UCL

INST0004 Programming 2

Lecture 08: Inheritance and Polymorphism

Module Leader:
Dr Daniel Onah

2023-24


UCL

Copyright Note

Important information to adhere to

Summary of Previous Lecture

Recap of previous lecture

Re: Cap☺

In the last lecture we looked at ...

- looked at more of classes and objects
- we understand object decomposition, state and behaviour
- we looked at public interface and encapsulation
- understand how to create a simple class structure
- we discussed about class variables and constructors
- we discussed how to import a helper class into another class
- we discussed about the concept of static and class methods
- we implement and test a few class programs

Learning Outcomes

The learning outcomes for the lecture

The objective of this week's lecture is to introduce the concept of object-oriented programming in Python. At the end of the lecture, you should be able to:

- the basic concepts of object-oriented programming
- understand classes creation
- understand objects creation
- concepts and types of constructors
- concepts of inheritance
- concepts of polymorphism
- various Python methods to achieve object-oriented programming

Introduction to Inheritance

What is inheritance in OOP?

The object-oriented framework includes inheritance as a key component. Inheritance allows us to reuse program codes from another class. Since we can use the program codes from existing classes to build another class without starting from the beginning, inheritance provides this opportunity. In the concept of inheritance, a child class inherits the parent class properties, the associated data members, and the functions defined in the parent. The parent class is known as the base class and the child class is know as the derived class. Here the **derived class** can inherit all features of the **base class** while having it's own unique features. The logic of inheritance is simple, you just need to declare the parent or base class after the child or derived class.

Introduction to Inheritance

What is inheritance in OOP?

In Python, a derived class can inherit a base class by simply declaring the **base class** after the name of the **derived class** in parenthesis or brackets. The syntax below demonstrates how to inherit a base class into a derived class.

Syntax

```
#!/usr/bin/env python
"""
A Syntax for inheriting the base class into a derived class
"""
class <derived-class_Name> (base-class_Name):
# body of the derived class
```

Multiple base classes inheritance

How do we inherit multiple classes?

In Python, a derived class can inherit the properties of multiple base classes. We can achieve this by declaring all the base classes inside the brackets. Let's look at the syntax below that would help us to understand how this is done.

Syntax

```
#!/usr/bin/env python
"""
A Syntax for inheriting multiple base classes into a derived class
"""
class <derived-class_Name> (<base-class_Name-1>, <base-class_Name-2>,
    ... <base-class_Name-n>):
# The derived class body
```

Inheritance: Example

Creating base and derived classes

Let's look at an example of how to create a **base** class and a **derived** class. We will also see how the derived class inherits the properties of the base class.

```
#!/usr/bin/env python
"""
A program to demonstrate class inheritance
"""
# Inheritance example
# creating a base (parent) class
class Base():
    # defining a function inside the base class
    def show_base(self):
        print("This is Base class function")
```

Inheritance: Example

Creating base and derived classes

```
# creating a derived class by passing the base class in parenthesis
class Derived(Base):
    # defining the function inside the derived class
    def show_derived(self):
        print("This is Derived class function")
# creating object of the derived class
obj = Derived()
# accessing functions of both the base class and the derived class
obj.show_base()
obj.show_derived()
```

Inheritance: Solution

Creating base and derived classes

Output

```
This is Base class function
This is Derived class function
```

Types of Inheritance

What are the different types of inheritance?

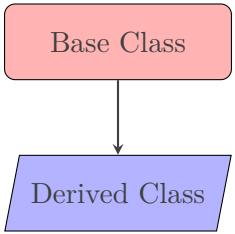
There are four basic types of inheritance in Python. This depends on the number of child or derived and parent or base groups involved. This types are as follows:

- Single inheritance
- Multi-level inheritance
- Multiple inheritance
- Hierarchical inheritance

Single Inheritance

What are the different types of inheritance?

A derived class can inherit properties from a **single** base class, allowing for code reuse and new features to existing code. Let's illustrate this with the figure below.



Single Inheritance: Example

Creating base and derived classes

Let's look at an example of how to create a **single inheritance** program. We will also see how the derived class inherits the properties of the base class.

```
#!/usr/bin/env python
"""
A program to demonstrate single inheritance
"""
# single inheritance example
# Creating a base class
class Base():
    # defining function inside the base class
    def base_func(self, base):
        print("Base class function")
        self.base = base
        print("Base class: ", self.base)
```

Single Inheritance: Example

Creating base and derived classes

```
13 # creating a derived class
14 class Derived(Base):
15     # defining function inside the derived class
16     def derived_func(self, derived):
17         print("Derived class function")
18         self.derived = derived
19         print("Derived class: ", self.derived)
20 # creating object of derived class
21 obj = Derived()
22 # accessing functions of both base abd derived classes
23 obj.base_func(15)
24 obj.derived_func(25)
```

Single Inheritance: Solution

Creating base and derived classes

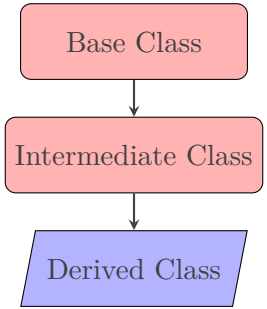
Output

```
Base class function
Base class:  15
Derived class function
Derived class:  25
```

Multi-level Inheritance

What is multi-level inheritance?

A multi-level inheritance, features or properties from the base and derived classes are inherited or passed down to the new derived class. There is no limit to the number of levels in a multi-level inheritance. Let's look at the diagram below to help us understand this concept properly.



Multi-level Inheritance: Example

Creating base, intermediate and derived classes

Let's look at an example on how to create a **multi-level inheritance** program. We will also see how the derived class inherits the properties of both the intermediate class and the base class.

```
#!/usr/bin/env python
"""
A program to demonstrate Multi-level inheritance
"""
# creating a base class
class Base():
    def __init__(self):
        print("Base constructor")
    # defining function insde base class
    def funBase(self, base):
        print("Base class function")
        self.base = base
        print("Base class: ", self.base)
```

Multi-level Inheritance: Example

Creating base, intermediate and derived classes

```
14 # creating intermediate class
15 class Intermediate(Base): # this inherits features from the base class
16     # defining function inside intermediate class
17     def funIntm(self, intm):
18         print("Intermediate class function")
19         self.intm = intm
20         print("Intermediate class: ", self.intm)
21 # creating derived class
22 class Derived(Intermediate): # this inherits features from intermediate
23     # defining function inside derived class
24     def funDerived(self, derived):
25         print("Derived class function")
26         self.derived = derived
27         print("Derived class: ", self.derived)
```

Multi-level Inheritance: Example

Creating base, intermediate and derived classes

```
28 # creating object of derived class
29 obj = Derived()
30 # accessing object of derived class
31 obj.funBase(20)
32 obj.funIntm(40)
33 obj.funDerived(60)
```

Multi-level Inheritance: Solution

Creating base, intermediate and derived classes

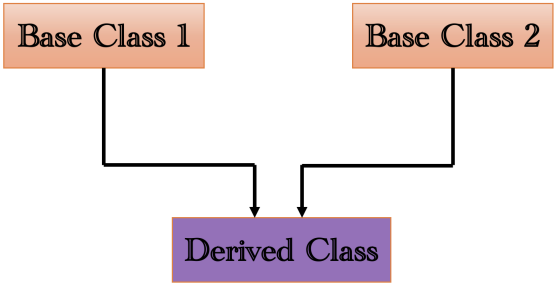
Output

```
Base constructor
Base class function
Base class:  20
Intermediate class function
Intermediate class:  40
Derived class function
Derived class:  60
```

Multiple Inheritance

What is multiple inheritance?

A multiple inheritance is a type of inheritance in which a class may be derived from multiple base classes. In this approach, the derived class **inherits all of the features and properties of the multiple base classes**. Let's see an example to help use understand how a derived class inherits properties of two base classes.



Multiple Inheritance: Example

Creating two base classes and a derived class

Let's look at an example on how to create **multiple inheritance** program. We will also see how the derived class inherits the properties of two base classes.

```
#!/usr/bin/env python
"""
A program to demonstrate multiple inheritance
"""
#  creating first base class
class Base1():
    # defining function inside base class 1
    def funcBase1(self,base1):
        print("Base Class 1 Function")
        self.base1 = base1
        print("Base Class 1 Value: ", self.base1)
```

Multiple Inheritance: Example

Creating two base classes and a derived class

```
12 # creating second base class
13 class Base2():
14     # defining function inside base class 2
15     def funcBase2(self, base2):
16         print("Base Class 2 Function")
17         self.base2 = base2
18         print("Base Class 2 Value: ", self.base2)
19 # creating a derived class
20 class Derived(Base1, Base2):
21     # defining function inside derived class
22     def funcDerived(self, derived):
23         print("Derived Class Function")
24         self.derived = derived
25         print("Derived Class Value: ", self.derived)
```

Multiple Inheritance: Example

Creating two base classes and a derived class

```
26 # creating object of derived class
27 obj = Derived()
28 # accessing functions of both base and derived classes
29 obj.funcBase1(15)
30 obj.funcBase2(25)
31 obj.funcDerived(35)
```

Multiple Inheritance: Solution

Creating two base classes and a derived class

Output

```
Base Class 1 Function
Base Class 1 Value:  15
Base Class 2 Function
Base Class 2 Value:  25
Derived Class Function
Derived Class Value:  35
```

Hierarchical Inheritance: Example

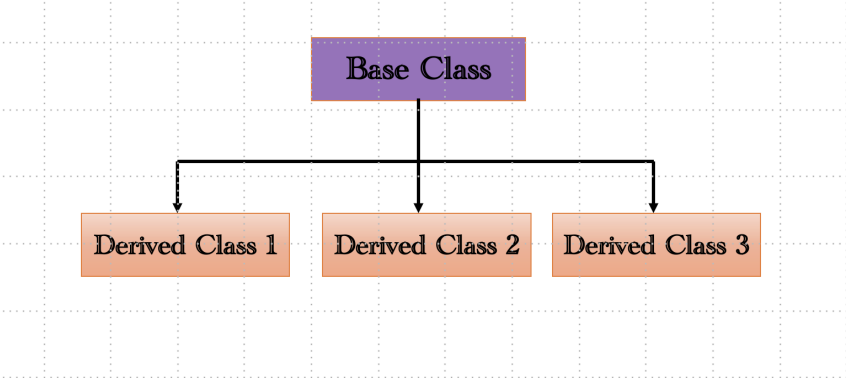
Creating multiple derived classes and a base class

Hierarchical Inheritance is specific form of inheritance where **multiple** derived classes inherit the features and properties of a **single** base class. This a type of inheritance that has a hierarchical structure of classes. Here a single base class can have multiple derived classes, and these derived classes can also have subclasses which can further inherit the features of these derived classes, forming a hierarchy multiple of classes in one structure. Most importantly, in hierarchical inheritance, all features that are common or peculiar within the derived classes are features that are inherited from the base class.

Hierarchical Inheritance: Example

Creating multiple derived classes and a base class

Let's illustrate the concept of hierarchical inheritance in which three derived classes inherit the features from a one base class.



Hierarchical Inheritance: Example

Creating multiple derived classes and a base class

Let's look at an example on how to create **hierarchical inheritance** program. We will also see how the objects of the derived classes are created and accessing the functions of both the base class and the derived classes.

```
#!/usr/bin/env python
"""
A program to demonstrate hierarchical inheritance
"""
# creating base class
class Base():
    #defining function in side the base class
    def funB(self, base):
        print("Base class 1 function")
        self.base = base
        print("Base class 1 value: ", self.base)
```

Hierarchical Inheritance: Example

Creating multiple derived classes and a base class

```
12 # creating first derived class
13 class Derived1(Base):
14     # defining function inside derived class
15     def funD1(self, derived1):
16         print("Derived class 1 function")
17         self.derived1 = derived1
18         print("Derived class 1 value: ", self.derived1)
19 # creating second derived class
20 class Derived2(Base):
21     # defining function inside derived class
22     def funD2(self, derived2):
23         print("Derived class 2 function")
24         self.derived2 = derived2
25         print("Derived class 2 value: ", self.derived2)
```

Hierarchical Inheritance: Example

Creating multiple derived classes and a base class

```
26 # creating the third derived class
27 class Derived3(Base):
28     # defining function inside derived class
29     def funD3(self, derived3):
30         print("Derived class 3 function")
31         self.derived3 = derived3
32         print("Derived class 3 value: ", self.derived3)
33 # creatibg object of first derived class
34 obj1 = Derived1()
35 # accessing functions from derived classes
36 print("------- First dervived class --------")
37 obj1.funB(10)
38 obj1.funD1(15)
39 # creating object of second derived class
40 obj2 = Derived2()
```

Hierarchical Inheritance: Example

Creating multiple derived classes and a base class

```
41 # accessing functions from derived classes
42 print("------- Second derived class --------")
43 obj2.funB(105)
44 obj2.funD2(205)
45 # creating object of the third derived class
46 obj3 = Derived3()
47 # accessing functions from derived classes
48 print("------- Third dervied class --------")
49 obj3.funB(1010)
50 obj3.funD3(2020)
```

Hierarchical Inheritance: Solution

Creating multiple derived classes and a base class

Output

```
------- First dervived class --------
Base class 1 function
Base class 1 value:  10
Derived class 1 function
Derived class 1 value:  15
------- Second derived class --------
Base class 1 function
Base class 1 value:  105
Derived class 2 function
Derived class 2 value:  205
------- Third dervied class --------
Base class 1 function
Base class 1 value:  1010
Derived class 3 function
Derived class 3 value:  2020
```

Relationships between classes

Checking the relationships between defined classes

In order to check the relationships between the defined classes within our program, we use the **issubclass(sub, sup)** method. By using this method, we are simply checking whether the **first class** is a subclass (or derived) of the **second class**. If this condition is true, the program will return True; otherwise, it will return False.

Relationships between classes: Example

Checking the relationships between defined classes

Let's look at an example on how to check the **relationships** between classes in the program.

```
#!/usr/bin/env python
"""
A program to check the relationship between classes
"""
# creating classes
class FirstClass():
    def funFirst(self):
        print("This is the first Base class")
class SecondClass():
    def funSecond(self):
        print("This is the second Base class")
class DerivedClass(FirstClass, SecondClass):
    def checkFun(self):
        print("This is the Derived class")
```

Relationships between classes: Example

Checking the relationships between defined classes

```
15 # creating object of the derived class
16 obj = DerivedClass()
17 print("DerivedClass is sub class of SecondClass: ", issubclass(
       DerivedClass, SecondClass))
18 print("FirstClass is sub class of SecondClass:  ", issubclass(FirstClass
       , SecondClass))
19 print("DerivedClass is sub class of FirstClass: ", issubclass(
       DerivedClass, FirstClass))
20 print("SecondClass is sub class of FirstClass:  ", issubclass(
       SecondClass, FirstClass))
21 print("FirstClass is sub class of DerivedClass:  ", issubclass(
       FirstClass, DerivedClass))
22 print("SecondClass is sub class of DerivedClass:  ", issubclass(
       SecondClass, DerivedClass))
```

Relationships between classes: Solution

Checking the relationships between defined classes

Output

```
DerivedClass is sub class of SecondClass:  True
FirstClass is sub class of SecondClass:   False
DerivedClass is sub class of FirstClass:  True
SecondClass is sub class of FirstClass:   False
FirstClass is sub class of DerivedClass:   False
SecondClass is sub class of DerivedClass:   False
```

Relationships between classes and objects

Checking the relationships between defined classes and objects

To check for the relationship between objects and classes, we use the **isinstance()** method. Here is the first parameter , obj , is an instance of the second parameter, class, then the program will returns True and otherwise returns False.

Relationships between objects and classes: Example

Checking the relationships between objects and classes

Let's look at an example on how to check the **relationships** between objects and classes in the program.

```
#!/usr/bin/env python
"""
A program to implement the isinstance() of method
"""
# creating the classes
class FirstClass:
    def funFirstClass(self):
        print("First Base class")
class SecondClass:
    def funSecondClass(self):
        print("Second Base class")
```

Relationships between objects and classes: Example

Checking the relationships between objects and defined classes

```
12 class Derived:
13     def funDerived(self):
14         print("Derived class")
15 obj = Derived()
16 print("obj is instance of FirstClass: ", isinstance(obj,FirstClass))
17 print("obj is instance of SecondClass: ", isinstance(obj,SecondClass))
18 print("obj is instance of Derived: ", isinstance(obj,Derived))
```

Output

```
obj is instance of FirstClass:  False
obj is instance of SecondClass:  False
obj is instance of Derived:  True
```

Polymorphism in Python

Concept of Polymorphism

Polymorphism is a concept in object-oriented programming which refers to having many forms. The concept of Polymorphism allows us to use a single interface for inputting various data types, classes, and for a varying number of inputs. Plolymorphism can be described in Python in different or many ways. Polymor – Takes multiple forms. **Polymorphism** is a class that takes multiple forms. Its the process of treating two different types of object as if they are the same in the same way using interfaces. The term polymorphism refer to phenomenon whereby methods and operators can have the same name, but exhibit different behaviour.

Polymorphism in Python

Concept of Polymorphism

Polymor – Takes multiple forms. Polymorphism is a class that takes multiple forms. Its the process of treating two different types of object as if they are the same in the same way using interfaces. For instance, **Method overloading** is actually one example of polymorphism. It is an important feature of object-oriented programming languages. This refers, in general, to the phenomenon of having methods and operators with the same name performing different functions.

Polymorphism in Python

Concept of Polymorphism

Objects that are of more than one type is said to support polymorphic types. Polymorphism can be achieve as follows:

- operator overloading
- Method overloading
- Method overriding
- Type Polymorphism

Polymorphism in Operators

Polymorphism Operators

As you already know in Python programming, the plus (+) operator is frequently used for so many operations. This simply means it does not have a single application. This operator (+) is used to **perform arithmetic operation such as the addition of numbers or data types** such as integer, float and so on. Similarly, this operator (+) could also be used to **concatenate string data type**.

Polymorphism Operators: Example

Using plus operator in Polymorphism

Let's look at an example on how to illustrate the concept **polymorphism** using an operator (+) in the program below.

```
#!/usr/bin/env python
"""
A programme to demonstrate the concept of Polymorphism
"""
# example program
x = 19
y = 27
print("The sum of x and y is: ", y + x)
X = "INST0004"
Y = "Python"
Z = "Programming 2"
print(X+" "+Y+" " +Z)
```

Relationships between classes: Solution

Checking the relationships between defined classes

Output

```
The sum of x and y is:  46
INST0004 Python Programming 2
```

Polymorphism and Functions

Functions used in Polymorphism

So many built-in functions in Python works on variety of data types. For example the len() function can be used with the concept Polymorphism. In Python, the len() function is allowed to work with variety of data types. The len() function can operate with various data types such as list, tuple, set, and dictionary. It does return unique information about specific data types. Let's look at an example to help us understand this better.

Polymorphism Function: Example

Using len() function to illustrate Polymorphism

```
#!/usr/bin/env python
"""
A program to demonstrate function polymorphism
"""
# using the len() function
print(len("INST0004 Python Programming 2"))
print(len(["Danny", "Onah", "Daniel", "Dan"]))
print(len({"ID": 2935478, "Address": "London"}))
print(len("12345"))
```

Output

```
29
4
2
5
```

Polymorphism in Class Methods

Using class methods and Polymorphism

Python allows various classes to have the same name methods, here we could also use the principle of polymorphism in constructing class methods. We may create a for loop that could iterate over a tuple of objects in a class or program. **We can call the methods afterwards without any regard to the class type of each object created**. *The program will presume that each class has these methods*. Let's look at an example to help us understand this concept better.

Polymorphism in Class Methods: Example

Using class methods and Polymorphism

```
#!/usr/bin/env python
"""
A program to demonstrate polymorphism in class methods
"""
# polymorphism in classes
class FirstClass():
    def funFirst(self):
        print("Function 1 of first class")
    def funSecond(self):
        print("Function 2 of first class")
class SecondClass():
    def funFirst(self):
        print("Function 1 of second class")
    def funSecond(self):
        print("Function 2 of second class")
```

Polymorphism in Class Methods: Example

Using class methods and Polymorphism

```
16 # creating object of first class
17 obj1 = FirstClass()
18 # creating object of second class
19 obj2 = SecondClass()
20 # iterating over the objects
21 for i in (obj1, obj2):
22     i.funFirst()
23     i.funSecond()
```

Output

```
Function 1 of first class
Function 2 of first class
Function 1 of second class
Function 2 of second class
```

Method overriding

Polymorphism in inheritance

Polymorphism in Python allows us to identify methods in the child class (**derived class**) with the same name as the parent class's (**base class**) methods. As you can remember, in inheritance, the methods of the parent class are passed on to the child class. However, a child class method that has inherited properties from a parent class may be modified according to the program functionalities. This is very useful when the inherited methods or properties from the parent class does not suit the child class functions. In such instances, the method is **re-implemented** in the child class. Method overriding is the process of re-implementing a method in a child's class.

Method overriding: Example

Polymorphism in inheritance

```
#!/usr/bin/env python
"""
A program to demonstrate method overriding
"""
# Example of method overloading
class Employee:
    def base(self):
        print("This is the base class")
    def funct1(self): # superclass method to override
        print("Employee class")
class Department(Employee):
    def funct1(self):  # method overriding
        print("Department class")
class Office(Employee):
    def funct1(self):  # method overriding
        print("Office class")
```

Method overriding: Example

Polymorphism in inheritance

```
17 # creating employee object and invoking the functions
18 employee = Employee()
19 employee.base()
20 employee.funct1()
21
22 # creating department object and invoking the functions
23 print('-----------------------------------------------')
24 dept = Department()
25 dept.base()
26 dept.funct1()
27
28 # creating office object and invoking the functions
29 print('-----------------------------------------------')
30 office = Office()
31 office.base()
32 office.funct1()
```

Method overriding: Example

Polymorphism in inheritance

Output

```
This is the base class
Employee class
------------------------------------------------
This is the base class
Department class
------------------------------------------------
This is the base class
Office class
```

Method Overriding and Overloading

Difference between method overriding & overloading

There are a few common difference between method overriding and method overloading. Let's look at some of the difference:

- **Signatures:**
  - In method overriding, methods must have the same name and same signature.
  - In method overloading, methods must have the same name and different signatures.
- **Return type or print statement:**
  - In method overriding, the return type or print statement must be the same or co-variant.
  - In method overloading, the return type or print statement can or cannot be the same, but we just have to change the parameter.

Built-in Class Functions

Python built-in class functions

Let's look at a few built-in class functions.

- getattr **(obj, name, default):** This Python built-in function is used to get the object's attributes.
- setattr **(obj, name, value):** This is used to give a specific value to an object's attribute.
- delattr **(obj, name):** This is used to delete specific attribute.
- hasattr **(obj, name):** This built-in function returns True if the object has a particular attribute, otherwise returns False.

Built-in Class Functions: Example

Python built-in class functions

```
#!/usr/bin/env python
"""
A program to demonstrate the built-in class function
"""
class Employee:
    def __init__(self, Emp_name, Emp_id, Emp_age):
        self.Emp_name = Emp_name
        self.Emp_id = Emp_id
        self.Emp_age = Emp_age
# creating object of Employee class
employee = Employee("Danny", 129476, 26)

# displaying the Employee class information
print("\nEmployee Name: ", employee.Emp_name, "\nEmployee ID: ",
    employee.Emp_id,"\nEmployee Age:", employee.Emp_age)
```

Built-in Class Functions: Example

Python built-in class functions

```
15 # printing attribute name of the employee object
16 print("-------displaying using built-in functions-------")
17 print(getattr(employee, "Emp_name")) # notice the order of argument
18
19 # changing the value of the Emp_age attribute to 32
20 setattr(employee, "Emp_age", 32)
21
22 # display modified value of Emp_age
23 print(getattr(employee, "Emp_age"))
24
25 # prints True if the Employee class contains the attribute with Emp_id
26 print(hasattr(employee, "Emp_id"))
27
28 # deleting an attribute from the Employee class
29 delattr(employee, "Emp_id")
```

Built-in Class Functions: Example

Python built-in class functions

```
30 # printing deleted attribute will show an error message
31 print(employee.Emp_id)
```

Output

```
Employee Name:  Danny
Employee ID:  129476
Employee Age: 26
-------displaying using built-in functions--------
Danny
32
True
--------------------error message ---------------------------
Traceback (most recent call last):
File "<string>", line 31, in <module>
    print(employee.Emp_id)
AttributeError: 'Employee' object has no attribute 'Emp_id'
```

Built-in Class Attributes

Python built-in class attributes

In addition to the other attributes, a Python class has several built-in class attributes that provide information about the class.The built-in class attributes are below:

- **___dict___**: This class attribute is used to provide a dictionary with class namespace details.
- **___doc___**: This class attribute has a string for the documentation of a class.
- **___name___**: This class attribute is used to get the name of the class.
- **___module___**: This class attribute us to get to the module where this class is defined.
- **___bases___**: This class attribute has a tuple that includes all base classes.

Built-in Class Attributes: Example

Python built-in class attributes

```
#!/usr/bin/env python
"""
A program to demonstrate class attributes
"""

# creating the employee class
class Employee:
    def __init__(self, Emp_name, Emp_id, Emp_age):
        self.Emp_name = Emp_name
        self.Emp_id = Emp_id
        self.Emp_age = Emp_age

# creating a function to print some parameter list of the constructor
def show(self):
    print(self.Emp_name)
    print(self.Emp_age)
```

Built-in Class Functions: Example

Python built-in class functions

```
17 # creating the employee objects
18 obj = Employee("Elaine", 24589023, 31)
19
20 # printing the object using built-in class attributes
21 print(obj.__doc__)# display the class document
22 print(obj.__dict__) # display the employee object created
23 print(obj.__module__) # display the main module
24 print(show.__name__) # display the name of the function
25 print(Employee.__bases__) # display the employee class type
```

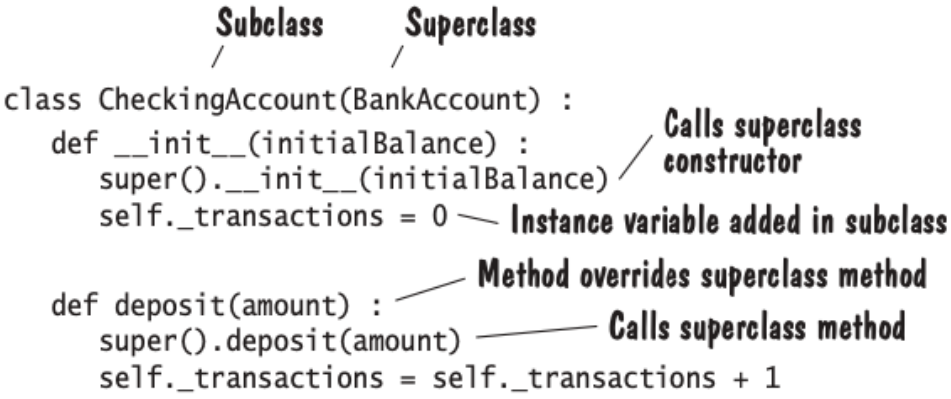
Output

```
None
{'Emp_name': 'Elaine', 'Emp_id': 24589023, 'Emp_age': 31}
__main__
show
(<class 'object'>,)
```

Key Points

Key points to remember about inheritance...

Inheritance




Summary

Summary

Let's revise the concepts of today's lecture

In this lecture we discuss the following:

- We discuss object-oriented programming.
- We discussed about the relationship between classes and objects.
- We looked at various concepts of object-oriented programming such as: inheritance, polymorphism
- We discussed about the concept of data abstraction/encapsulation and constructors.
- we looked at Python built-in class functions.

Further Reading

chapters to find further reading on the concepts

You can read further this week's lecture from the following textbook chapters:

- Python for Everyone (3/e) : **By Cay Horstmann & Rance Necaise** - *Chapter 10 Inheritance*
- Learning Python (5^{th} Edition): **By Mark Lutz** - *Chapter 4 Introducing Python Object Types, Chapter 26 OOP: The Big Picture, Chapter 31 Designing with Classes*
- Python Programming for absolute beginner (3rd Edition): **By Michael Dawson** - *Chapter 9 Object-Oriented Programming: The BlackJack Game*
- Python Crash Course - A hands-on, project-based introduction to programming (2nd Edition): **By Eric Matthes** - *Chapter 9 Classes*

Next Lecture 9

In week 9

Lecture 9: Sets, Sorting and Searching