python for Computational Problem Solving - pCPS - OOP in python
Lecture Slides - Class #51

Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University

python for Computational Problem Solving Syllabus

Unit V: Object Oriented Programming - 10 Hours

- OOP in python
- classes and objects
  - inheritance
  - polymorphism.
- Error handling & Exceptions - try, except and raise
- exception propagation

OOP - Polymorphism in python

OOP - Polymorphism in python

- **Polymorphism** is taken from the Greek words **Poly (many)** and **morphism (forms)**.
- **Polymorphism** means that the same function name can be used for different types.
- **Polymorphism** makes programming more intuitive and easier.
- A **child** class **inherits** all the **methods** from the **parent** class.
- During **Inheritance** in some situations, the **method** inherited from the **parent** class **doesn't** quite **fit** into the **child** class, resulting in **re-implementing** the method in the child class.
- There are different **ways** to use **polymorphism** in **python**.
- We can use **different function**, **class methods** or **objects** to define **polymorphism**.

OOP - Function and Objects Polymorphism in python

- We can create a **function** that can take **any object**, allowing for **polymorphism**

```
# Polymorphism with Functions and Objects

class Paalak():

    def type(self):
        print('Vegetable')

    def Color(self):
        print('Green')

class WaterMelon():

    def type(self):
        print('Fruit')

    def Color(self):
        print('Green from Outside Red from Inside')

def Brain(Object):
    Object.type()
    Object.Color()

Object = Paalak()
Brain(Object)
print('--------------------')
Object = WaterMelon()
Brain(Object)
```

```
Vegetable
Green
--------------------
Fruit
Green from Outside Red from Inside
```

OOP - Polymorphism with Class Methods in python

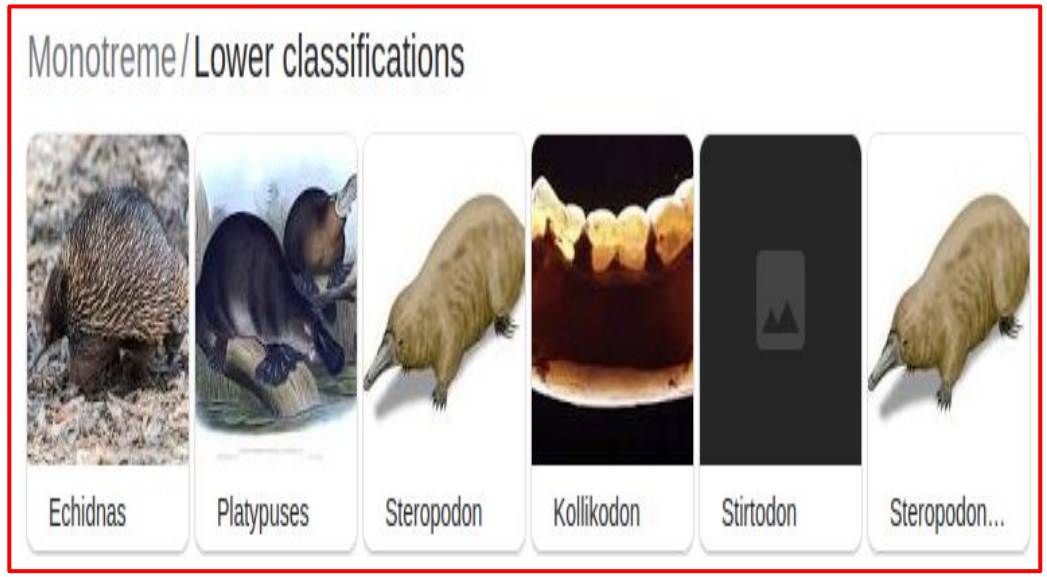
- **python** uses **two different** class **types** in the **same way** as **function** polymorphism.
- Firstly we have to create a for loop that **iterates** through a **tuple** of **objects**.
- Subsequently, we have to **call** the **methods without** being **concerned** about its **class** type.
- We **assume** that these **methods** actually **exist** in each class.

```
# Polymorphism with class Methods
class Karnataka():
    def Capital(self):
        print('Bengaluru')

    def Language(self):
        print('Kannada')

class TamilNadu():
    def Capital(self):
        print('Chennai')

    def Language(self):
        print('Tamil')

class Maharashtra():
    def Capital(self):
        print('Mumbai')

    def Language(self):
        print('Marathi')

O1 = Karnataka()
O2 = TamilNadu()
O3 = Maharashtra()
for State in (O1,O2,O3):
    State.Capital()
    State.Language()
    print('--------------------------')
```

```
Bengaluru
Kannada
--------------------------
Chennai
Tamil
--------------------------
Mumbai
Marathi
```

OOP - Polymorphism with Inheritance in python

- **Polymorphism** in python **defines methods** in the **child** class that have the **same name** as the **methods** in the **parent** class.
- In **Inheritance**, we already know that the **child** class **inherits** the **methods** from the **parent** class.
- It is **possible** to **modify** a **method** in a **child** class that it has inherited from the **parent** class.
- This is **mostly used** in **cases** where the **method inherited** from the parent class **doesn't fit** the child class.
- This **process** of **re-implementing** a method in the child class is known as **Method Overriding**.



OOP - Polymorphism with Inheritance in python

- **Polymorphism** in python **defines methods** in the **child** class that have the **same name** as the **methods** in the **parent** class.
- In **Inheritance**, we already know that the **child** class **inherits** the **methods** from the **parent** class.
- It is **possible** to **modify** a **method** in a **child** class that it has inherited from the **parent** class.
- This is **mostly used** in **cases** where the **method inherited** from the parent class **doesn't fit** the child class.
- This **process** of **re-implementing** a method in the child class is known as **Method Overriding**.

```
class Mammal():
    def Intro(self):
        print('Mammals have hair or fur; are warm-blooded; Most of them Cannot Fly')
    def Flight(self):
        print('It is a non Flying Mammal')
    def Eggs(self)    :
        print('Mammals mostly do not Lay Eggs')

class Cheetah(Mammal):
    def Intro(self):
        print('Cheetahs are Mammals covered almost entirely with small black spots on a background')
    def Flight(self):
        print('Cheetah is a non Flying Mammal')
    def Eggs(self)    :
        print('Cheetah do not Lay Eggs')

class Bat(Mammal):
    def Intro(self):
        print('Bats are mammals of the order Chiroptera.')
    def Flight(self):
        print('Bat is a Flying Mammal')
    def Eggs(self)    :
        print('Bats do not Lay Eggs')

class Platypus(Mammal):
    def Intro(self):
        print('The Platypus is a Monotreme mammal found only in Australia.')
    def Flight(self):
        print('Platypus is a non Flying Mammal')
    def Eggs(self)    :
        print('Platypus Lay Eggs')

O1 = Mammal()
O2 = Cheetah()
O3 = Bat()
O4 = Platypus()
O1.Intro()
O2.Flight()
O3.Eggs()
O4.Eggs()
```

```
Mammals have hair or fur; are warm-blooded; Most of them Cannot Fly
Cheetah is a non Flying Mammal
Bats do not Lay Eggs
Platypus Lay Eggs
```

OOP - Method Resolution Order in python

- **Method resolution Order describes** the **search path** of the **class** which **python uses** to **get** the **appropriate method** in **classes** that **contain** the **multi-inheritance**

- The **method** or **attributes** is explored in the **current class**, if the **method** is **not present** in the **current class**, the search **moves** to the **parent** classes, and **so on**

```
class A:
    def f1(self):
        print('Parent Class A')

class B(A):
    def f1(self):
        print('Class B')

class C(A):
    def f1(self):
        print('Class C')

O1 = A()
O2 = B()
O3 = C()

O1.f1()
O2.f1()
O3.f1()
```

```
Parent Class A
Class B
Class C
```

OOP - super in python

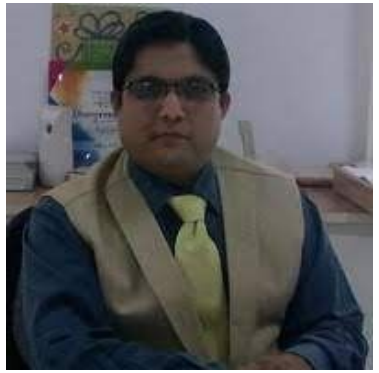
- The **super() builtin** returns a **proxy** object i.e, **temporary object** of the **superclass** that **allows** us to **access** methods of the **base** class.
- In python, **super()** has **two** major **use cases**:
  - **Allows** us to **avoid** using the **base** class **name explicitly**
  - **Working** with Multiple **Inheritance**
- **Technically**, super() **doesn't** return a **method,** but rather returns a **proxy object**.
- This is an **object** that **delegates** calls to the **correct** class **methods** without making an **additional object** in order **to do so**.

```
class Mammal(object):
  def __init__(self, Name):
    print(Name, 'is a warm-blooded animal.')

  def F1(self):
    print('Parent Class')

class Cat(Mammal):
  def __init__(self):
    print('Cat has four legs.')
    super().__init__('Cat')

  def F1(self):
    print('Cat Class')
    super().F1()


O1 = Cat()
O1.F1()
```

```
Cat has four legs.
Cat is a warm-blooded animal.
Cat Class
Parent Class
```

End of class #51

Thank you



Nitin V Pujari
Faculty, Computer Science
Dean - IQAC, PES University
nitin.pujari@pes.edu

For Course Digital Deliverables visit www.pesuacademy.com