Computing Laboratory

Object-oriented Programming with Python

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

January, 2021

Object-oriented Programming (OOP)

Objects are the basic runtime entities in an object-oriented system.
Objects encapsulate data and method.

Class and Object

Classes are the building blocks in OOP

A class is like a template that describes the behaviors/states that objects of its type support. It can be considered as a user-defined entity that can combine data and functionality together.

Class and Object

Classes are the building blocks in OOP

A class is like a template that describes the behaviors/states that objects of its type support. It can be considered as a user-defined entity that can combine data and functionality together.

A class contains:

- Data members known as *attributes* of the class
- Member functions known as *methods* of the class

Class and Object

Classes are the building blocks in OOP

A class is like a template that describes the behaviors/states that objects of its type support. It can be considered as a user-defined entity that can combine data and functionality together.

A class contains:

- Data members known as *attributes* of the class
- Member functions known as *methods* of the class

An object is an instance of a class

Empty class

Defining an empty class in Python:

```
class PythonProgrammer:
    pass
```

Note: `pass` is used as a placeholder with no errors getting created.

Object of an empty class

Creating an object of an empty class:

```
PP = PythonProgrammer() # Instantiation
```

Object of an empty class

Creating an object of an empty class:

```
PP = PythonProgrammer() # Instantiation
```

What is the output of the following code?

```
class PythonProgrammer:
    pass
PP = PythonProgrammer()
print(PP)
```

Object of an empty class

Creating an object of an empty class:

```
PP = PythonProgrammer() # Instantiation
```

What is the output of the following code?

```
class PythonProgrammer:
    pass
PP = PythonProgrammer()
print(PP)
```

Output:

```
<__main__.PythonProgrammer object at 0x7f3d6b7f4b70>
```

Object of an empty class

What is the output of the following code?

```
class PythonProgrammer:
    pass
PP1 = PythonProgrammer()
print(PP1)
PP2 = PythonProgrammer()
print(PP2)
PP1 == PP2
```

Object of an empty class

What is the output of the following code?

```
class PythonProgrammer:
    pass
PP1 = PythonProgrammer()
print(PP1)
PP2 = PythonProgrammer()
print(PP2)
PP1 == PP2
```

Output:

```
<__main__.PythonProgrammer object at 0x7f3d6b7f4da0>
<__main__.PythonProgrammer object at 0x7f3d6b7f4e48>
False
```

Note: Though `PP1` and `PP2` are instances of the same class, they represent two distinct objects in memory.

Using an empty class

Using an empty class in Python:

```
class PythonProgrammer:
    pass
PP1 = PythonProgrammer()
PP1.fullname = "Malay Bhattacharyya"
PP1.height = 5.6
PP1.age = 38
PP2 = PythonProgrammer()
PP2.fullname = "Mandar Mitra"
PP2.height = 5.7
PP2.age = 49
print("Average age:", (PP1.age + PP2.age) / 2)
```

Using an empty class

Using an empty class in Python:

```
class PythonProgrammer:
    pass
PP1 = PythonProgrammer()
PP1.fullname = "Malay Bhattacharyya"
PP1.height = 5.6
PP1.age = 38
PP2 = PythonProgrammer()
PP2.fullname = "Mandar Mitra"
PP2.height = 5.7
PP2.age = 49
print("Average age:", (PP1.age + PP2.age) / 2)
```

Output: `Average age: 43.5`

Standard class

Defining a standard class in Python:

```
class PythonProgrammer:
    fullname =  "Guido van Rossum"
    height = 5.66
    age = 64
    def feature(self):
        print("Creator of Python is: " + self.fullname)
```

- Attributes are always public and can be accessed using the dot (.) operator.

Standard class – The use of `self`

- Class methods must take the first parameter as `self`.

Standard class – The use of `self`

- Class methods must take the first parameter as `self`.
- Python provides a value to the `self` parameter not the user.

Standard class – The use of `self`

- Class methods must take the first parameter as `self`.
- Python provides a value to the `self` parameter not the user.
- The `self` parameter value is used for referencing. An object is linked to the class through this.

Object of a standard class

Creating an object of a standard class:

```
PP = PythonProgrammer() # Instantiation
```

Using a standard class

Using a standard class in Python:

```
class PythonProgrammer:
    fullname =  "Guido van Rossum"
    height = 5.66
    age = 64
    def feature(self):
        print("Creator of Python is: " + self.fullname)
PP = PythonProgrammer()
PP.feature() # Method via object
print("Height: " + str(PP.height)) # Attribute via object
print("Age: " + str(PP.age)) # Attribute via object
```

Using a standard class

Using a standard class in Python:

```
class PythonProgrammer:
    fullname =  "Guido van Rossum"
    height = 5.66
    age = 64
    def feature(self):
        print("Creator of Python is: " + self.fullname)
PP = PythonProgrammer()
PP.feature() # Method via object
print("Height: " + str(PP.height)) # Attribute via object
print("Age: " + str(PP.age)) # Attribute via object
```

Output:
```
Creator of Python is: Guido van Rossum
Height: 5.66
Age: 64
```

Immutable classes and objects

Immutable classes are Python classes whose objects can not be modified once created, and such objects are known as immutable objects. Any modification in immutable objects results into a new object.

Immutable classes and objects

Immutable classes are Python classes whose objects can not be modified once created, and such objects are known as immutable objects. Any modification in immutable objects results into a new object.

Objects of built-in types like int, float, bool, str, tuple, unicode are immutable in Python.

Immutable classes and objects

Immutable classes are Python classes whose objects can not be modified once created, and such objects are known as immutable objects. Any modification in immutable objects results into a new object.

Objects of built-in types like int, float, bool, str, tuple, unicode are immutable in Python.

Objects of built-in types like list, set, dict are mutable.

Encapsulation

An object = data + method

Constructor

A constructor is used to initialize the object's state

Constructors are methods that are useful for any kind of initialization you want to perform with the objects.

Constructor

A constructor is used to initialize the object's state

Constructors are methods that are useful for any kind of initialization you want to perform with the objects.

Note: A constructor is executed as soon as an object of a class is instantiated.

Constructor

Defining a constructor:

```
class PythonProgrammer:
    def __init__(self, fullname, height, age):
        self.fullname = fullname
        self.height = height
        self.age = age
```

Constructor

Defining a constructor:

```
class PythonProgrammer:
    def __init__(self, fullname, height, age):
        self.fullname = fullname
        self.height = height
        self.age = age
```

- `self.fullname = fullname` creates the attribute `fullname` and assigns the value of the parameter `fullname` to it.
- `self.height = height` creates the attribute `height` and assigns the value of the parameter `height` to it.
- `self.age = age` creates an attribute called age and assigns the value of the parameter age to it.

Constructor

Using a constructor:

```
class PythonProgrammer:
    def __init__(self, fullname, height, age):
        self.fullname = fullname
        self.height = height
        self.age = age
    def show(self):
        print(self.fullname)
PP = PythonProgrammer("Guido van Rossum", 5.66, 64)
PP.show()
```

Constructor

Using a constructor:

```
class PythonProgrammer:
    def __init__(self, fullname, height, age):
        self.fullname = fullname
        self.height = height
        self.age = age
    def show(self):
        print(self.fullname)
PP = PythonProgrammer("Guido van Rossum", 5.66, 64)
PP.show()
```

Output: `Guido van Rossum`

Constructor

Using a constructor:

```
class PythonProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def include(self, height):
        self.height = height
    def show(self):
        print(self.fullname)
        print(self.height)
PP = PythonProgrammer("Guido van Rossum")
PP.include(5.66)
PP.show()
```

Constructor

Using a constructor:

```
class PythonProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def include(self, height):
        self.height = height
    def show(self):
        print(self.fullname)
        print(self.height)
PP = PythonProgrammer("Guido van Rossum")
PP.include(5.66)
PP.show()
```

Output:
```
Guido van Rossum
5.66
```

Constructor

Using a constructor:

```
class PythonProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
        # We cannot write return(self.fullname) here
    def include(self, height):
        self.height = height
        return(self.height)
    def send(self):
        return(self.fullname)
PP = PythonProgrammer("Guido van Rossum")
print(PP.send())
print(PP.include(5.66))
```

Constructor

Using a constructor:

```
class PythonProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
        # We cannot write return(self.fullname) here
    def include(self, height):
        self.height = height
        return(self.height)
    def send(self):
        return(self.fullname)
PP = PythonProgrammer("Guido van Rossum")
print(PP.send())
print(PP.include(5.66))
```

Output:
```
Guido van Rossum
5.66
```

Polymorphism

Polymorphism is the condition of occurrence in different forms

Python allows polymorphism in many different forms:

- At the operator level (also known as operator overloading)
- At the function level (also known as function overloading)
- At the class level (also known as method overloading)

Polymorphism

Example of operator overloading:

```
i, j = 2000, 20
print(i + j)

s1 = "Computing"
s2 = "Lab"
print(s1 + " " + s2)
```

Polymorphism

Example of operator overloading:

```
i, j = 2000, 20
print(i + j)

s1 = "Computing"
s2 = "Lab"
print(s1 + " " + s2)
```

Output:
```
2020
Computing Lab
```

Polymorphism

Example of function overloading:

```
print(len("Python"))
print(len(["ver1", "ver2", "ver3"]))
print(len({"Creator": "Guido", "Nationality": "Dutch"}))
```

Polymorphism

Example of function overloading:

```
print(len("Python"))
print(len(["ver1", "ver2", "ver3"]))
print(len({"Creator": "Guido", "Nationality": "Dutch"}))
```

Output:
```
6
3
2
```

Polymorphism

Example of method overloading:

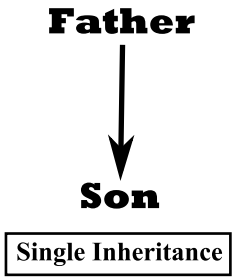
```
class PythonProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print("Creator of Python is: " + self.fullname)
class JavaProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print("Creator of Java is: " + self.fullname)
PP = PythonProgrammer("Guido van Rossum")
PP.show()
JP = JavaProgrammer("James Gosling")
JP.show()
```

Inheritance

Inheritance means acquiring the properties of another class

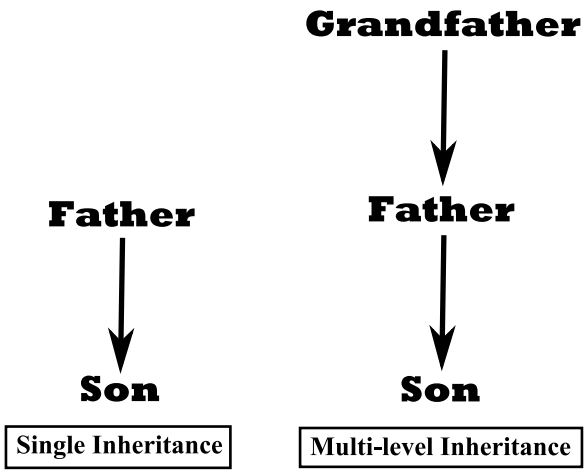
Inheritance

Inheritance means acquiring the properties of another class



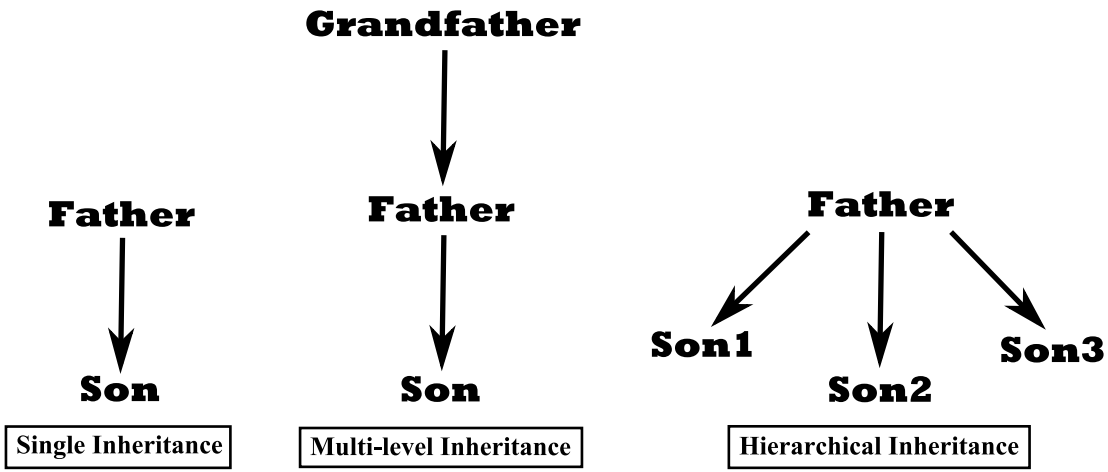
Inheritance

Inheritance means acquiring the properties of another class



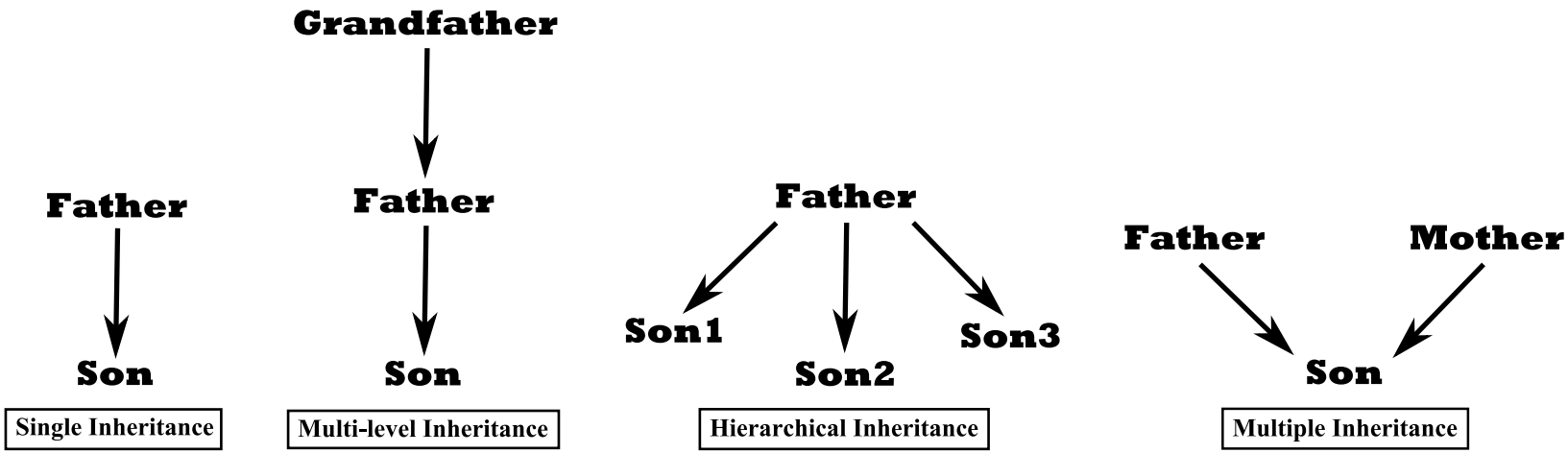
Inheritance

Inheritance means acquiring the properties of another class



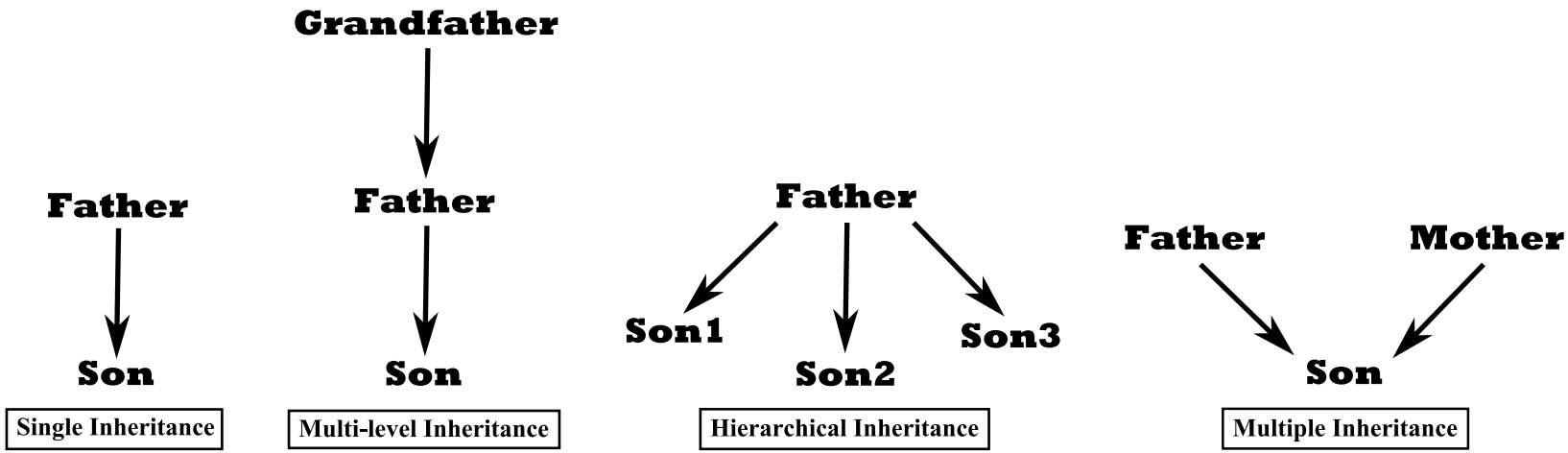
Inheritance

Inheritance means acquiring the properties of another class



Inheritance

Inheritance means acquiring the properties of another class



Note: subclass (child class/derived class) inherits the properties of the superclass (parent class/base class)

Inheritance

The syntax:

```
class SuperClass:
    Attributes of SuperClass
    Methods of SuperClass
class SubClass(SuperClass):
    Attributes of SubClass
    Methods of SubClass
```

Single inheritance

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    pass
CP = CProgrammer("Dennis Ritchie")
CP.show()
PP = PythonProgrammer("Guido van Rossum")
PP.show()
```

Single inheritance

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    pass
CP = CProgrammer("Dennis Ritchie")
CP.show()
PP = PythonProgrammer("Guido van Rossum")
PP.show()
```

Output: Dennis Ritchie is a C programmer
Guido van Rossum is a C programmer

Multi-level inheritance

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    pass
class SmartProgrammer(PythonProgrammer):
    pass
SP = SmartProgrammer("Guido van Rossum")
SP.show()
```

Multi-level inheritance

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    pass
class SmartProgrammer(PythonProgrammer):
    pass
SP = SmartProgrammer("Guido van Rossum")
SP.show()
```

Output: `Guido van Rossum is a C programmer`

Hierarchical inheritance

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    pass
class JavaProgrammer(CProgrammer):
    pass
PP = PythonProgrammer("Guido van Rossum")
PP.show()
JP = JavaProgrammer("James Gosling")
JP.show()
```

Hierarchical inheritance

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    pass
class JavaProgrammer(CProgrammer):
    pass
PP = PythonProgrammer("Guido van Rossum")
PP.show()
JP = JavaProgrammer("James Gosling")
JP.show()
```

Output:
```
Guido van Rossum is a C programmer
James Gosling is a C programmer
```

Multiple inheritance

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class SmartProgrammer():
    def credit(self):
        print("He is smart!!!")
class PythonProgrammer(CProgrammer, SmartProgrammer):
    pass
PP = PythonProgrammer("Guido van Rossum")
PP.show()
PP.credit()
```

Multiple inheritance

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class SmartProgrammer():
    def credit(self):
        print("He is smart!!!")
class PythonProgrammer(CProgrammer, SmartProgrammer):
    pass
PP = PythonProgrammer("Guido van Rossum")
PP.show()
PP.credit()
```

Output: Guido van Rossum is a C programmer
He is smart!!!

Method overriding in inheritance

Priority of a method in the subclass is always more than in the superclass. This is reflected through method overriding.

Method overriding in inheritance

Priority of a method in the subclass is always more than in the superclass. This is reflected through method overriding.

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    def show(self):
        print(self.fullname + " is a Python programmer")
PP = PythonProgrammer("Guido van Rossum")
PP.show()
```

Method overriding in inheritance

Priority of a method in the subclass is always more than in the superclass. This is reflected through method overriding.

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    def show(self):
        print(self.fullname + " is a Python programmer")
PP = PythonProgrammer("Guido van Rossum")
PP.show()
```

Output: `Guido van Rossum is a Python programmer`

Attribute overriding in inheritance

Priority of an attribute in the subclass is always more than in the superclass. This is reflected through attribute overriding.

Attribute overriding in inheritance

Priority of an attribute in the subclass is always more than in the superclass. This is reflected through attribute overriding.

```
class CProgrammer:
    status = 'inactive'
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    status = 'active'
PP = PythonProgrammer("Guido van Rossum")
PP.show()
print("He is " + PP.status)
```

Attribute overriding in inheritance

Priority of an attribute in the subclass is always more than in the superclass. This is reflected through attribute overriding.

```
class CProgrammer:
    status = 'inactive'
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    status = 'active'
PP = PythonProgrammer("Guido van Rossum")
PP.show()
print("He is " + PP.status)
```

Output: `Guido van Rossum is a Python programmer`
`He is active`

Checking inheritance

The built-in functions `isinstance()` and `issubclass()` can be used for checking inheritances.

- `isinstance(O, A)` returns `True` if the object `O` is an instance of the class `A` or other classes derived from it.
- `issubclass(X, A)` returns `True` if the class `X` is a subclass of the class `A`.

Checking inheritance

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    pass
CP = CProgrammer("Dennis Ritchie")
PP = PythonProgrammer("Guido van Rossum")
print(isinstance(CP, PythonProgrammer))
print(isinstance(PP, CProgrammer))
```

Checking inheritance

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    pass
CP = CProgrammer("Dennis Ritchie")
PP = PythonProgrammer("Guido van Rossum")
print(isinstance(CP, PythonProgrammer))
print(isinstance(PP, CProgrammer))
```

Output: False
True

Checking inheritance

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    pass
CP = CProgrammer("Dennis Ritchie")
PP = PythonProgrammer("Guido van Rossum")
print(issubclass(CProgrammer, PythonProgrammer))
print(issubclass(PythonProgrammer, CProgrammer))
```

Checking inheritance

```
class CProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def show(self):
        print(self.fullname + " is a C programmer")
class PythonProgrammer(CProgrammer):
    pass
CP = CProgrammer("Dennis Ritchie")
PP = PythonProgrammer("Guido van Rossum")
print(issubclass(CProgrammer, PythonProgrammer))
print(issubclass(PythonProgrammer, CProgrammer))
```

Output: False
True

Dunder/Magic/Special methods in Python

- `__init__()` for object initialization
- `__call__()` for callable objects
- `__repr__()` for object representation
- `__str__()` for object representation
- `__add__()` for operator overloading
- `__eq__()` for operator overloading
- `__lt__()` for operator overloading
- `__getitem__()` for iteration
- `__len__()` for iteration
- `__reversed__()` for iteration
- `__enter__()` for context manager support
- `__exit__()` for context manager support

Dunder/Magic/Special methods in Python

- `__init__()` for object initialization
- `__call__()` for callable objects
- `__repr__()` for object representation
- `__str__()` for object representation
- `__add__()` for operator overloading
- `__eq__()` for operator overloading
- `__lt__()` for operator overloading
- `__getitem__()` for iteration
- `__len__()` for iteration
- `__reversed__()` for iteration
- `__enter__()` for context manager support
- `__exit__()` for context manager support

Note: *Dunder* signifies *Double Underscores*.

Dunder/Magic/Special methods in Python

```
class PythonProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def __repr__(self):
        return 'Items: {}'.format(self.fullname.split())
    def show(self):
        print(self.fullname)
PP = PythonProgrammer("Guido van Rossum")
print(PP)
PP.show()
```

Dunder/Magic/Special methods in Python

```
class PythonProgrammer:
    def __init__(self, fullname):
        self.fullname = fullname
    def __repr__(self):
        return 'Items: {}'.format(self.fullname.split())
    def show(self):
        print(self.fullname)
PP = PythonProgrammer("Guido van Rossum")
print(PP)
PP.show()
```

Output: Items: ['Guido', 'van', 'Rossum']
Guido van Rossum