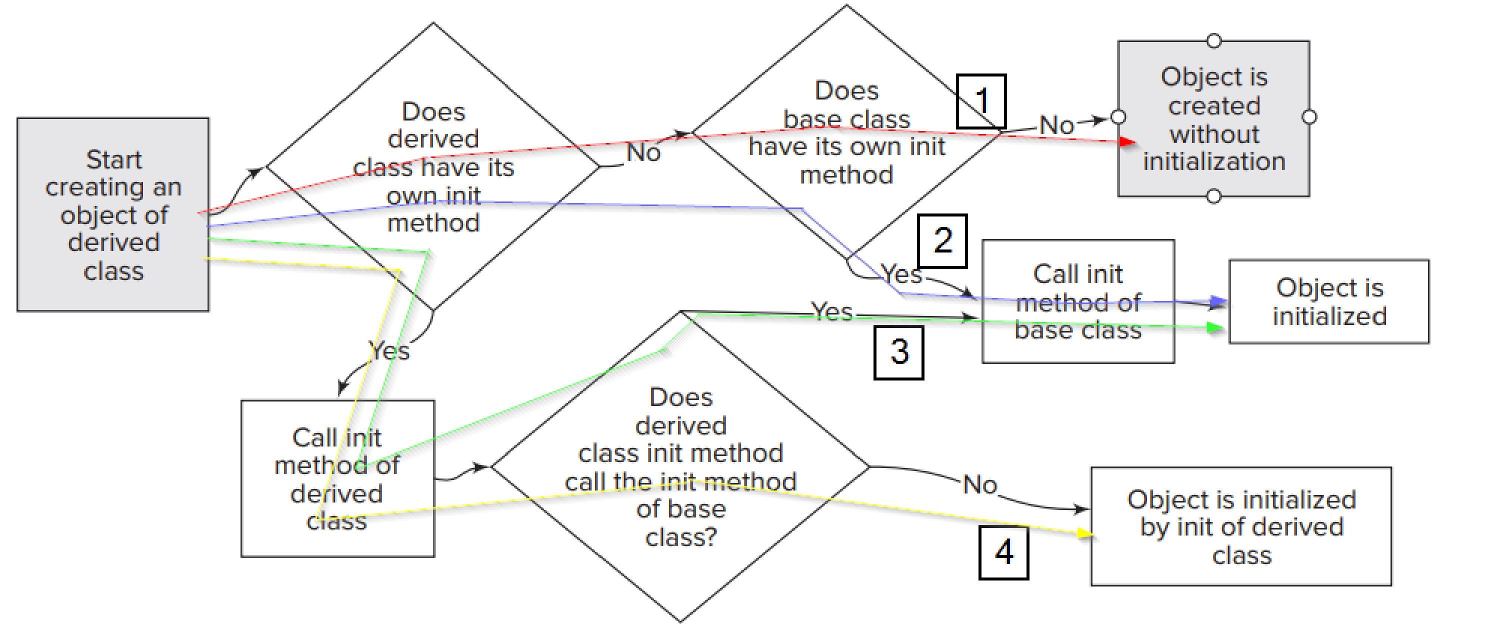
See the book for detailed explanation

```
class Pet:
    def __init__(self, pName= 'No name'):
        self.pName = pName

class Dog(Pet):
    def __init__(self, pName, sound= 'bark'):
        self.sound = sound
        #super(Dog, self).__init__(pName)     In Python 2.x
        super().__init__(pName)              # In Python 3.x
# Create object d
d = Dog('Tommy', 'Woff!')    #Create an instance of Dog
print('Sound is -> ', d.sound)
print('Name of pet-> ', d.pName)
```

```
#OUTPUT
Sound is ->  Woff!
Name of pet->  Tommy
```

Possible routes of intialization of an object of a derived class

- **First the interpreter checks if the derived class i.e. the sub-classed class has an `__init__()` method of its own. If yes then this method is called first.**
- **Within the `__init__()` method of the derived class there may or may not be a call to the init method of the base class.**
- **If there is a call to `__init__()` method of base class, then this `__init__()` method is also called.**
- **The figure shows the 4 possible routes. Routes are marked with colored lines and box with numbers 1 to 4**



Start creating an object of derived class
Does derived class have its own init method
No
Does base class have its own init method
1
No
Object is created without initialization
Yes
2
Call init method of base class
Object is initialized
Yes
Yes
3
Call init method of derived class
Does derived class init method call the init method of base class?
No
4
Object is initialized by init of derived class

FIGURE 14.2 All possible routes that may be taken when an object of a derived class is being initialized by init method. (Note that to save space, __init__() is referred to as init method)

Use of super() to call methods other than `__init__()` of base class also

The following example code that both the parent class ie Pet and the derived class ie Dog have a method `walk()`. So the `walk()` method of Dog “overrides” the `walk()` method of Pet. The code shows how super() is used inside the `walk()`

```
class Pet:
    def __init__(self, pName= 'No name'):
        self.pName = pName
    def walk(self):
        print('Pet is walking')

class Dog(Pet):
    def __init__(self, pName, sound= 'bark'):
        self.sound = sound
        #super(Dog, self).__init__(pName)     In Python 2.x
        super().__init__(pName)             # In Python 3.x
    def walk(self):
        print('Dog is walking')
        #super(Dog, self).walk()              In Python 2.x
        super().walk()                      # In Python 3.x
# Create object d
d = Dog('Tommy', 'Woff!')    #Create an instance of Dog
d.walk()
```

```
# Output
Dog is walking
Pet is walking
```

Calling the __init__() methods of the parent class by using the name of the parent class

It is possible to access the overridden methods of the parent class by using the name of the parent class. This can be done for the `__init__()` method and also for other methods.

Following is the code where the derived class first calls its own `__init__()` and then specifically calls the `__init__()` of the parent class by using the name Pet of the parent class:

```python
class Pet:
    def __init__(self, pName= 'No name'):
        self.pName = pName
        print('constructor of Pet class called')

class Dog(Pet):
    def __init__(self, pName, sound= 'bark'):
        print('Constructor of Dog class called')
        Pet.__init__(self, pName) #Calling __init__() of Pet class

# Create object d of class Dog
d = Dog('Tommy', 'Woff!')    #Create an instance of Dog
print('Name of pet-> ', d.pName)
```

```
# Output
Constructor of Dog class called
constructor of Pet class called
Name of pet->  Tommy
```

Abstract methods -- 1

- In Python, it is possible to create a class with a method but the method is not implemented in the class.
- So to use this method, you must derive a child class from this parent class and then implement the method in the child class.
- The question then is why would you need to do this?
- The answer can be given by an example: suppose you have a Pet class from which you derive various child classes like say Dog and Cat.
- You want both the Dog and the Cat class to have a method say speak, but you don’t want to implement the speak method in the parent Pet class.
- This can be implemented by having an abstract speak() method in base class which is implemented in child classes Dog and Cat.

Abstract methods -- 2

The following script shows how abstract methods work. The parent class Pet has a method speak(), which is not implemented in the parent class.

From this class 2 classes namely Dog and Cat have been derived. The Dog class implements the speak() method while the Cat class does not. So the speak() method of Dog class will work but the speak() method of the cat class will not work and throw an error.

```
class Pet:
    def speak(self):    # Abstract method
        raise NotImplementedError("Please implement this method")

class Dog(Pet):
    def speak(self):    # Abstract method implemented in Dog class
        print('Dog barks')

class Cat(Pet):             # Abstract method not implemented in Cat class
    pass
# Create objects d of Dog class and c of Cat class
d = Dog()
d.speak() # output is Dog barks
c = Cat()
c.speak() #Error
```

Multiple inheritance

Under the topic of multiple inheritance, the following topics are discussed in the book:-

1. **Multiple inheritance**
2. **Potential problems in multiple inheritance.**
3. **Python in-built class attribute __mro__**
4. **Abstract methods and abstract class.**
5. **Creating custom containers**

Multiple Inheritance

The following script shows how multiple inheritance works.

Here we have a class named DerivedClassName and it is derived from 3 base classes namely Base1, Base2 and Base3.

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
        .
        .
        .
    <statement-N>
```

Potential problem in multiple inheritance -- 1

- In multiple inheritance there is one potential problem.
- Suppose a method is defined in more than one parent class, then which of the methods should be implemented?
- In Python, the rule is depth-first, left-to-right.
- What does this mean?
- It means in the above example, the Python interpreter will first check up the leftmost parent class which here is Base1 and go all up to its parent, i.e., to the greatest depth.
- But if it does not find the implementation in Base1 or any of its parents, it will next look up the next parent to the right which is Base2 and go right up to all its parents.
- If it does not find the implementation of the method in its parents, it will go to Base3 and so on.
- So the algorithm first goes to the entire depth of the leftmost class, i.e., Base1 (the entire depth means look up all the parent classes of Base1 as well).
- If the method is not found in the leftmost Parent or Base class, then it searches the entire depth of the base class to its right and so on.

Potential problem in multiple inheritance -- 2

The following example code has 5 classes. The super-most class is GrFather. It has 2 derived classes: (1) Father and (2) Mother.

From these 2 derived classes, 2 further classes are derived, namely: (1) Child1 and (2) Child2.

The Only difference between Child1 and child2 is the “order” of parents.

```
class GrFather: # GrandFather
    def snore(self):
        print('Grandfather snoring')

class Father(GrFather): # Father derived class from GrFather
    def earn(self):
        print('Father earns')

class Mother(GrFather): # Mother derived class from GrFather
    def earn(self):
        print('Mother earns')

class Child1(Father, Mother): # parent class Father comes before Mother
    pass

class Child2(Mother, Father): # parent class Mother comes before Father
    pass
# Create objects c1 and c2 of classes Chid1 and Child2
c1 = Child1()
c2 = Child2()
c1.snore() # Call snore() of GrFather class
c1.earn() # Will Call earn() of Father Class
c2.earn() # Will Call earn() of Mother class
```

```
#OUTPUT
Grandfather snoring
Father earns
Mother earns
```

Potential problem in multiple inheritance -- 3

The figure below shows the hierarchy of the 5 classes. Note the difference in "order" of the parent classes in Child1 and Child2

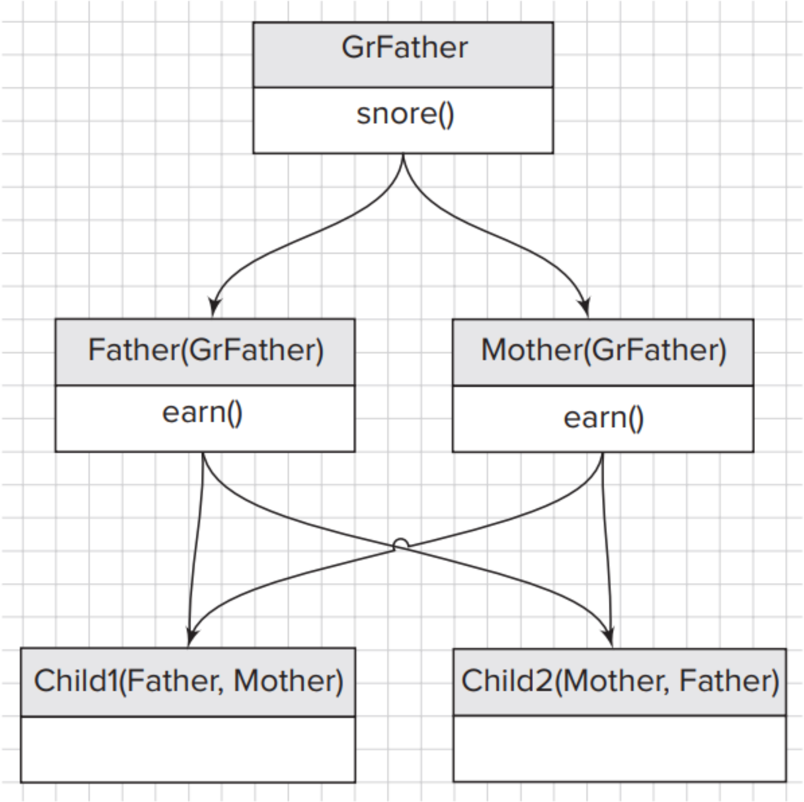


FIGURE 14.3 Two cases of multiple inheritance, i.e., Child1(Father, Mother) and Child2(Mother, Father). (Note the difference in "order" of Father and Mother)

Python in-built class attribute __mro__

- **Python has an in-built class attribute __mro__.**
- **The attribute __mro__ specifies the MRO (Method Resolution Order) which is used when “resolving a method”.**
- **The attribute __mro__ is a “tuple of classes.” This tuple of classes is used by the Python interpreter in “method resolution”**
- **This attribute is a tuple of classes that are considered when looking for base classes during method resolution.**

Abstract methods and abstract class -- 1

- Note that the term “abstract base class” is at times confusing because in Python there is also a module for Abstract Base Classes called abc.
- Here the discussion is not on the abc module but rather on the concept of abstract base classes.
- In fact you may use the term “abstract classes” instead of “abstract base classes” to refer to the general concept of abstract class in Python.
- The word “base” simply indicates that this class “needs to be inherited from”.

Abstract methods and abstract class -- 2

Concept of abstract method and Abstract Base Class:

- **If a class has a method which is declared but not implemented, it is called an abstract method.**
- **If a class contains an abstract method, then it is called an abstract class.**
- **Since an abstract class has abstract methods which have not been implemented, so it cannot be instantiated.**
- **So in order to use an abstract class, it has to be subclassed, i.e., a class has to be derived or inherited from it.**
- **However, note that subclass, i.e., derived class of an abstract class are not required to implement the abstract methods of the base class. So while you cannot instantiate an abstract class, you can instantiate a subclass derived from it (provided it does not have abstract methods of its own).**
- **Abstract class, is especially useful when a programmer wants to define a common API for a set of subclasses. Having common API is especially useful when third-party developers are going to add plugins for your application.**

Creating custom containers -- 1

- In Python strings, list, etc. are all containers, i.e., they “contain” other objects.
- One type of container is a sequence and the other type is a mapping. In a sequence, the index is an integer.
- Thus, strings, lists, tuples are all sequences. On the other hand, a dictionary is a “mapping” from a key to a value.
- So if the container is a mapping, then it has a key and a value.
- In Python it is possible to create your own custom containers.
- For example, you could create a class say Vehicle and then have a vehicle object which in turn could act just like a list like say vehicle[0], vehicle[1], etc.
- To do this, Python provides many methods.
- Two of them are discussed here. They are __getitem__() and __setitem__().
- So if a class implements a __setitem__() method, then the object can have index or keys and those keyed items can be set to values.
- So if a class say Vehicle implements __setitem__(), then you could have objects of Vehicle class with index or keys. The following example will clarify the concept:

Creating custom containers -- 2

The following script shows how to create a class Vehicle, which acts as a custom container. Note that it implements 2 methods __setitem__() and __getitem__()

```
class Vehicle(object):
     def __init__(self, totalV):
         self.totalV = [None]*totalV
     def __setitem__(self, vehicle_number, vehicle_name):
          self.totalV[vehicle_number] = vehicle_name
     def __getitem__(self, vehicle_number):
          return self.totalV[vehicle_number]


# Create object vehicle of class Vehicle
vehicle = Vehicle(3)
# Since Vehicle class implements __setitem__() and __getitem__()
# you can have index for vehicle object
vehicle[0] = 'truck'
vehicle[1] = 'car'
print('vehicle[0]->', vehicle[0], 'vehicle[1]->', vehicle[1])
# Note vehicle[2] also exists but its value is None
print('vehicle[2]->', vehicle[2])
```

```
#OUTPUT
vehicle[0]-> truck vehicle[1]-> car
vehicle[2]-> None
```

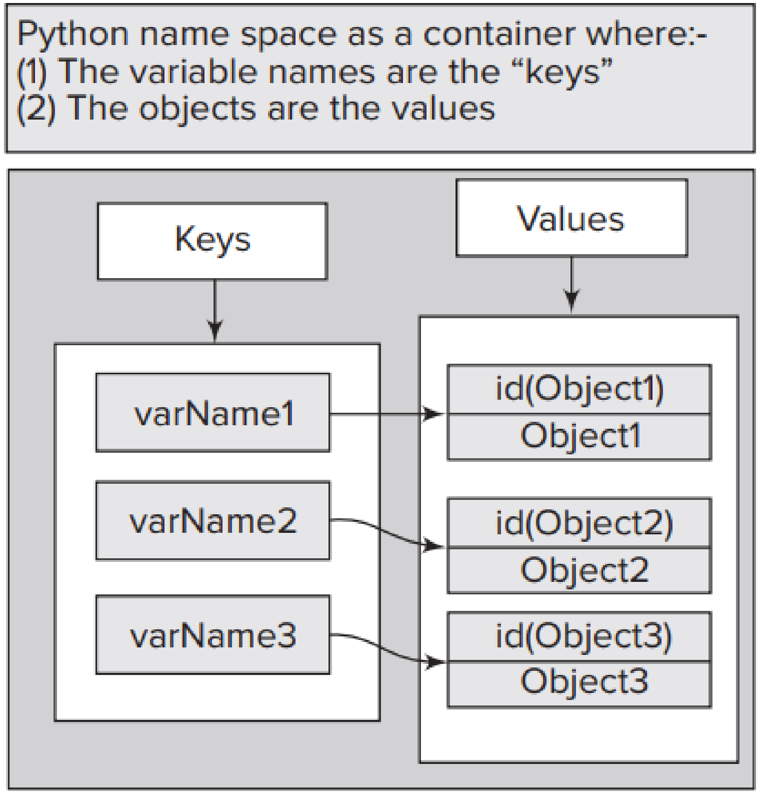
Concept of namespace -- 1

- Namespace is simply a container for mapping names of objects to their corresponding objects.
- Suppose you have three objects referred to by variables varName1, varName2 and varName3.
- Call the objects to which they refer to as object1, object2, object3.
- Please note that you can refer to these objects only by their names, i.e., varName1, varName2 and varName3, but internally these three objects are linked or mapped by their id to their respective variable names.
- Python identifies each object through its id().
- Once an object is created, it has a unique id and this id is linked to its variable name.

Concept of namespace -- 2

So you can actually think of the name space as a dictionary (let’s call it dictNamespace) as follows:

```
dictNamespace = {‘varName1’: object1, ‘varName2’: object2,’varName3’: object3}
```



Python name space as a container where:-
(1) The variable names are the “keys”
(2) The objects are the values

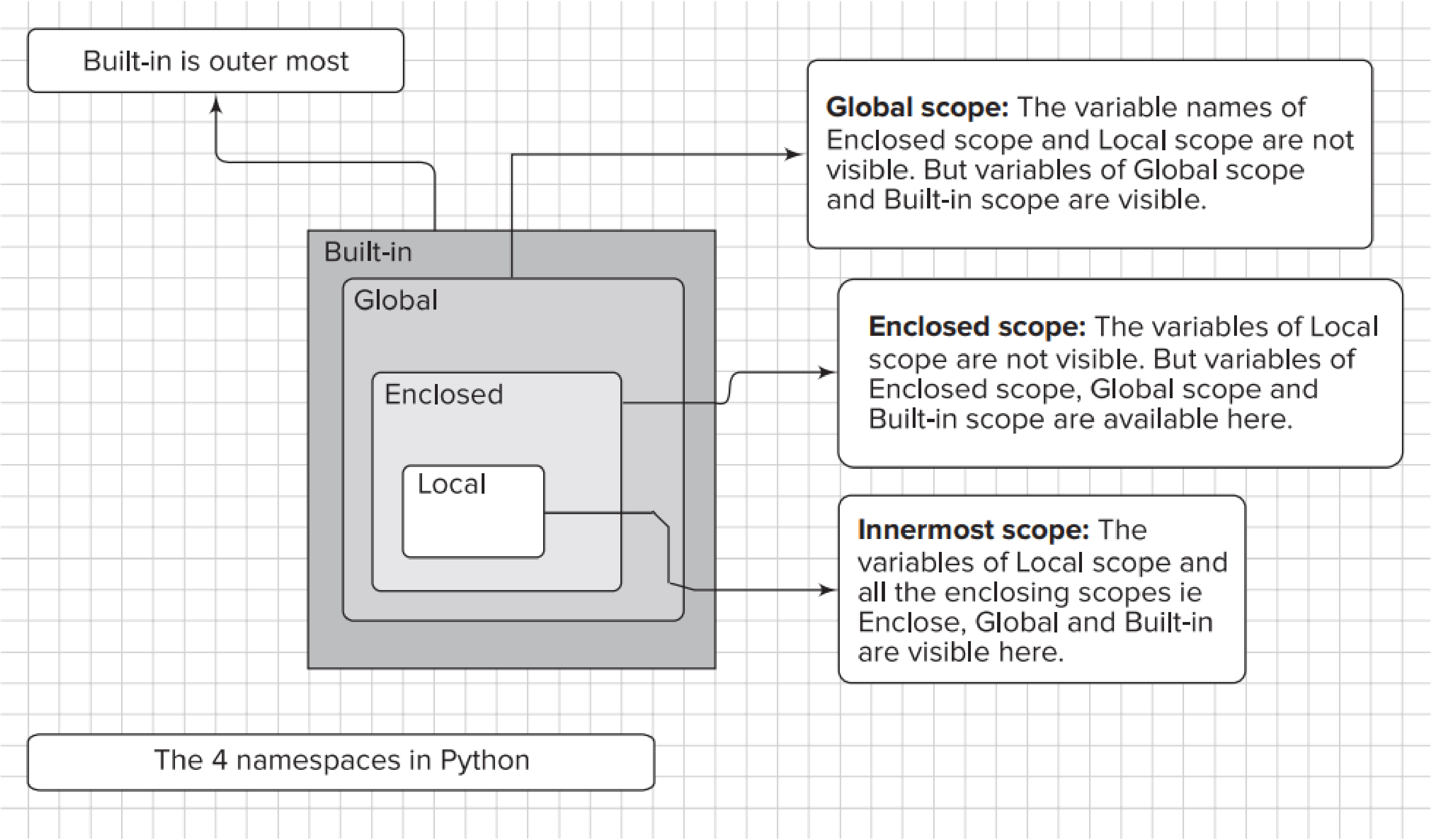
FIGURE 14.4 Python namespace as a “dictionary container”

Concept of namespace -- 3

- There are four different namespaces in Python and they are nested one inside another.
- These four namespaces can be remembered as LEGB (Local, Enclosing, Global and Built-in).
- The innermost is Local and Built-in is the outermost.
- All the variable names in the outer namespace are visible to the inner scope but none of the names in the inner scope are visible to outer scope. (There are ways to work around, i.e., change this but in general this holds).
- These 4 namespaces are shown in the figure on next slide

Concept of namespace -- 4

The figure below shows 4 namespaces (1) Built-in (2) Global (3) Enclosed and (4) Local



Global scope: The variable names of Enclosed scope and Local scope are not visible. But variables of Global scope and Built-in scope are visible.

Enclosed scope: The variables of Local scope are not visible. But variables of Enclosed scope, Global scope and Built-in scope are available here.

Innermost scope: The variables of Local scope and all the enclosing scopes ie Enclose, Global and Built-in are visible here.

FIGURE 14.5 Four name spaces, i.e., LEGB

Namespace dictionary __dict__

- As mentioned earlier, every namespace is a key value pair.
- The key is the name of the attribute in the namespace and its value is the value of that attribute in that namespace.
- This dictionary of attributes and their corresponding values can be accessed using the __dict__ attribute of the class or its instance.
- Please remember that a class has a different namespace than its instance.
- Suppose you have a class say myClass and an instance of this class say myObject.
- Then the namespace of myClass.__dict__ is different from the namespace myObject.__dict__.

The script in next slide shows that the namespace of a class is different from the namespace of an instance of the class

The following script has a class Dog and an “instance” of this class called blackDog. The script shows the use of __dict__ attribute and how this __dict__[“key”] = value can be used to add new attributes to an object.

```python
# class_namespace_instance_namespace.py
class Dog:
    dog_sound = 'bark'# Class variable common to all instances of Dog
    def __init__(self, color):
        self.color = color

# Create instance of Dog
blackDog = Dog('black')

# class and instance namespace
print('Class namespace-> ',Dog.__dict__)    # Gives Class namespace
print('Instance or object namespace-> ', blackDog.__dict__)# object namespace

# You can add attributes and their values to instance namespace
# A new attribute dog_act with value 'Wag tail' added
blackDog.__dict__['dog_act'] = 'Wag tail'
print("New attribute 'dog_act' with value ->",blackDog.__dict__['dog_act'])
# The class attribute dog_sound modified
blackDog.__dict__['dog_sound'] = 'Loud Bark'
print('Class attribute dog_sound changed->',blackDog.__dict__['dog_sound'])
```

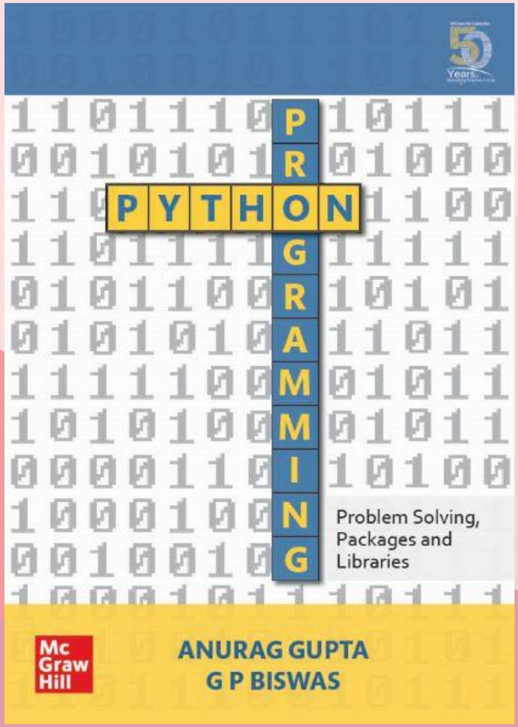
```
# Output …
Class namespace->  {'__module__': '__main__', 'dog_sound': 'bark', '__doc__': None, '__init__':
<function Dog.__init__ at 0x00E50A98>, '__dict__': <attribute '__dict__' of 'Dog' objects>,
'__weakref__': <attribute '__weakref__' of 'Dog' objects>}
Instance or object namespace->  {'color': 'black'}
New attribute 'dog_act' with value -> Wag tail
Class attribute dog_sound changed-> Loud Bark

```

Beyond Text

The “Beyond Text” section of this chapter has following topics:

1. Studying C3 Linearization Algorithm.
2. __new__() versus __init__() methods
3. Understanding meta-classes in Python
4. “Real subclasses” versus “Virtual subclasses”
5. Abstract Base Class Module, i.e., abc

Students may like to study these topics.



PYTHON PROGRAMMING

Problem Solving, Packages and Libraries

ANURAG GUPTA
G P BISWAS

Because learning changes everything.®

Thank You!

For any queries or feedback contact us at:

@ support.india@mheducation.com

☎ 1800-103-5875

www.mheducation.co.in